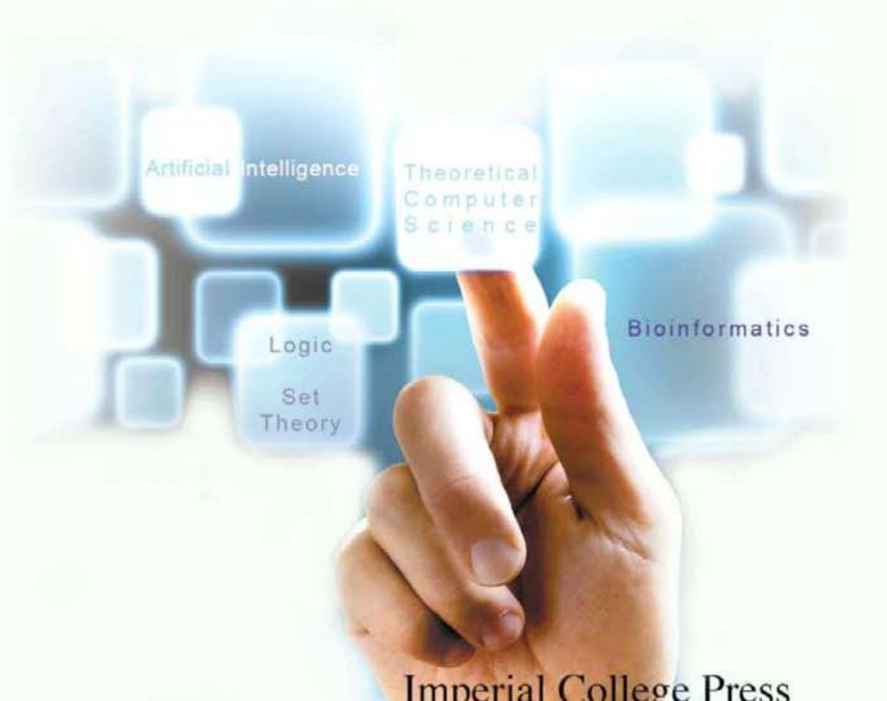
Mathematics, Computing, Language, and Life:
Frontiers in Mathematical Linguistics and Language Theory

Vol. 2

Edited by
Carlos Martín-Vide

Scientific Applications of Language Methods



Imperial College Press

Scientific Applications of Language Methods

Mathematics, Computing, Language, and Life:
Frontiers in Mathematical Linguistics and Language Theory

Vol. 2

Scientific Applications of Language Methods

Edited by

Carlos Martín-Vide

Universitat Rovira i Virgili, Spain

ICP

Imperial College Press

Published by

Imperial College Press
57 Shelton Street
Covent Garden
London WC2H 9HE

Distributed by

World Scientific Publishing Co. Pte. Ltd.

5 Toh Tuck Link, Singapore 596224

USA office: 27 Warren Street, Suite 401-402, Hackensack, NJ 07601

UK office: 57 Shelton Street, Covent Garden, London WC2H 9HE

British Library Cataloguing-in-Publication Data
A catalogue record for this book is available from the British Library.

Mathematics, Computing, Language, and Life: Frontiers in Mathematical Linguistics and Language Theory — Vol. 2
SCIENTIFIC APPLICATIONS OF LANGUAGE METHODS

ISBN-13 978-1-84816-544-1
ISBN-10 1-84816-544-7

Printed in Singapore.

Preface

Language theory, as originated from Chomsky's seminal work in the fifties last century and in parallel to Turing-inspired automata theory, was first applied to natural language syntax within the context of the first unsuccessful attempts to achieve reliable machine translation prototypes. After this, the theory proved to be very valuable in the study of programming languages and the theory of computing.

In the last 15–20 years, language and automata theory has experienced quick theoretical developments as a consequence of the emergence of new interdisciplinary domains and also as the result of demands for application to a number of disciplines.

Language methods (i.e. formal language methods) have been applied to a variety of fields, which can be roughly classified as:

- Computability and complexity,
- Natural language processing,
- Artificial intelligence, cognitive science, and programming,
- Bio-inspired computing and natural computing,
- Bioinformatics.

The connections of this broad interdisciplinary domain with other areas include: computational linguistics, knowledge engineering, theoretical computer science, software science, molecular biology, etc.

This volume gives just a few examples of the sort of research involved in this framework, with the intention to reflect the spirit of the whole book series.

Carlos Martín-Vide

This page is intentionally left blank

Contents

1. Descriptional Complexity — An Introductory Survey 1

M. Holzer and M. Kutrib

1.1 Introduction 1
1.2 Descriptional Systems and Complexity Measures 3
1.3 Measuring Sizes 6
1.4 Measuring Resources 33
1.5 Non-Recursive Trade-Offs 39
References 51

2. Classifying All Avoidable Sets of Partial Words of Size Two 59

F. Blanchet-Sadri, B. Blakeley, J. Gunter, S. Simmons and E. Weissenstein

2.1 Introduction 60
2.2 Preliminaries 62
2.3 Unavoidable Sets of Partial Words of Size Two 64
2.4 Canonical Forms 70
2.5 The Answer to the Conjectures 74
2.6 The Classification 95
2.7 Conclusion 99
References 100

3. On Glushkov $\mathbb{K}$-graphs 103

P. Caron and M. Flouret

3.1 Introduction 103
3.2 Definitions 105

3.3 Acyclic Glushkov WFA Properties 114
3.4 Glushkov $\mathbb{K}$-graph with Orbits 124
3.5 Algorithm for Orbit Reduction 127
3.6 Conclusion 131
References 131

4. Natural Language Dictionaries Implemented as Finite Automata 133

J. Daciuk, J. Piskorski and S. Ristov

4.1 Dictionaries as Finite Automata 133
4.2 Automata as Mappings 149
4.3 Construction Methods 165
4.4 Internal Structure and Compression 187
References 200

5. Tree-Language Based Querying of Hierarchically Structured and Semi-Structured Data 205

A. Berlea

5.1 Introduction 205
5.2 Preliminaries 207
5.3 Regular Forest Languages 217
5.4 Grammar Queries 230
5.5 Practical Application: A Pattern Language for XML Querying 259
5.6 Online Querying 275
5.7 Summary and Outlook 294
5.8 Proofs of Theorems 295
References 306

6. Quotient Monoids and Concurrent Behaviours 313

R. Janicki, J. Kleijn and M. Koutny

6.1 Introduction 314
6.2 Preliminaries 318
6.3 Partial Orders and Order Structures 326
6.4 Mazurkiewicz Traces 334
6.5 Comtraces 337
6.6 Generalised Comtraces 354
6.7 Elementary Net Systems 360

6.8 EN-systems with Inhibitor and Mutex Arcs 372
6.9 Concluding Remarks 382
References 383

7. Correction Queries in Active Learning 387

C. Tîrnăucă

7.1 Introduction 387
7.2 Preliminaries 389
7.3 Learning Models 391
7.4 Learning with Correction Queries 393
7.5 Polynomial Time Learning with Correction Queries 403
References 416

8. Applications of Grammatical Inference in Software Engineering: Domain Specific Language Development 421

M. Mernik, D. Hrnčič, B. R. Bryant and F. Javed

8.1 Introduction 422
8.2 Analysis Phase of DSL Development 424
8.3 Design Phase of DSL Development 426
8.4 Grammatical Inference and Language Design 429
8.5 Case Study 445
8.6 Related Work 448
8.7 Conclusion 452
References 453

9. Small Size Insertion and Deletion Systems 459

A. Alhazov, A. Krassovitskiy, Y. Rogozhin and S. Verlan

9.1 Introduction 459
9.2 Definitions 461
9.3 Basic Simulation Principles 466
9.4 Insertion-Deletion Systems with Rules of Small Size . . . 470
9.5 Context-Free Insertion-Deletion Systems 472
9.6 One-Sided Contextual Insertion-Deletion Systems 481
9.7 Pure Insertion Systems 497
9.8 Graph-Controlled Insertion-Deletion Systems 503
9.9 Graph-Controlled Insertion-Deletion Systems with Priorities 514

9.10 Bibliographical Remarks 520
References 521

10. Accepting Networks of Evolutionary Word and Picture Processors: A Survey 525

F. Manea, C. Martín-Vide and V. Mitrana

10.1 Introduction 525
10.2 Basic Definitions 527
10.3 Computational Power 536
10.4 Universal ANEPs and ANEPFCs 541
10.5 A Direct Simulation 544
10.6 Accepting Networks of Splicing Processors 545
10.7 Problem Solving with ANEPs/ANEPFCs 548
10.8 Accepting Networks of Picture Processors 549
References 558

11. Quantum Automata and Periodic Events 563

C. Mereghetti and B. Palano

11.1 Introduction 563
11.2 Preliminaries 566
11.3 Testing Periodicity on Unary 1qfa's 570
11.4 Synthesis of 1qfa's Inducing Periodic Events 572
11.5 Application to Periodic Languages 580
References 582

12. Soliton Circuits and Network-Based Automata: Review and Perspectives 585

M. Bartha and M. Krész

12.1 Introduction 586
12.2 Basic Concepts 588
12.3 Soliton Graphs and Automata 593
12.4 Elementary Decomposition of Soliton Graphs and Automata 597
12.5 Characterizing Soliton Automata 600
12.6 Complete Systems of Soliton Automata 610
12.7 Algorithms for Soliton Automata 615
12.8 Extensions of the Model and Further Research 623

12.9 Summary 627
References 628

13. Inferring Leadership Structure from Data on a Syntax Change in English 633

W. Garrett Mitchener

13.1 Introduction 634
13.2 The Available Data 639
13.3 Formulation of the Variable Influence Model 640
13.4 Fitting the *Do*-support Data 644
13.5 Results and Discussion 650
13.6 Future Directions 652
13.7 Conclusion 656
References 658

14. Weighted Automata Modeling for High Density Linkage Disequilibrium Mapping 663

T. Trang

14.1 Introduction 664
14.2 Biological Problem and Formulation 667
14.3 Fundamentals of Discrete Structures 676
14.4 Variable Length Finite Automata (VLFA) 682
14.5 Experimental Results on SNP Data 706
14.6 Conclusion 719
References 720

Author Index 723

Subject Index 725

Chapter 1

Descriptional Complexity — An Introductory Survey

Markus Holzer and Martin Kutrib
*Institut für Informatik, Universität Giessen,
Arndtstr. 2, 35392 Giessen, Germany,
E-mail:* {`holzer,kutrib`}`@informatik.uni-giessen.de`

The purpose of the paper is to give an introductory survey of the main aspects and results regarding the relative succinctness of different representations of languages, such as finite automata, regular expressions, pushdown automata and variants thereof, context-free grammars, and descriptional systems from a more abstract perspective. Basic properties of these descriptional systems and their size measures are addressed. The tradeoffs between different representations are either bounded by some recursive function, or reveal the phenomenon that the gain in economy of description can be arbitrary. In the latter case there is no recursive function serving as upper bound. We discuss developments relevant to the descriptional complexity of formal systems. The results presented are not proved but we merely draw attention to the big picture and some of the main ideas involved.

1.1 Introduction

In the field of theoretical computer science the term *descriptional complexity* has a well known meaning as it stands. Since the beginning of computer science descriptional complexity aspects of systems (automata, grammars, rewriting systems, etc.) have been a subject of intensive research [111]—since more than a decade the Workshop on "Descriptional Complexity of Formal Systems" (DCFS), formerly known as the Workshop on "Descriptional Complexity of Automata, Grammar, and Related Structures," has contributed substantially to the development of this field of research. The broad field of descriptional complexity of formal systems includes, but is not limited to, various measures of complexity of automata, grammars, languages and of related descriptional systems, succinctness of descriptional systems, trade-offs between complexity and mode of operation, etc., to mention a few.

The time has come to give an introductory survey of the main aspects and results regarding the relative succinctness of different representations of languages by finite automata, pushdown automata and variants thereof, context-free grammars, and descriptional systems from a more abstract perspective. Our tour mostly focuses on results that were found at the advent of descriptional complexity, for example, [52, 53, 59, 60, 98, 109, 112]. To this end, we have to unify the treatment of different research directions from the past. See also [38] for a recent survey of some of these results. Our write up obviously lacks completeness and it reflects our personal view of what constitute the most interesting relations of the aforementioned devices from a descriptional complexity point of view. In truth there is much more to the subject in question, than one can summarize here. For instance, the following current active research directions were not addressed in this summary: we skipped almost all results from the descriptional complexity of the operation problem which was revitalized in [137] after the dawn in the late 1970's. Moreover we will discuss anything on the subject of magic numbers a research field initiated in [73], and on the related investigations of determinization of nondeterministic finite automata accepting subregular languages done in [14] and others, and finally we left out the interesting field of research on the transition complexity of nondeterministic finite automata which has received a lot of attention during the last years [26, 46, 69, 70, 97].

In the next section, basic notions are given, and the basic properties of descriptional systems and their complexity measures are discussed and presented in a unified manner. A natural and important measure of descriptional complexity is the size of a representation of a language, that is, the length of its description. Section 1.3 is devoted to several aspects and results with respect to complexity measures that are recursively related to the sizes. A comprehensive overview of results is given concerning the question: how succinctly can a regular or a context-free language be represented by a descriptor of one descriptional system compared with the representation by an equivalent descriptor of the other descriptional system? Section 1.4 generalizes this point of view. Roughly speaking some, say, structural resource is fixed and its descriptional power is studied by measuring other resources. So, the complexity measures are not necessarily recursively related to the sizes of the descriptors. Here we stick with context-free grammars and subclasses as descriptional systems. Finally, Section 1.5 deals with the phenomenon of non-recursive trade-offs, that is, the trade-offs between representations of languages in different descriptional systems are not bounded by any recursive function. With other words, the gain in economy of description can be arbitrary. It turned out that most of the proofs appearing in the literature are basically relying on one of two fundamental schemes. These proof schemes are presented in a unified manner. Some important results are collected in a compilation of non-recursive trade-offs.

1.2 Descriptional Systems and Complexity Measures

We denote the set of nonnegative integers by $\mathbb{N}$, and the powerset of a set S by 2^S. In connection with formal languages, strings are called *words*. Let Σ^* denote the set of all words over a finite alphabet Σ. The *empty word* is denoted by λ, and we set $\Sigma^+ = \Sigma^* - \{\lambda\}$. For the *reversal of a word* w we write w^R and for its *length* we write $|w|$. A *formal language* L is a subset of Σ^*. In order to avoid technical overloading in writing, two languages L and L' are considered to be equal, if they differ at most by the empty word, that is, $L - \{\lambda\} = L' - \{\lambda\}$. Throughout the article two automata or grammars are said to be *equivalent* if and only if they accept or generate the same language. We use $\subseteq$ for *inclusions* and $\subset$ for *strict inclusions*.

We first establish some notation for descriptional complexity. In order to be general, we formalize the intuitive notion of a representation or description of a family of languages. A *descriptional system* is a collection of encodings of items where each item *represents* or *describes* a formal language. In the following, we call the items *descriptors*, and identify the encodings of some language representation with the representation itself. A formal definition is:

Definition 1.1. A *descriptional system* $\mathcal{S}$ is a set of finite descriptors, such that each descriptor $D \in \mathcal{S}$ describes a formal language $L(D)$, and the underlying alphabet $\text{alph}(D)$ over which D represents a language can be read off from D. The *family of languages represented (or described)* by $\mathcal{S}$ is $\mathscr{L}(\mathcal{S}) = \{ L(D) \mid D \in \mathcal{S} \}$. For every language L, the set $\mathcal{S}(L) = \{ D \in \mathcal{S} \mid L(D) = L \}$ is the set of its descriptors in $\mathcal{S}$.

Example 1.2. Pushdown automata (PDA) can be encoded over some fixed alphabet such that their input alphabets can be extracted from the encodings. The set of these encodings is a descriptional system $\mathcal{S}$, and $\mathscr{L}(\mathcal{S})$ is the family of context-free languages (CFL). $\square$

Now we turn to measure the descriptors. Basically, we are interested in defining a complexity measure as general as possible to cover a wide range of approaches, and in defining it as precise as necessary to allow a unified framework for proofs. So, we consider a *complexity measure* for a descriptional system $\mathcal{S}$ to be a total, recursive mapping $c : \mathcal{S} \to \mathbb{N}$. The properties total and recursive are straightforward.

Example 1.3. The family of context-free grammars is a descriptional system. Examples for complexity measures are the number productions appearing in a grammar, or the number of nonterminals, or the total number of symbols, that is, the length of the encoding. $\square$

Common notions as the *relative succinctness of descriptional systems* and our intuitive understanding of descriptional complexity suggest to consider the *size of descriptors*. From the viewpoint that a descriptional system is a collection of encoding strings, the length of the strings is a natural measure for the size. We denote it by length. In fact, we will use it to obtain a rough classification of different complexity measures. We distinguish between measures that (with respect to the underlying alphabets) are recursively related with length and measures that are not.

Definition 1.4. Let $\mathcal{S}$ be a descriptional system with complexity measure c. If there is a total, recursive function $g : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ such that $\mathsf{length}(D) \leq g(c(D), |\mathrm{alph}(D)|)$, for all $D \in \mathcal{S}$, then c is said to be an *s-measure*.

Example 1.5. Let us consider a widely accepted measure of complexity for finite automata, that is, their number of states, which is denoted by state. The formal definition of a finite automaton is given in the next section. Is state an s-measure? What makes a difference between the number of states (say, for deterministic finite automata (DFA)) and the lengths of encoding strings? The answer is obvious, encoding strings are over some fixed alphabet whereas the input alphabet of DFAs is not fixed *a priori*. The number of transitions depends on the input alphabet while the number of states does not. But states and transitions both determine the lengths of encoding strings. Nevertheless, when finite automata are addressed then, actually, a fixed given input alphabet is assumed tacitly. Since we regarded this aspect in the definition of s-measures, the answer to the first question is yes, the number of states of finite automata is an s-measure. To this end, given a deterministic finite automaton A, we may choose $g(\mathsf{state}(A), \text{alph}(A)) = k \cdot \mathsf{state}(A) \cdot \text{alph}(A)$, where $\mathsf{state}(A) \cdot \text{alph}(A)$ is the number of transition rules, and k is a mapping that gives the length of a rule dependent on the actual encoding alphabet, the number of states and the number of input symbols.

Similarly, we can argue for other types of finite automata as nondeterministic or alternating ones either with one-way or two-way head motion, etc. If the number of transition rules depends on the number of states and the number of input symbols (and, of course, on the type of the automaton in question), and the length of the rules is bounded dependent on the type of the automaton, then state is an s-measure. $\square$

Whenever we consider the relative succinctness of two descriptional systems $\mathcal{S}_1$ and $\mathcal{S}_2$, we assume the intersection $\mathscr{L}(\mathcal{S}_1) \cap \mathscr{L}(\mathcal{S}_2)$ to be non-empty.

Definition 1.6. Let $\mathcal{S}_1$ be a descriptional system with complexity measure c_1, and $\mathcal{S}_2$ be a descriptional system with complexity measure c_2. A total function $f : \mathbb{N} \to \mathbb{N}$, is said to be an *upper bound* for the increase in complexity when changing from a descriptor in $\mathcal{S}_1$ to an equivalent descriptor in $\mathcal{S}_2$, if for all $D_1 \in \mathcal{S}_1$ with $L(D_1) \in \mathscr{L}(\mathcal{S}_2)$ there exists a $D_2 \in \mathcal{S}_2(L(D_1))$ such that $c_2(D_2) \leq f(c_1(D_1))$.

If there is no recursive function serving as upper bound, the *trade-off is said to be non-recursive*. That is, whenever the trade-off from one descriptional system to another is non-recursive, one can choose an arbitrarily large recursive function f but the gain in economy of description eventually exceeds f when changing from the former system to the latter.

Definition 1.7. Let $\mathcal{S}_1$ be a descriptional system with complexity measure c_1, and $\mathcal{S}_2$ be a descriptional system with complexity measure c_2. A total function $f : \mathbb{N} \to \mathbb{N}$, is said to be a *lower bound* for the increase in complexity when changing from a descriptor in $\mathcal{S}_1$ to an equivalent descriptor in $\mathcal{S}_2$, if for infinitely many $D_1 \in \mathcal{S}_1$ with $L(D_1) \in \mathscr{L}(\mathcal{S}_2)$ there exists a *minimal* $D_2 \in \mathcal{S}_2(L(D_1))$ such that $c_2(D_2) \geq f(c_1(D_1))$.

1.3 Measuring Sizes

This section is devoted to several aspects of measuring descriptors with s-measures. A main field of investigation deals with the question: how succinctly can a language be represented by a descriptor of one descriptional system compared with the representation by an equivalent descriptor of the other descriptional system? An upper bound for the trade-off gives the maximal gain in economy of description, and conversely, the maximal blow-up (in terms of descriptional complexity) for simulations between the descriptional systems. A maximal lower bound for the trade-off terms the costs which are necessary in the worst cases.

1.3.1 *Descriptional Systems for Regular Languages*

Regular languages are represented by a large number of descriptional systems. So, it is natural to investigate the succinctness of their representations with respect to s-measures in order to optimize the space requirements. In this connection, many results have been obtained. On the other hand, the descriptional complexity of regular languages still offers challenging open problems. In the remainder of this subsection we collect and discuss some of these results and open problems.

1.3.1.1 *Finite Automata*

Here we measure the costs of representations by several types of finite automata in terms of the number of states, which is an s-measure by Example 1.5. Probably the most famous result of this nature is the simulation of nondeterministic finite automata by DFAs. Since several results come up with tight bounds in the exact number of states, it is advantageous to recall briefly the definitions of finite automata on which the results rely.

Definition 1.8. A *nondeterministic finite automaton* (NFA) is a quintuple $A = (Q, \Sigma, \delta, q_0, F)$, where Q is the finite set of *states*, Σ is the finite set of *input symbols*, $q_0 \in Q$ is the *initial state*, $F \subseteq Q$ is the set of *accepting states*, and $\delta : Q \times \Sigma \to 2^Q$ is the *transition function.*

A finite automaton is *deterministic* (DFA) if and only if $|\delta(q, a)| = 1$, for all states $q \in Q$ and letters $a \in \Sigma$. In this case we simply write $\delta(q, a) = p$ instead of $\delta(q, a) = \{p\}$ assuming that the transition function is a mapping $\delta : Q \times \Sigma \to Q$.

The *language accepted* by the finite automaton $A = (Q, \Sigma, \delta, q_0, F)$ is defined as $L(A) = \{\, w \in \Sigma^* \mid \delta(q_0, w) \cap F \neq \emptyset \,\}$, where the transition function δ is naturally extended to $\delta : Q \times \Sigma^* \to 2^Q$.

So, any DFA is *complete*, that is, the transition function is total, whereas it may be a partial function for NFAs in the sense that the transition function of nondeterministic machines may map to the empty set. Note that, therefore, a rejecting sink state is counted for DFAs, whereas it is not counted for NFAs. For further details we refer to [67].

It is well known that for any NFA one can always construct an equivalent DFA [119]. This so-called *powerset construction*, where each state of the DFA is associated with a subset of NFA states, turned out to be optimal, in general. That is, the bound on the number of states necessary for the construction is tight in the sense that for an arbitrary n there is always some n-state NFA which cannot be simulated by any DFA with strictly less than 2^n states [98, 109, 112]. So, NFAs can offer exponential savings in the number of states compared with DFAs.

Theorem 1.9 (NFA to DFA conversion). *Let $n \geq 1$ be an integer and A be an n-state NFA. Then 2^n states are sufficient and necessary in the worst case for a DFA to accept $L(A)$.*

For the particular cases of *finite* or *unary* regular languages the situation is significantly different. The conversion for finite languages over a binary alphabet was solved in [103] with a tight bound in the exact number of states. The general case of finite languages over a k-letter alphabet was shown in [124] with an asymptotically tight bound.

Theorem 1.10 (Finite NFA to DFA conversion). *Let $n \geq 1$ be an integer and A be an n-state NFA accepting a finite language over a binary alphabet. Then $2 \cdot 2^{\frac{n}{2}} - 1$ if n is even, and $3 \cdot 2^{\frac{n-1}{2}} - 1$ if n is odd, states are sufficient and necessary in the worst case for a DFA to accept $L(A)$. If A accepts a finite language over a k-letter alphabet, for $k \geq 3$, then $\Theta(k^{\frac{n}{1+\log_2 k}})$ states are sufficient and necessary in the worst case.*

Thus, for finite languages over a two-letter alphabet the costs are only $\Theta(2^{\frac{n}{2}})$. The situation is similar when we turn to the second important special case, the unary languages. The general problem of evaluating the costs of unary automata simulations was raised in [128], and has led to emphasize some relevant differences with the general case. For state complexity issues of unary finite automata *Landau's function*

$F(n) = \max\{ \operatorname{lcm}(x_1, \ldots, x_k) \mid x_1, \ldots, x_k \geq 1 \text{ and } x_1 + \cdots + x_k = n \}$, which gives the maximal order of the cyclic subgroups of the symmetric group on n elements, plays a crucial role. Here, lcm denotes the least common multiple. Since F depends on the irregular distribution of the prime numbers, we cannot expect to express $F(n)$ explicitly by n. In [89, 90] the asymptotic growth rate $\lim_{n\to\infty}(\ln F(n)/\sqrt{n \cdot \ln n}) = 1$ was determined, which for our purposes implies the (sufficient) rough estimate $F(n) \in e^{\Theta(\sqrt{n \cdot \ln n})}$. The following asymptotic tight bound on the unary NFA by DFA simulation was presented in [22, 23]. Its proof is based on a normalform (Chrobak normalform) for unary NFAs introduced in [22]. Each n-state unary NFA can be replaced by an equivalent $O(n^2)$-state NFA consisting of an initial deterministic tail and some disjoint deterministic loops, where the automaton makes only a single nondeterministic decision after passing through the initial tail, which chooses one of the loops.

Theorem 1.11 (Unary NFA to DFA conversion). *Let $n \geq 1$ be an integer and A be an n-state NFA accepting a unary language. Then $e^{\Theta(\sqrt{n \cdot \ln n})}$ states are sufficient and necessary in the worst case for a DFA to accept $L(A)$.*

For languages that are unary *and* finite in [103] it has been shown that nondeterminism does not help at all. Finite unary DFAs are up to one additional state as large as equivalent minimal NFAs. Moreover, it is well known that nondeterminism cannot help for all languages.

Example 1.12. Any NFA accepting the language

$$L_n = \{ w \in \{a, b\}^* \mid |w| = i \cdot n, \text{ for } i \geq 0 \},$$

for an integer $n \geq 1$, has at least n states, and L_n is accepted by an n-state DFA as well. $\square$

On the one hand, we have seen that for certain languages unlimited nondeterminism cannot help. On the other hand, for unary languages accepted by NFAs in Chrobak normalform, that is, by NFAs that make at most *one* nondeterministic step in every computation, one can achieve a trade-off which is strictly less than 2^n but still exponential. This immediately brings us to the question in which cases nondeterminism can help to represent a regular language succinctly. A model with very weak nondeterminism are deterministic finite automata with multiple entry states (NDFA) [33, 134]. Here the sole guess appears at the beginning of the computation, that is, by choosing one out of k initial states. So, the nondeterminism is not only limited in its amount but also in the situation at which it may appear. Converting an NDFA with k initial states into a DFA by the powerset construction shows immediately that any reachable state contains at most k states of the NDFA. This gives an upper bound for the conversion. In [66] it has been shown that this upper bound is tight.

Theorem 1.13 (NDFA to DFA conversion). *Let $n, k \geq 1$ be integers satisfying $k \leq n$, and A be an n-state NDFA with k entry states. Then $\sum_{i=1}^{k} \binom{n}{i}$ states are sufficient and necessary in the worst case for a DFA to accept $L(A)$.*

So, for $k = 1$ we obtain DFAs while for $k = n$ we are concerned with the special case that needs $2^n - 1$ states. Interestingly, NFAs can be exponentially concise over NDFAs. The following lower bound has been derived in [79].

Theorem 1.14 (NFA to NDFA conversion). *Let $n \geq 1$ be an integer and A be an n-state NFA. Then $\Omega(2^n)$ states are necessary in the worst case for a NDFA to accept $L(A)$.*

The concept of limited nondeterminism in finite automata is more generally studied in [82]. There, a bound on the number of nondeterministic steps allowed during a computation as well as on the maximal number of choices for every nondeterministic step is imposed. While the maximal number of choices is three, the bound on the number of steps is given by a function that depends on the number of states. This implies that in any computation the NFAs can make a finite number of nondeterministic steps only. But the situations at which nondeterminism appears are not restricted *a priori*. The order of magnitude of the functions considered is strictly less than the logarithm, that is, for a bounding function f we have $f \in o(\log)$. The upper bound for the costs of the conversion into a DFA follows from the powerset construction. Due to the restrictions any reachable state contains at most $3^{f(n)}$ states of the NFA. The next theorem summarizes this observation and the lower bound shown in [82].

Theorem 1.15 (Limited NFA to DFA conversion). *Let $n \geq 1$ be an integer, A be an n-state NFA, and $f : \mathbb{N} \to \mathbb{N}$ be function of order $o(\log)$. Then $\sum_{i=0}^{3^{f(n)}} \binom{n}{i}$ states are sufficient and $\sum_{i=0}^{2^{f(\sqrt{n})}} \binom{O(\sqrt{n})}{i}$ is a lower bound for the worst case state complexity for a deterministic finite automaton to accept $L(A)$.*

Note that the upper bound $\sum_{i=0}^{3^{f(n)}} \binom{n}{i}$ is of order $o(2^n)$, if $f(n) \in o(\log n)$. The precise bound for the conversion is an open problem.

Next, we turn to finite automata whose descriptional capacity is stronger than NFAs due to their additional resources. We start with alternating finite automata which have been developed in [21], and recall their definition from that paper. To this end, we identify the logical values *false* and *true* with 0 and 1 and write $\{0,1\}^Q$ for the set of finite functions from Q into $\{0,1\}$, and $\{0,1\}^{\{0,1\}^Q}$ for the set of Boolean formulas (functions) mapping $\{0,1\}^Q$ into $\{0,1\}$.

Definition 1.16. An *alternating finite automaton* (AFA) is a quintuple $A = (Q, \Sigma, \delta, q_0, F)$, where Q, Σ, q_0, and F are as for NFAs, and $\delta : Q \times \Sigma \to \{0,1\}^{\{0,1\}^Q}$ is the *transition function.* The transition function maps pairs of states and input symbols to Boolean formulas.

Before we define the language accepted by the AFA A we have to explain how a word is accepted. As the input is read (from left to right), the automaton "builds" a propositional formula, starting with the formula q_0, and on reading an input a, replaces every $q \in Q$ in the current formula by $\delta(q, a)$. The input is *accepted* if and only if the constructed formula on reading the whole input evaluates to 1 on substituting 1 for q, if $q \in F$, and 0 otherwise. This substitution defines a mapping from Q into $\{0,1\}$ which is called the *characteristic vector* f_A of A. Then the *language accepted* by the AFA A is defined as $L(A) = \{\, w \in \Sigma^* \mid w \text{ is accepted by } A \,\}$.

Example 1.17. [136] Let $A = (\{q_0, q_1, q_2\}, \{a, b\}, \delta, q_0, \{q_2\})$ be an AFA with transition function defined through

$$\begin{aligned} &\delta(q_0, a) = q_1 \wedge q_2, && \delta(q_0, b) = 0, \\ &\delta(q_1, a) = q_2, && \delta(q_1, b) = q_1 \wedge q_2, \\ &\delta(q_2, a) = \overline{q}_1 \wedge q_2, && \delta(q_2, b) = q_1 \vee \overline{q}_2. \end{aligned}$$

On input *aba* the propositional formula evolves as follows. Starting with q_0 after reading the first input symbol a the formula is $q_1 \wedge q_2$. After reading b we obtain $(q_1 \wedge q_2) \wedge (q_1 \vee \overline{q}_2)$, and after reading the last symbol a the formula $(q_2 \wedge (\overline{q}_1 \wedge q_2)) \wedge (q_2 \vee (\overline{\overline{q}_1 \wedge q_2}))$.

After substituting the characteristic vector, that is, 0 for q_0, q_1, and 1 for q_2 we have $(1 \wedge (\overline{0} \wedge 1)) \wedge (1 \vee (\overline{\overline{0} \wedge 1}))$ which evaluates to 1. Therefore, the input *aba* is accepted. □

It is worth mentioning that sometimes in the literature an equivalent definition of AFAs appears, where the state set is partitioned into existential and universal states.

At the same period as alternating finite automata were developed in [15] the so-called *Boolean automata* were introduced. Note, that several authors use the notation "alternating finite automata" but rely on the definition of Boolean automata. Though it turned out that both types are almost identical, there are differences with respect to the initial configurations. While for AFAs the computation starts with the fixed propositional formula q_0, a Boolean automaton starts with an arbitrary propositional formula. Clearly, this does not increase their computational capacities. However, it might make the following difference from a descriptional complexity point of view.

Lemma 1.18 (Boolean automata to AFA conversion). *Let $n \geq 1$ be an integer and A be an n-state Boolean automaton. Then $n + 1$ states are sufficient for an AFA to accept $L(A)$.*

In the first step of the simulation, the additional state of the AFA is used to derive the successors of the initial propositional formula of the Boolean automaton from the fixed initial propositional formula q_0 of the AFA. The additional state is unreachable afterwards. It is an open problem whether or not the additional state is really necessary, that is, whether the bound of $n + 1$ is tight. See [30] for more details on alternating finite automata having an initial state that is unreachable after the first step.

Next we consider the descriptional capacity of AFAs compared with NFAs and DFAs. The tight bound of 2^{2^n} states for the conversion of n-state AFAs into DFAs has already been shown in the fundamental papers [21] for AFAs and [15, 92] for Boolean automata.

Theorem 1.19 (AFA/Boolean automata to DFA conversion). *Let $n \geq 1$ be an integer and A be an n-state AFA or Boolean automaton. Then 2^{2^n} states are sufficient and necessary in the worst case for a DFA to accept $L(A)$.*

The original proofs of the upper bound rely on the fact that an AFA or a Boolean automaton can enter only finitely many internal situations, which are given by Boolean functions depending on n Boolean variables associated with the n states. The number of 2^{2^n} such functions determines the upper bound.

The proofs provide little insight in the descriptional capacity of AFAs compared with NFAs. In [30] constructions are presented that show how to convert an AFA into an equivalent nondeterministic finite automaton with multiple entry states (NNFA). Let $A = (Q, \Sigma, \delta, q_0, F)$ be an n-state AFA with $Q = \{q_0, q_1, \ldots, q_{n-1}\}$ and characteristic vector f_A. Then we consider the NNFA $A' = (\{0,1\}^Q, \Sigma, \delta', Q_0, \{f_A\})$, where the set of initial states is $Q_0 = \{\, u \in \{0,1\}^Q \mid u(q_0) = 1 \,\}$, and the transition function is defined by $\delta'(u, a) = \{\, u' \mid \delta(u', a) = u \,\}$, for all $u, u' \in \{0,1\}^Q$ and $a \in \Sigma$. So, the NNFA simulates the AFA by guessing the sequence of functions of the form $\{0,1\}^Q$ that appear during the evaluation of the propositional formula computed by the AFA in reverse order. Since there are 2^n such functions we obtain the upper bound stated in Theorem 1.20. Moreover, since the powerset construction works also fine for the determinization of NNFAs, the presented construction also reveals the upper bound for the AFA to DFA conversion already stated in Theorem 1.19. The construction for Boolean automata is derived from above by considering the initial Boolean formula f_0 of the Boolean automaton and to change the set of initial states of the NNFA accordingly. To this end, it suffices to define Q_0 to be $\{\, u \in \{0,1\}^Q \mid f_0(u(q_0), u(q_1), \ldots, u(q_{n-1})) = 1 \,\}$. From the construction we derive the upper bound of the next theorem.

Theorem 1.20 (AFA to NNFA conversion). *Let $n \geq 1$ be an integer and A be an n-state AFA or Boolean automaton. Then 2^n states are sufficient and necessary in the worst case for an NNFA to accept $L(A)$.*

The matching lower bound of Theorem 1.19 is shown in [21] for AFAs by witness languages in a long proof. Before we come back to this point for Boolean automata, we turn to an interesting aspect of AFAs and Boolean automata. One can observe that the construction of the simulating NNFA is backward deterministic [21]. So, the reversal of a language accepted by an n-state AFA or Boolean automaton is accepted by a *not necessarily complete* 2^n-state DFA which in turn can be simulated by a $(2^n + 1)$-state *complete* DFA. This result has significantly be strengthened in [92], where it is shown that the reversal of *every* n-state DFA language is accepted by a Boolean automaton with $\lceil \log_2 n \rceil$ states. With other words, *with restriction to reversals* of regular languages a Boolean automaton can *always* save exponentially many states compared to a DFA. The next theorem summarizes these results.

Theorem 1.21 (Reversed AFA to DFA conversion). *Let $n \geq 1$ be an integer and A be an n-state AFA or Boolean automaton. Then $2^n + 1$ states are sufficient and necessary in the worst case for a DFA to accept the reversal of $L(A)$. If the minimal DFA accepting the reversal of $L(A)$ does not have a rejecting sink state, then 2^n states are sufficient. Moreover, the reversal of every language accepted by an n-state DFA is accepted by a Boolean automaton with $\lceil \log_2 n \rceil$ states.*

The theorem leaves open whether the reversal of *every* n-state DFA language is also accepted by some *AFA* with $\lceil \log_2 n \rceil$ states. However, we know that $\lceil \log_2 n \rceil + 1$ states are sufficient for this purpose.

Now we are prepared to argue for the matching lower bound of Theorem 1.19 for Boolean automata in a simple way. It is well known that for any $m \geq 1$ there is an m-state DFA A such that any DFA accepting the reversal of $L(A)$ has 2^m states [92]. Setting $m = 2^n$ we obtain a 2^n-state DFA language $L(A)$ whose reversal is accepted by a Boolean automaton with n states by Theorem 1.21. On the other hand, the reversal of $L(A)$ takes at least 2^{2^n} states to be accepted deterministically.

Next we argue that the upper bound of Theorem 1.20 cannot be improved in general. To this end, let A be an n-state AFA or Boolean automaton such that any equivalent DFA has 2^{2^n} states. Let m be the minimal number of states for an equivalent NNFA. Since the NNFA can be simulated by a DFA with at most 2^m states, we conclude $2^m \geq 2^{2^n}$, that is, the NNFA has at least $m \geq 2^n$ states.

So far, we have only considered nondeterministic finite automata with multiple entry states. It is known that any such NNFA can be simulated by an NFA having one more state. The additional state is used as new sole initial state which is appropriately connected to the successors of the old initial states. On the other hand, in general this state is needed. For example, consider the language $\{ a^n \mid n \geq 0 \} \cup \{ b^n \mid n \geq 0 \}$ which is accepted by a 2-state NNFA but takes at least three states to be accepted by an NFA. Nevertheless, it is an open problem whether there are languages accepted by n-state AFAs or Boolean automata such that any equivalent NFA has at least $2^n + 1$ states. In [30] it is conjectured that this bound presented in the following theorem is tight.

Lemma 1.22 (AFA to NFA conversion). *Let $n \geq 1$ be an integer and A be an n-state AFA or Boolean automaton. Then $2^n + 1$ states are sufficient and 2^n is a lower bound for the worst case state complexity for an NFA to accept $L(A)$.*

We now direct our attention to the question whether alternation can always help to represent a regular language succinctly. We have seen already that nondeterminism cannot help for all languages. So, how about the worst case of the language representation by alternating finite automata? The situation seems to be more sophisticated. Theorem 1.21 says that for reversals of n-state DFA languages we can *always* achieve an exponential saving of states. Interestingly, this potential gets lost when we consider the n-state DFA languages itself (instead of their reversals). The next theorem and its corollary are from [93].

Theorem 1.23. *For every integer $n \geq 1$ there exists a minimal DFA A with n states such that any AFA or Boolean automaton accepting $L(A)$ has at least n states.*

The DFAs $A_n = (\{q_0, q_1, \ldots, q_{n-1}\}, \{a, b\}, \delta, q_1, F)$ witness the theorem for all integers $n \geq 2$, where $F = \{\, q_i \mid 0 \leq i \leq n-1 \text{ and } i \text{ even} \,\}$ and the transition function given by

$$\delta(q_i, a) = q_{(i+1) \bmod n} \quad \text{and} \quad \delta(q_i, b) = \begin{cases} q_i & \text{for } 0 \leq i \leq n-3 \\ q_{n-1} & \text{for } i \in \{n-2, n-1\}. \end{cases}$$

Each DFA A_n has the property that any DFA A_n' accepting the reversal of $L(A)$ has at least 2^n states. Moreover, A_n and A_n' both are minimal, complete and do not have a rejecting sink state [92]. Assume that $L(A)$ is accepted by some AFA or Boolean automaton with $m < n$ states. Then the reversal of $L(A)$ would be accepted by some DFA having at most 2^m states by Theorem 1.21. This is a contradiction since $2^m < 2^n$.

Corollary 1.24. *Let $n \geq 1$ be an integer and A be an n-state DFA such that any DFA accepting the reversal of $L(A)$ has at least 2^n states and no rejecting sink state. Then any AFA or Boolean automaton accepting $L(A)$ has at least n states.*

We have already seen that unary NFAs can be much more concise than DFAs, but yet not as much as for the general case. So, we next continue to draw that part of the picture with respect to alternating finite automata. In general, the simulation of AFAs by DFAs may cost a double exponential number of states. The unary case is cheaper. Since every unary language coincides trivially with its reversal, the upper bound of the following theorem is immediately derived from Theorem 1.21. The lower bound can be seen by considering the single word language $L_n = \{a^{2^n-1}\}$. For all $n \geq 1$, the language L_n is accepted by some minimal $(2^n + 1)$-state DFA and an n-state AFA. So, we derive the next theorem.

Theorem 1.25 (Unary AFA to DFA conversion). *Let $n \geq 1$ be an integer and A be an n-state AFA accepting a unary language. Then $2^n + 1$ states are sufficient and necessary in the worst case for a DFA to accept $L(A)$. If the minimal DFA does not have a rejecting sink state, then 2^n states are sufficient.*

Interestingly, to some extend for unary languages it does not matter in general, whether we simulate an AFA deterministically or nondeterministically. The tight bounds differ at most by one state. The upper bound of this claim follows since any DFA is also an NFA and NFAs are not necessarily complete. The lower bound is again witnessed by the single word languages L_n, which require 2^n states for any NFA accepting it.

Corollary 1.26 (Unary AFA to NFA simulation). *Let $n \geq 1$ be an integer and A be an n-state AFA accepting a unary language. Then 2^n states are sufficient and necessary in the worst case for an NFA to accept $L(A)$.*

Theorem 1.23 revealed that alternation cannot help to reduce the number of states of DFAs or NFAs in all cases. The same is true for nondeterministic simulations of DFAs in general and in the unary case. The latter can be seen by the unary languages $\{a^n\}^*$, for $n \geq 1$. However, for unary languages alternation does help. By Theorem 1.25 we know already that any AFA simulating an n-state DFA accepting a unary language has not less than $\lceil \log_2 n \rceil - 1$ states. Once more the unary single word languages L_n are witnesses that this saving can be achieved. This gives rise to the next theorem.

Theorem 1.27 (Unary DFA to AFA conversion). *Let $n \geq 1$ be an integer and A be an n-state DFA accepting a unary language. Then $\lceil \log_2 n \rceil - 1$ states are necessary for an AFA to accept $L(A)$. Moreover, there exists a minimal DFA A with n states accepting a unary language such that any minimal AFA accepting $L(A)$ has exactly $\lceil \log_2 n \rceil - 1$ states.*

Finally, we derive the *always possible* savings for unary NFA by AFA simulations as follows. Given some n-state NFA accepting a unary language, by Theorem 1.11 we obtain an equivalent DFA that has at most $e^{\Theta(\sqrt{n \cdot \ln n})} = 2^{\Theta(\sqrt{n \cdot \ln n})}$ states. Now Theorem 1.21 in combination with Lemma 1.18 says essentially that there is an equivalent AFA with $\Theta(\sqrt{n \cdot \ln n})$ states. In order to see that these savings are optimal in general, consider a unary n-state NFA such that any equivalent DFA must have $e^{\Theta(\sqrt{n \cdot \ln n})}$ states. Since the bound of Theorem 1.11 is tight such automata exist. Clearly, any equivalent AFA has at least $\Theta(\sqrt{n \cdot \ln n})$ states. Otherwise there would be an equivalent DFA with less than $e^{\Theta(\sqrt{n \cdot \ln n})}$ states by Theorem 1.25.

Theorem 1.28 (Unary NFA to AFA conversion). *Let $n \geq 1$ be an integer and A be an n-state NFA accepting a unary language. Then $\Theta(\sqrt{n \cdot \ln n})$ states are sufficient and necessary in the worst case for an AFA to accept $L(A)$.*

The justification of the second part of the theorem gives rise to the following corollary.

Corollary 1.29. *Let $n \geq 1$ be an integer and A be a unary n-state NFA such that any equivalent DFA has at least $e^{\Theta(\sqrt{n \cdot \ln n})}$ states. Then any AFA accepting $L(A)$ has at least $\Theta(\sqrt{n \cdot \ln n})$ states.*

The final resource we investigate here with respect to its descriptional capacity is two-way head motion. We denote deterministic and nondeterministic finite automata that may move their head to the right as well as to the left by 2DFA and 2NFA. We first sketch the development of results for general regular languages. Then we turn to unary languages again.

Concerning the simulation of 2DFA by DFA an $\Theta(n^n)$ asymptotically tight bound was shown in [127]. Moreover, the proof implied that any n-state 2NFA can be simulated by an NFA with at most $n2^{n^2}$ states. The well-known proof of the equivalence of two-way and one-way finite automata *via* crossing sequences reveals a bound of $O(2^{2n \log n})$ states [67]. Recently in [77] it was noted that a straightforward elaboration on [127] shows that the cost can be brought down to even $n(n+1)^n$. However, this bound still wastes exponentially many states, as [10] shows that $8^n + 2$ states suffice by an argument based on length-preserving homomorphisms. Recently, the problem was finally solved in [77] by establishing the tight bound of the following theorem.

Theorem 1.30 (2NFA/2DFA to NFA conversion). *Let $n \geq 1$ be an integer and A be an n-state 2NFA or an n-state 2DFA. Then $\binom{2n}{n+1}$ states* *are sufficient and necessary in the worst case for an NFA to accept $L(A)$.*

Furthermore, the following tight bounds in the exact number of states for the 2DFA and 2NFA conversion to not necessarily complete DFAs have been shown in [78].

Theorem 1.31 (2NFA/2DFA to DFA conversion). *Let $n \geq 1$ be an integer and A be an n-state 2DFA. Then $n(n^n - (n-1)^n)$ states are sufficient and necessary in the worst case for a DFA to accept $L(A)$. Moreover, if A is an n-state 2NFA, then $\sum_{i=0}^{n-1} \sum_{j=0}^{n-1} \binom{n}{i}\binom{n}{j}(2^i - 1)^j$ states are sufficient and necessary in the worst case.*

The bounds reveal that two-way head motion is a very powerful resource with respect to descriptional complexity. Interestingly, when simulating two-way devices by NFAs, it does not matter whether the two-way device is nondeterministic or not. From this point of view, two-way head motion can compensate for nondeterminism.

Nevertheless, challenging problems are still open. The question of the costs for trading two-way head motion for nondeterminism, that is, the costs for simulating (two-way) NFA by 2DFAs is unanswered for decades. It was raised by Sakoda and Sipser in [122]. They conjectured that the upper bound is exponential. The best lower bound currently known is $\Omega(n^2/\log n)$. It was proved in [4], where also an interesting connection with the open problem whether L equals NL is given. In particular, if $\mathsf{L} = \mathsf{NL}$, then for some polynomial p, all integers m, and all n-state 2NFAs A, there exists a $p(mk)$-state 2DFA accepting a subset of $L(A)$ including all words whose lengths do not exceed m. However, not only are the exact bounds of that problem unknown, but we cannot even confirm the conjecture that they are exponential.

The picture was complemented by the sophisticated studies on unary languages. The problem of Sakoda and Sipser has partially been solved for the unary case in [22] as follows.

Theorem 1.32 (Unary NFA to 2DFA conversion). *Let $n \geq 1$ be an integer and A be an n-state 2DFA accepting a unary language. Then $\Theta(n^2)$ states are sufficient and necessary in the worst case for a 2DFA to accept $L(A)$.*

The result has been shown with the surprisingly simple witness languages $L_n = \{\, a^k \mid k = n \cdot i + (n-1) \cdot j,\ \text{for } i, j \geq 1 \,\}$, for all integers $n \geq 1$.

An upper bound for the remaining case has been shown in [31].

Theorem 1.33 (Unary 2NFA to 2DFA conversion). *Let $n \geq 1$ be an integer and A be an n-state 2NFA accepting a unary language. Then $O(n^{\lceil \log_2(n+1)+3 \rceil})$ states are sufficient for a 2DFA to accept $L(A)$.*

In [22, 23] the same costs as for simulating unary NFAs by DFAs are derived for removing two-way head motion from deterministic automata.

Theorem 1.34 (Unary 2DFA to NFA/DFA conversion). *Let $n \geq 1$ be an integer and A be an n-state 2DFA accepting a unary language. Then $e^{\Theta(\sqrt{n \cdot \ln n})}$ states are sufficient and necessary in the worst case for an NFA or a DFA to accept $L(A)$.*

It turned out that, again, the same costs appear for removing two-way head motion from nondeterministic automata [108].

Theorem 1.35 (Unary 2NFA to NFA/DFA conversion). *Let $n \geq 1$ be an integer and A be an n-state 2NFA accepting a unary language. Then $e^{\Theta(\sqrt{n \cdot \ln n})}$ states are sufficient and necessary in the worst case for an NFA or a DFA to accept $L(A)$.*

The result was shown for the conversion to DFAs. This gives an upper bound also for the conversion to NFAs. The lower bound for the second case follows from the tight bounds of Theorem 1.34.

So, nondeterministic as well as two-way automata are hard to simulate for DFAs even if they accept unary languages. Since the bounds are the same, it seems that two-way motion is equally powerful as nondeterminism, but both together cannot increase the descriptional capacity. This observation is confirmed by the bounds for simulations by NFAs, where similarly as in the general case it does not matter whether the two-way device is nondeterministic or not. Nevertheless, from this point of view, two-way head motion can compensate for nondeterminism. Since unary 2DFAs can simulate NFAs increasing the number of states only polynomially, which is not possible the other way around, two-way motion turned out to be more powerful than nondeterminism.

Finally, we present the results concerning the relations (needless to say, with respect to descriptional complexity) between AFAs and 2NFAs as well as 2DFAs. These problems have been suggested to be investigated in [22]. The results are all derived in [108]. Starting with the 2NFA simulation by AFAs we conclude that any unary n-state 2NFA can be converted in a

DFA having at most $e^{\Theta(\sqrt{n \cdot \ln n})}$ states by Theorem 1.35. Next, the DFA is converted into an AFA with at most $\Theta(\sqrt{n \cdot \ln n})$ states by Theorem 1.27. For the lower bound, we argue as follows. Given a unary NFA that causes the maximal blow-up when converted to a DFA, we obtain an AFA from the DFA having at least $\Theta(\sqrt{n \cdot \ln n})$ states by Theorem 1.27. Moreover, for the 2NFA simulation by AFAs we can apply the same upper bound. Since in [22, 23] it has been shown that the witness languages for the fact that there is a unary NFA that causes the maximal blow-up when converted to a DFA are also accepted by n-state 2DFAs, the lower bound also applies for the 2DFA conversion.

Theorem 1.36 (Unary 2FA to AFA conversion). *Let $n \geq 1$ be an integer and A be an n-state 2NFA or an n-state 2DFA accepting a unary language. Then $\Theta(\sqrt{n \cdot \ln n})$ states are sufficient and necessary in the worst case for an AFA to accept $L(A)$.*

For the converse simulations a result from [11] is used. It says that any 2NFA accepting a single word language $L_n = \{a^{2^n-1}\}$ must have $\Omega(2^n)$ states, while it is accepted by some n-state AFA. On the other hand, in the order of magnitude we can derive the matching upper bound from the unary AFA to DFA conversion.

Theorem 1.37 (Unary AFA to 2FA conversion). *Let $n \geq 1$ be an integer and A be an n-state AFA accepting a unary language. Then $\Theta(2^n)$ states are sufficient and necessary in the worst case for a 2NFA or 2DFA to accept $L(A)$.*

In conclusion of this part of the section some approaches not discussed are worth mentioning. In [88] the state complexity of weak restarting automata is considered. The determinization of several finite automata for subregular language families is investigated in [14]. Several papers dealt with descriptional complexity questions of unambiguous descriptors [39, 71, 94, 120, 125, 130]. In order to attack and to solve the problem of Sakoda and Sipser for subclasses, sweeping automata are investigated in [3, 95, 110, 128]. Moreover, K-visit 2NFAs are studied in [84], and [9, 75] considered the problem for positional simulations.

1.3.1.2 *Regular Expressions*

One of the most basic theorems in formal language theory is that every regular expression can be effectively converted into an equivalent finite automaton, and *vice versa* [85]. Regular expressions are defined as follows.

Definition 1.38. Let Σ be an alphabet. Then $\emptyset$, λ, and every letter $a \in \Sigma$ are *regular expressions*. If r and s are regular expressions, then $(r + s)$, $(r \cdot s)$, and (r^*) are also regular expressions. The language $L(r)$ defined by a regular expression r is defined as follows: $L(\emptyset) = \emptyset$, $L(\lambda) = \{\lambda\}$, $L(a) = \{a\}$, $L(r+s) = L(r) \cup L(s)$, $L(r \cdot s) = L(r) \cdot L(s)$, and $L(r^*) = L(r)^*$.

For convenience, parentheses in regular expressions are sometimes omitted and the concatenation is simply written as juxtaposition. The priority of operators is specified in the usual fashion: Concatenation is performed before union, and star before both product and union.

In the literature one finds a lot of different complexity measures for regular expressions. The measure size is defined to be the total number of symbols (including $\emptyset$, λ, alphabetic symbols from Σ, all operation symbols, and parentheses) of a completely bracketed regular expression (for example, used in [1], where it is called length). Another measure related to the reverse polish notation of a regular expression is rpn, which gives the number of nodes in the syntax tree of the expressions (parentheses are not counted). This measure is equal to the length of a (parenthesis-free) expression in postfix notation [1]. The alphabetic width a-width is the total number of alphabetic symbols from Σ (counted with multiplicity) [105, 28].

In order to clarify our definitions we give a small example [29].

Example 1.39. Let $r = ((0 + ((1 \cdot 0)^*)) \cdot (1 + \lambda))$ be a regular expression. Then $\mathsf{size}(r) = 20$, $\mathsf{rpn}(r) = 10$, because the expression in postfix notation reads $010 \cdot^* + 1\lambda + \cdot$, and $\mathsf{a\text{-}width}(r) = 4$. □

Further not so well known measures are the ordinary length o-length [29], the width width [28], the length (dual to width) length [28], and the sum sum [50]. To our knowledge all these measures, except for the first one, never have been studied again since their introduction. See Table 1.1 for the inductive definition of the measures. We have also included the definition of the non s-measure star height height, which comes into play at the end of this subsection.

Next we restrict ourself to the first three mentioned measures size, rpn, and a-width. As usual, the size of a regular language L, that is $\mathsf{size}(L)$, is

Table 1.1 Inductive definitions of the measures size, rpn, a-width, o-length, width, length, sum, and height for regular expressions. $\circ$ refers to the measure to be defined.

Measure	$\emptyset$	λ	$a \in \Sigma$	$(r \cdot s)$	$(r)^*$	$(r + s)$
size	1	1	1	$\circ(r) + \circ(s) + 3$	$\circ(r) + 3$	$\circ(r) + \circ(s) + 3$
rpn	1	1	1	$\circ(r) + \circ(s) + 1$	$\circ(r) + 1$	$\circ(r) + \circ(s) + 1$
a-width	0	0	1	$\circ(r) + \circ(s)$	$\circ(r)$	$\circ(r) + \circ(s)$
o-length	1	1	1	$\circ(r) + \circ(s) + 2$	$\circ(r) + 3$	$\circ(r) + \circ(s) + 3$
width	0	0	1	$\max\{\circ(r), \circ(s)\}$	$\circ(r)$	$\circ(r) + \circ(s)$
length	0	0	1	$\circ(r) + \circ(s)$	$\circ(r)$	$\max\{\circ(r), \circ(s)\}$
sum	1	1	1	$\circ(r) + \circ(s)$	$\circ(r)$	$\begin{cases} 1, \text{ if } L(r) \subseteq \Sigma \text{ and } L(s) \subseteq \Sigma \\ \circ(r) + \circ(s), \text{ otherwise} \end{cases}$
height	0	0	0	$\max\{\circ(r) + \circ(s)\}$	$\circ(r) + 1$	$\max\{\circ(r), \circ(s)\}$

defined to be the minimum size among all regular expressions denoting L. The notions rpn(L), a-width(L), and height(L) are analogously defined. One can easily show that the measures rpn and a-width are linearly related to size, thus these are all s-measures. Relations between these measures have, for example, been studied in [28, 29, 45, 50, 56, 72].

Theorem 1.40 (Relation on basic regular expression measures). *Let L be a regular language. Then*

(1) $\mathit{size}(L) \leq 3 \cdot \mathit{rpn}(L)$ *and* $\mathit{size}(L) \leq 8 \cdot \mathit{a\text{-}width}(L) - 3$,
(2) $\mathit{a\text{-}width}(L) \leq \frac{1}{2} \cdot (\mathit{size}(L) + 1)$ *and* $\mathit{a\text{-}width}(L) \leq \frac{1}{2} \cdot (\mathit{rpn}(L) + 1)$,
(3) $\mathit{rpn}(L) \leq \frac{1}{2} \cdot (\mathit{size}(L) + 1)$ *and* $\mathit{rpn}(L) \leq 4 \cdot \mathit{a\text{-}width}(L) - 1$.

Because there are so many results on these measures, we further have to narrow our focus on some important aspects. We discuss some (recent) results on the conversion from regular expressions to finite automata and *vice versa* in more detail. Moreover, for our presentation we concentrate on the measure a-width for regular expressions.

Converting regular expressions to finite automata is well understood. The following theorem is due to [37] and [91, 92]. The upper bounds are obtained by effective constructions. For the regular expression to NFA conversion the so called *Glushkov* or *position* automaton is constructed [37], which has the property that the initial state has no incoming transitions.

Theorem 1.41 (Regular expression to FA conversion). *Let $n \geq 1$ be an integer and r be a regular expression with* a-width$(r) = n$. *Then $n + 1$ states are sufficient and necessary in the worst case for an NFA to accept $L(r)$. In case of a DFA $2^n + 1$ states are sufficient.*

The tightness of the bound for NFAs can be seen by the unary witness languages $L_n = \{a^n\}$, for integers $n \geq 1$, for which any NFA needs at least $n+1$ states. In general, techniques to prove lower bounds on the number of states for NFAs are presented in [8, 36, 47, 68]. More on the conversion from regular expressions to finite automata can be found in [121, 136]. The papers [20, 45, 56, 72] also may serve as good references for recent developments.

What concerns the other conversion direction? Probably the most popular algorithm for converting a finite automaton into an equivalent regular expression is the state elimination technique, which is a variant of the algorithm of McNaughton and Yamada [105]. All known algorithms covering the general case of infinite languages are based on the classical ones, which are compared in the survey [121]. The drawback is that all of these (structurally similar) algorithms return expressions of size $2^{O(n)}$ in the worst case [29, 48], assuming an alphabet size at most polynomial in the number of states. Recently a family of languages over a *binary* alphabet was exhibited for which this exponential blow up is inevitable [48].

Theorem 1.42 (FA to regular expression conversion). *Let $n \geq 1$ be an integer and A be an n-state finite automaton with input alphabet size at most polynomial in n. Then alphabetic width $2^{\Theta(n)}$ is sufficient and necessary in the worst case for a regular expression to describe $L(A)$.*

Note that the conversion problem for finite automata accepting unary languages becomes linear for DFAs and polynomial for NFAs [29]; in the latter case the proof is based on the Chrobak normalform for NFAs accepting unary languages. When changing the representation of a *finite* language from a DFA or NFA to an equivalent regular expression, a tight bound of order $n^{\Theta(\log n)}$ was shown in [50]. Unary finite languages accepted by finite automata can easily be converted to regular expressions of linear alphabetic width.

Before we close this section we want to comment on lower bound techniques for regular expressions. In the literature one can find at least three techniques. The first one is due to Ehrenfeucht and Zeiger, which however requires, in its original version, a largely growing alphabet. Recently, a variation of this method was used in [32] to get similar but weaker lower bounds. A technique based on communication complexity that applies only for finite languages is proposed in [50]. The most general technique, up to now, was developed in [48], where the s-measure $\mathsf{a\text{-}width}$ is related to the non s-measure star height height of regular languages—for a definition of star height we refer to Table 1.1.

Theorem 1.43 (Star height versus alphabetic width). *Let L be a regular language. Then $\mathsf{height}(L) \leq 3 \cdot \log(\mathsf{a\text{-}width}(L) + 1) + 1$.*

As mentioned above, the measure height is not an s-measure. This is easily seen because every finite language has star height 0, but can be of arbitrary size. Thus, the difference in the above theorem can be arbitrarily large. Nevertheless, language families are known, where this bound is tight up to a constant factor: define the languages L_n inductively by $L_1 = \lambda$ and $L_n = (0L_{n-1}1)^*$, for all integers $n \geq 1$. Then $\mathsf{a\text{-}width}(L_n)$ is clearly at most $2n$, but it is known from [104] that $\mathsf{height}(L_{2^n}) = n$, for all integers $n \geq 1$. Thus, the previous theorem can be used for proving lower bounds on $\mathsf{a\text{-}width}(L)$ by determining the star height of the language L.

The star height of regular languages has been intensively studied in the literature for more than 40 years, see [63, 83] for a recent treatment. Determining the star height can be in some cases reduced to the easier task of determining the cycle rank of a certain digraph, a digraph connectivity measure defined in [27] in the 1960s. Observe, that measuring the connectivity of digraphs is a very active research area [6, 7, 74].

1.3.2 *Pushdown Automata and Context-Free Grammars*

In the previous section the relative succinctness of descriptional systems representing the regular languages has been discussed. A more general treatment deals with descriptional systems having a non-empty intersection. For example, one can consider languages that are deterministic *and* linear context free in order to study the relative succinctness of deterministic pushdown automata and linear context-free grammars. A more particular approach is to represent regular languages by pushdown automata or context-free grammars.

1.3.2.1 *Pushdown Automata*

Measuring the size of a pushdown automaton (PDA) by its number of states, as is done for finite automata, is clearly ineligible. It is well known that every pushdown automaton can effectively be converted into an equivalent one having just one sole state [67]. But, in general, one has to pay with an increase in the number of stack symbols, and determinism is not preserved. For deterministic pushdown automata accepting by empty pushdown, the expressive power is known to increase strictly with the number of states [57]. A language is accepted by a one-state deterministic pushdown automaton if and only if it is a *simple* language, that is, it is generated by a context-free grammar of a very restricted form. So, measuring the size of a (deterministic) pushdown automaton by its number of stack symbols is also too crude. In fact, it is also possible to reduce the number of stack symbols if one pays with an increase in the number of states. The precise relations between states and stack symbols have been shown in [40] and [42].

Theorem 1.44. *For every (real-time) PDA with n states and t stack symbols and for every integer m in the range $1 \leq m < n$, there is an equivalent (real-time) PDA with m states and $t\lceil n/m \rceil^2 + 1$ stack symbols.*

The conversion preserves real-time behavior but not determinism. However, the construction is more or less the best possible, in the sense that an expansion in the stack alphabet to size $t\lceil n/m \rceil^2$ is sometimes unavoidable even if real-time behavior need not to be preserved, and also in the sense that no general state-reduction procedure can preserve determinism.

Theorem 1.45. *For every pair of positive integers n and t, there is a deterministic real-time PDA with n states and t stack symbols such that*

(1) every equivalent PDA with m states has at least $t\lceil n/m \rceil^2$ stack symbols, and
(2) every equivalent PDA having fewer than n states is not deterministic.

These results immediately raise the question of the converse transformations, that is, for transformations that reduce the number of stack symbols. The question has been answered in [42]. In particular, determinism can be preserved, but if it is allowed to introduce nondeterminism, states can be saved.

Theorem 1.46.

(1) For every (deterministic) PDA with n states and t stack symbols and for every integer r in the range $2 \leq r < t$, there is an equivalent (deterministic) PDA with r stack symbols and $O(n \cdot t/r)$ states.
(2) For every PDA with n states and t stack symbols and for every integer r in the range $2 \leq r < t$, there is an equivalent PDA with r stack symbols and $O(n\sqrt{t/r})$ states.

Both transformations do not preserve real-time behavior. However, again, these transformations are essentially the best possible ones. There are no transformations preserving determinism which always increase the number of states by less than $O(n \cdot t/r)$, there are no transformations which always increase the number of states by less than $O(n\sqrt{t/r})$, and there are no constructions which always preserve real-time behavior at all.

Theorem 1.47. *For every pair of positive integers n and t,*

(1) there is a deterministic PDA with n states and t stack symbols such that every equivalent deterministic PDA with r stack symbols has at least $n \cdot t/r$ states,
(2) there is a PDA with n states and t stack symbols such that every equivalent PDA with r stack symbols has at least $n\sqrt{t/r}$ states, and
(3) there is a deterministic real-time PDA with n states and t stack symbols such that every equivalent real-time PDA has at least t stack symbols.

So, besides the input alphabet, the number of states as well as the number of stack symbols have to be considered to measure the size of a pushdown automaton. But even their product is insufficient. For example, for all integers $n \geq 1$ the language $L_n = (a^n)^*$ can be accepted by a PDA with two states and two stack symbols that, in one move, is able to push n symbols on the stack. So, in addition, we have to take into account the lengths of the right-hand sides of the transition rules which can get long when a PDA pushes lots of symbols during single transitions.

Given a pushdown automaton $A = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ with state set Q, input alphabet Σ, stack alphabet Γ and transition δ mapping $Q \times (\Sigma \cup \{\lambda\}) \times \Gamma$ to finite subsets of $Q \times \Gamma^*$, we consider size to be the measure of complexity defined by $|Q| \cdot |\Sigma| \cdot |\Gamma| \cdot h$, where h is the length of the longest word pushed in a single transition. In order to see that size is an s-measure for pushdown automata we observe that there are at most $|Q| \cdot (|\Sigma|+1) \cdot |\Gamma|$ different left-hand sides and at most $|Q| \cdot |\Gamma|^h$ different right-hand sides of transition rules. So, there are at most $\mathsf{size}(A)^{h+2}$ transition rules, and we may choose $g(\mathsf{size}(A), \mathrm{alph}(A)) = k \cdot \mathsf{size}(A)^{h+2}$, where k is a mapping that gives the length of a rule dependent on the actual encoding alphabet, the number of states, input symbols, and stack symbols.

In order to avoid technical encumbrance sometimes PDAs are considered that are allowed to push at most two symbols during every transition. In [57] these PDAs are called *moderate.* It is well known that every (deterministic) PDA can effectively be converted into a moderate one. But how about the changes in size? For the conversion every transition rule of the form $\delta(p,a,t) \ni (q, u_1u_2\cdots u_m)$ with $m > 2$ is replaced by the rules $\delta(p,a,t) \ni (p_1, u_{m-1}u_m)$, $\delta(p_1,\lambda,u_{m-1}) \ni (p_2, u_{m-2}u_{m-1})$, $\delta(p_2,\lambda,u_{m-2}) \ni (p_3, u_{m-3}u_{m-2}), \ldots, \delta(p_{m-2},\lambda,u_2) \ni (q, u_1u_2)$, where $p_1, p_2, \ldots, p_{m-2}$ are new states not appearing in any other rule. Therefore, roughly $m+4$ symbols are replaced by $6(m-1)$ symbols. This implies that every PDA A can be converted into a moderate one having size $\Theta(\mathsf{size}(A))$.

Next we turn to some fundamental results in connection with the representation of regular languages by pushdown automata. In [129] the decidability of regularity for deterministic pushdown automata has been shown by a deep proof. This effective procedure revealed the following upper bound for the trade-off in descriptional complexity when deterministic pushdown automata accepting regular languages are converted into DFAs. Given a deterministic pushdown automaton with $n > 1$ states and $t > 1$ stack symbols that accepts a regular language. Then the number of states which is sufficient for an equivalent DFA is bounded by an expression of the order $t^{n^{n^n}}$. Later this triple exponential upper bound has been improved by one level of exponentiation in [132].

Theorem 1.48. *Let A be a deterministic pushdown automaton with n states, t stack symbols, and h is the length of the longest word pushed in a single transition. If $L(A)$ is regular then $2^{2^{O(n^2 \log n + \log t + \log h)}}$ states are sufficient for a DFA to accept $L(A)$.*

In the levels of exponentiation this bound is tight, since the following double exponential lower bound has been obtained in [109]. It is open whether the precise lower bound or the precise upper bound can be improved in order to obtain matching bounds.

Theorem 1.49. *Let $n \geq 1$ be an integer. Then there is a language L_n accepted by a deterministic pushdown automaton of size $O(n^3)$, and each equivalent DFA has at least 2^{2^n} states.*

The theorem is witnessed by languages L_n that are subsets of $\{0, 1, a_1, a_2, \ldots, a_n\}^*\{0,1\}^n$. The subsets are specified by an accepting deterministic pushdown automaton which operates as follows:

(1) Push the input onto the stack until symbol a_1 appears. If a_1 does not occur, reject the input.
(2) Set $i = 2$.

(3) If the next input is 0, pop the stack until the first occurrence of a_i. Else if the next input is 1, pop the stack until the second occurrence of a_i. Else if any other input appears or the occurrences of a_i are not found, reject the input.
(4) Increment i by one.
(5) If $i \leq n$, repeat step (3).
(6) If the digit on top of the stack is 1 and there are no more input symbols, accept the input. Otherwise reject the input.

The natural next step is to consider the trade-offs when nondeterministic or finite-turn pushdown automata accepting regular languages are converted into DFAs. But in [109] it has been shown that for *any* given recursive function f and arbitrarily large integers n, there exists a nondeterministic pushdown automaton of size n representing a regular language, such that any equivalent DFA has at least $f(n)$ states. This implies that there does not exist a recursive function serving as upper bound for the trade-off. We deal with this phenomenon in Section 1.5 in more detail. However, the situation is different when unary languages are considered. It is well known that every unary context-free language is regular [35]. From the viewpoint of descriptional complexity, unary nondeterministic pushdown automata have been investigated in [116] where PDAs having a *strong normalform* are considered. In particular, the normalform PDAs are such that (1) at the start of the computation the stack contains only the bottom-of-stack symbol which is never pushed or popped, (2) the input is accepted if and only if the automaton reaches an accepting state, the stack contains only the bottom-of stack symbol, and all the input has been scanned, (3) if the PDA moves the input head, then no operations are performed on the stack, and (4) every push adds *exactly* one symbol on the stack. The complexity measure used in [116] is the product of the number of states and the number of stack symbols. While this measure is reasonable for the normalform PDAs, it distorts the results with respect to PDAs that are measured, for example, by size. However, similarly as for moderate PDAs it is not hard to convert a given PDA into a normalform PDA, whereby the number of stack symbols is increased by at most one, and the number of states is increased linearly in size of the given automaton. This implies that every PDA A can be converted into a normalform one having size $\Theta(\mathsf{size}(A))$.

Theorem 1.50. *For every normalform PDA with n states and t stack symbols accepting a unary language there is an equivalent NFA with $2^{2n^2t+1}+1$ states, and an equivalent DFA with $2^{n^4t^2+2n^2t+1}$ states. Therefore, for every* *arbitrary PDA of size s accepting a unary language there is an equivalent NFA with $2^{O(s^2)}$ states, and an equivalent DFA with $2^{O(s^4)}$ states.*

Similar investigations for deterministic pushdown automata have been done in [115]. It is proved that each unary deterministic pushdown automaton of size s can be converted into a DFA with a number of states exponential in s. Moreover, this bound is tight in the order of magnitude.

Theorem 1.51. *For every normalform deterministic pushdown automaton with n states and t stack symbols accepting a unary language there is an equivalent DFA with $2^{n \cdot t}$ states. Therefore, for every arbitrary deterministic PDA of size s accepting a unary language there is an equivalent DFA with $2^{O(s)}$ states.*

Theorem 1.35 says that any unary n-state 2NFA can be simulated by a DFA with $e^{\Theta(\sqrt{n \cdot \ln n})}$ states. This suggests the possibility of a smaller gap between the descriptional complexities of unary deterministic pushdown automata and 2NFAs. However, even in this case the gap can be exponential.

Theorem 1.52. *Let $s \geq 1$ be an integer. Then there is a unary language L_s accepted by a deterministic pushdown automaton of size $8s + 4$, and each equivalent DFA has at least 2^s states. Moreover, each equivalent 2NFA has also at least 2^s states.*

The theorem is witnessed by the languages $L_s = \{a^{2^s}\}^*$ containing multiples of 2^s in unary notation. By a result in [107] it can be shown that every 2NFA accepting L_s must have at least 2^s states. The studies in [115] are complemented by answering the question whether or not for each unary regular language there exists an exponential gap between the sizes of deterministic pushdown automata and finite automata negatively.

Theorem 1.53. *Let $n \geq 1$ be an integer. Then there is a language L_n accepted by a DFA as well as by a 2NFA with 2^n states, and each equivalent deterministic pushdown automaton is at least of size $O(2^n/n^2)$.*

Finite-turn pushdown automata accepting (letter-)bounded languages are studied in [102], where a language is said to be bounded if it is a subset of $a_1^* a_2^* \cdots a_m^*$, for some alphabet $\Sigma = \{a_1, a_2, \ldots, a_m\}$. It is known that arbitrary finite-turn pushdown automata accept exactly the ultralinear languages. While it will turn out in Theorem 1.92 that the increase in size when converting arbitrary PDAs accepting ultralinear languages to finite-turn PDA cannot be bounded by any recursive function, for bounded languages an exponential trade-off is obtained.

Theorem 1.54. *Let $\Sigma = \{a_1, a_2, \ldots, a_m\}$ be an alphabet. For every PDA of size s accepting a subset of $a_1^* a_2^* \cdots a_m^*$ there is an equivalent $(m-1)$-turn pushdown automaton of size $2^{O(s^2)}$.*

In addition, in [102] a conversion algorithm is presented and the optimality of the construction is shown by proving tight lower bounds. Furthermore, the question of reducing the number of turns of a given finite-turn PDA is studied. Again, a conversion algorithm is provided which shows that in this case the trade-off is at most polynomial.

Theorem 1.55. *Let $n \geq 1$ be an integer. Then there is a bounded language $L_n \subseteq a_1^* a_2^* \cdots a_m^*$ which is accepted by an $(m-1)$-turn pushdown automaton of size $2^{O(n)}$, but cannot be accepted by any PDA making strictly less than $m-1$ turns, and for each integer $k \geq m-1$, every k-turn pushdown automaton accepting L_n is at least of size $2^{O(n)}$, for sufficiently large n.*

Theorem 1.56. *For every normalform k-turn pushdown automaton of size s accepting a bounded language $L \subseteq a_1^* a_2^* \cdots a_m^*$ there is an equivalent normalform $(m-1)$-turn pushdown automaton of size $O(m^6 s^{4\lfloor \log_2 k \rfloor + 8})$. Therefore, for every arbitrary k-turn pushdown automaton of size s accepting a bounded language $L \subseteq a_1^* a_2^* \cdots a_m^*$ there is an equivalent $(m-1)$-turn pushdown automaton of size $O(s^{6 \log k})$.*

1.3.2.2 *Context-Free Grammars*

Next we turn to compare the relative succinctness of pushdown automata and context-free grammars (CFG). Both descriptional systems are known to capture the context-free languages and, therefore, it is natural to ask whether a given context-free language can be more concisely described by an automaton or a grammar.

Definition 1.57. A *context-free grammar* (CFG) is a quadruple $G = (N, T, P, S)$, where N is a finite set of nonterminals, T is a finite set of terminal symbols, $S \in N$ is the axiom, and P is a finite set of productions of the form $A \to \alpha$, where $A \in N$ and $\alpha \in (N \cup T)^*$.

A context-free grammar G is in *Chomsky normalform* (CNF grammar) if every production in P is of the form $A \to BC$ or $A \to a$, for $A, B, C \in N$ and $a \in T$.

The *language generated* by a CFG $G = (N, T, P, S)$ is defined as

$$L(G) = \{\, w \in T^* \mid S \Rightarrow^* w \,\},$$

where $\Rightarrow^*$ denotes the reflexive, transitive closure of the derivation relation $\Rightarrow$.

Several measures for CFGs have been proposed in the literature. A detailed treatment can be found in Section 1.4. Here we are particularly interested in s-measures that allow a comparison with PDAs. To this end, for a given context-free grammar G let $\mathsf{size}(G)$ be the product of the number of productions and the length of the longest right-hand side of a production in P. Clearly, size is an s-measure for context-free grammars. On the other hand, the number of nonterminals, that is, $|N|$ is *not* an s-measure for CFGs in general. Assume contrarily it is. Then there is a recursive function g such that $\mathsf{length}(G) \leq g(|N|, T)$, for every CFG G. But for the grammar $G = (\{S\}, \{a\}, \{S \to a^{g(1,\{a\})+1}\}, S)$ the value $\mathsf{length}(G)$ exceeds $g(|N|, T)$. However, the number of nonterminals *is* an s-measure for context-free grammars in Chomsky normalform. In this case there are at most $|N|$ different left-hand sides of productions, and at most $|N|^2 + |T|$ right-hand sides. So, there are at most $|N|^3 + |N| \cdot |T|$ productions containing at most three nonterminal and terminal symbols, and we may choose $g(|N|, T) = k \cdot (|N|^3 + |N| \cdot |T|)$, where k is a mapping that gives the precise length of a production depending on the actual encoding alphabet, the number of nonterminals and terminal symbols. Note, that in this case the underlying descriptional system is a strict subfamily of the context-free grammars, the grammars in Chomsky normalform.

First, we present some results concerning the conversion of finite automata into CNF grammars. By standard construction every n-state DFA or NFA can be converted into a regular right-linear grammar with n nonterminals. This, in turn, can be converted into a CNF grammar with $n + i$ nonterminals, where i is the number of input symbols. The following lower bound has been obtained in [25].

Theorem 1.58. *Let $n \geq 1$ be an integer. Then there is a language L_n accepted by a DFA with $2^n + n + 1$ states and by an NFA with $2^n + n$ states, and every CNF grammar generating L_n has at least $\Omega(2^n/n)$ nonterminals.*

Upper and lower bound can significantly be improved in the unary case. Moreover, for DFAs they are tight in the order of magnitude [25].

Theorem 1.59. *Let $n \geq 1$ be an integer and A be an n-state DFA accepting a unary language. Then $\Theta(n^{1/3})$ nonterminals are sufficient and necessary* *in the worst case for a CNF grammar to generate $L(A)$. In case of an NFA $\Theta(n^{2/3})$ nonterminals are sufficient for a CNF grammar to generate $L(A)$.*

Next, the conversion of pushdown automata into context-free grammars is considered. The so-called *triple construction* is the standard method for converting a PDA which accepts by empty stack into a CFG [67]. Given a PDA with n states and t stack symbols it constructs a CFG with n^2t+1 nonterminals if $n > 1$, and n^2t nonterminals otherwise. The result has been complemented in [116], where it is shown by a modified triple construction that the resulting context-free grammar can be converted in a CNF grammar without introducing new nonterminals when the PDA is in normalform.

Theorem 1.60. *For every normalform PDA with n states and t stack symbols there is an equivalent CNF grammar with n^2t+1 nonterminals. Therefore, for every arbitrary PDA of size s there is an equivalent CNF grammar with $O(s^2)$ nonterminals.*

In the worst case, this number of nonterminals is necessary even if the PDA is real time, deterministic, and accepts by empty stack [41].

Theorem 1.61. *Let $n, t \geq 1$ be integers. Then there is a language $L_{n,t}$ accepted by a deterministic real-time pushdown automaton by empty stack that has n states, t stack symbols, and size $O(nt)$, and each equivalent CFG has at least n^2t+1 nonterminals if $n > 1$, and n^2t nonterminals otherwise.*

It follows that there are context-free languages which can be recognized by pushdown automata of size $O(nt)$, but which cannot be generated by context-free grammars of size smaller than $O(n^2t)$. Moreover, the standard construction for converting a pushdown automaton to an equivalent context-free grammar is optimal with respect to the number of nonterminals.

The situation is better for unary languages [115].

Theorem 1.62. *For every unary normalform deterministic pushdown automaton of size s there is an equivalent CNF grammar at most of size $O(s)$.*

Theorems 1.58, 1.59, and 1.59 dealt with the simulation of finite automata by CNF grammars. Since for unary context-free languages there are always equivalent DFAs and NFAs, the converse simulations are also worth studying. The following results are proven in [116].

Theorem 1.63. *For every unary CNF grammar with n nonterminals there is an equivalent NFA with $2^{2n-1}+1$ states.*

This upper bound is close to optimal.

Theorem 1.64. *Let $n \geq 1$ be an integer. Then there is a unary CNF with n nonterminals such that every equivalent NFA has at least $2^{n-1}+1$ states.*

By Theorem 1.63, given a unary CNF with n nonterminals there is an equivalent NFA with $2^{O(n)}$ states. This automaton can be transformed into a DFA applying the powerset construction or the determinization procedure for unary automata presented in [22]. In both cases, the number of states of the resulting DFA is bounded by a function which grows at least double exponential in n. In [116] it is proved that this cost can drastically be reduced.

Theorem 1.65. *For every unary CNF grammar with n nonterminals there is an equivalent DFA with 2^{n^2} states.*

Note, in the particular case $n = 1$ the upper bound given in Theorem 1.65 does not hold. It is not difficult to show that the only non-empty languages generated by unary CNF grammars with one variable are $\{a\}$ and $\{\, a^k \mid k \geq 1 \,\}$. The minimal DFAs accepting these languages have three and two states, respectively. However, the upper bound stated in Theorem 1.65 is tight.

Theorem 1.66. *For infinitely many integers $n \geq 1$, there is a unary CNF with n nonterminals such that every equivalent DFA has at least $2^{O(n^2)}$ states.*

The relations between the subfamily of finite-turn pushdown automata accepting (letter-)bounded languages and CNF grammars are studied in [102].

Theorem 1.67. *For every CNF grammar with n nonterminals generating a bounded language $L \subseteq a_1^* a_2^* \cdots a_m^*$ there is an equivalent normalform $(m-1)$-turn PDA with $2^{O(n)}$ states and $O(1)$ stack symbols.*

For the situation where the given grammar is not necessarily in Chomsky normalform the following result has been shown.

Theorem 1.68. *For every context-free grammar of size s generating a bounded language $L \subseteq a_1^* a_2^* \cdots a_m^*$ there is an equivalent normalform $(m-1)$-turn PDA with $2^{O(s)}$ states and $O(1)$ stack symbols.*

The lower bound is given by the next theorem.

Theorem 1.69. *Let $n \geq 1$ be an integer. Then there is a unary language L_n generated by a CNF grammar with $n+1$ nonterminals such that for each integer $k \geq 1$, every normalform k-turn pushdown automaton accepting L_n is at least of size $2^{O(n)}$, for sufficiently large n.*

So far, essentially (deterministic) pushdown automata and context-free grammars (in Chomsky normalform) have been discussed. A sub-class of CFGs that characterize the deterministic context-free languages are LR(k) grammars, where $k \geq 1$, can be seen as the length of the lookahead of a corresponding LR(k) parser. Already the class of LR(1) grammars characterizes the deterministic context-free languages. The use of a longer lookahead k does not increase their generative capacity. But from a descriptional complexity point of view the question whether a longer lookahead can reduce the size of a grammar has been answered in [96] affirmatively. The practical relevance of these results is immediate. A sequence of languages L_n is presented such that there is a progressive trade-off in the size of the LR(k) grammars as the length of the lookahead varies.

Theorem 1.70. *Let $n \geq 2$ and $k \geq 0$ be integers satisfying $k \leq n - 9\log n$. Then there is a language L_n such that every LR(k) grammar generating L_n is at least of size $2^{\Theta(n-k)}$.*

1.4 Measuring Resources

The investigation of several aspects of measuring descriptors with s-measures is responding to an interest to optimize the space requirements. Basically, some resources which are recursively related to the length of the description are measured, and the relative succinctness of different types of descriptors is studied. But what makes the difference between two types of descriptors? Roughly speaking, it is their different equipment with resources. For example, the difference between a DFA and an NFA is made by the resource *nondeterminism*, or the difference between an NFA and a PDA is caused by the resource *stack*. So, in more general terms, we fix some, say, structural resources and study their descriptional power by measuring other resources. This immediately raises the question which resources are structural and which can be measured. Here, the only restriction we impose is to have descriptional systems with recursive complexity measures. Given a descriptor its complexity must be computable. So, even though they are naturally motivated the s-measures are special cases, only. This section is devoted to descriptional systems measured by measures which are *not* recursively related to their length.

A context-free grammar $G = (N, T, P, S)$ is *right-linear* or *regular* if every production in P is of the form $A \to uB$ or $A \to u$, for $A, B \in N$ and $u \in T^*$. The states of a finite automaton correspond roughly to the number of nonterminals of a regular grammar and *vice versa*. This leads to the idea of considering the number of nonterminals as a measure for arbitrary context-free grammars. In the forthcoming we stick with context-free grammars and subclasses as descriptional systems, while the below given definitions easily generalize to arbitrary phrase structure grammars. For a context-free grammar $G = (N, T, P, S)$, we define the following three measures [52]:

$$\mathsf{var}(G) = |N|,$$
$$\mathsf{prod}(G) = |P|,$$

and

$$\mathsf{symb}(G) = \sum_{(A \to \alpha) \in P} (|A| + |\alpha| + 1).$$

In order to clarify the definitions we present an example [52].

Example 1.71. Let $n \geq 1$ be an integer. Consider the context-free grammar $G = (\{S, A\}, \{a\}, P, S)$ with the three productions $S \to A^n$, $A \to a$, and $A \to aa$. Then $L(G) = \{\, a^i \mid n \leq i \leq 2n \,\}$ and $\mathsf{var}(G) \leq 2$, $\mathsf{prod}(G) \leq 3$, and $\mathsf{symb}(G) \leq n + 9$. In fact, there exist an equivalent context-free grammar G' with $\mathsf{var}(G') = 1$ because language $L(G)$ is finite and can be easily generated by a CFG with one nonterminal only. $\square$

As already mentioned previously, the measure var is not an s-measure. The same holds true for the measure prod, while symb is obviously recursively related to the size of the CFG. Observe, that it heavily depends on the underlying descriptional system, if a measure becomes an s-measure. For instance, the number of variables is *not* even an s-measure for regular grammars in general, but it becomes an s-measure if the descriptional system is chosen to be that of *regular grammars in normalform*, that is, every production is of the form $A \to aB$ or $A \to a$. Further measures based on other criteria induced by grammatical levels, derivation trees, derivation steps, etc., are introduced and studied in [51, 52, 54, 55].

Before we summarize some results on the measures var and prod we find it worth mentioning, that basic algorithmic problems for most of the measures on context-free grammars and languages are undecidable. For instance, to determine the complexity of a given language, to construct a minimal equivalent grammar, to decide minimality of a given grammar, and so on. For further readings on this topic we refer to [52, 51, 54, 19]. Now we turn to the first result on the number of variables for context-free grammars [51].

Theorem 1.72 (var for CFL). *Let $n \geq 1$ be an integer. Then there exists a regular language L_n over a binary alphabet, such that $\mathsf{var}(G) \geq n$ for every context-free grammar generating L_n.*

Thus, the var-measure for context-free grammars induces a dense and strict hierarchy of increasing levels of difficulties. This property of inducing a dense hierarchy is known in the literature as the *connectedness property* with respect to an alphabet [52]. If we consider the classification of context-free grammars in terms of the number of productions, we find a similar statement as above [52]. For *every* alphabet T and all integers $n \geq 1$ there is a finite language L_n over T, such that $\mathsf{var}(G) \geq n$ for every context-free grammar generating L_n. Thus, the prod-measure is also connected, even for unary alphabets. The witness language to show the result on the number of productions is the finite language $\{\, a^{2i} \mid 0 \leq i \leq n-1 \,\}$. On the other hand, if the alphabet is unary, the situation for var changes drastically [51].

Theorem 1.73 (var for unary CFL). *Let L be a unary context-free grammar. Then two nonterminals are sufficient and necessary in the worst case for an equivalent context-free grammar.*

Concerning the smallest level of the number of nonterminals, in [53] it is mentioned that the language generated by $G = (\{S\}, T, P, S)$, where $P = \{S \to \alpha \mid \alpha \in F\,\}$ for some finite $F \subseteq (\{S\} \cup T)^*$, is equal to the iterated S-substitution of F, that is, $L(G) = F^{\uparrow^S}$. Here for a letter a and two languages L_1 and L_2, the *a-substitution* of L_2 in L_1, denoted by $L_1 \uparrow^a L_2$, is defined by

$$\begin{aligned} L_1 \uparrow^a L_2 = \{\, & u_1 v_1 u_2 \ldots u_k v_k u_{k+1} \mid u_1 a u_2 a \ldots a u_{k+1} \in L_1, \\ & a \text{ does not occur in } u_1 u_2 \ldots u_{k+1}, \text{ and } v_1, v_2 \ldots, v_k \in L_2 \,\}, \end{aligned}$$

and the *iterated a-substitution* of language L, denoted by $L^{\uparrow^a}$, is defined by

$$L^{\uparrow^a} = \{\, w \in L \cup (L \uparrow^a L) \cup (L \uparrow^a L \uparrow^a L) \cup \cdots \mid \text{word } w \text{ has no occurrence of letter } a \,\},$$

where any further bracketing is omitted since a-substitution is obviously associative. The definition of iterated substitution expressions gives a nice and convenient way to specify context-free languages in terms of expressions. For the characterization of context-free languages by means of expressions we refer also to [106] and [135]. Although these approaches are very similar there are subtle differences; see the former reference for the relation between McWhirter's expressions and Gruska's substitution model. The relation between auxiliary symbols in substitution expressions and the number of nonterminals in context-free grammars is discussed in [53] in more detail (cf. [44]).

Next, let us restrict our attention to regular grammars. Here the the smallest level with respect to the number of nonterminals is more handy to describe. One can easily show that if a regular language L is generated by regular grammar with one nonterminal, then there exist two regular sets R_1 and R_2 such that $L = R_1^* R_2$. This result can further be generalized to higher levels.

We continue with some results on the comparison between context-free and regular grammars. The natural question arises, whether the former have advantages compared with the latter according to the measures considered so far? This question was answered in [52], where the following result was shown.

Theorem 1.74 (var and prod for CFL versus REG). *There is a regular language L such that any regular grammar generating L has strictly more nonterminals than a minimal equivalent context-free grammar. The statement remains valid for the number of productions.*

Observe, that the strict increase in complexity already happens for non-self-embedding context-free grammars. Here a context-free grammar is *self-embedding* if there is a nonterminal X such that there is a derivation $X \Rightarrow^* \alpha X \beta$, for both non-empty α and β. It is well known that non-self-embedding context-free grammars generate regular languages only. A closer look on the previous theorem reveals even more. In fact, an easy example given in [52] shows that the difference between the number of nonterminals in equivalent context-free and regular grammars can be arbitrarily large.

For all integers $n \geq 1$, consider the language $L_n = (a^*bab^*)^n$. Two nonterminals are sufficient for a context-free grammar while each regular grammar needs at least $2n$ nonterminals in order to generate L_n. Analogously one can show that four productions are sufficient for a context-free grammar while each regular grammar needs at least $2n$ productions. Further examples showing that the measures can yield essential different classifications, when the underlying grammar is varied, can be found in [52]. In this way so called *bounded* complexity measure results are obtained.

Another field of research that is related to bounded complexity measures is the study of the descriptional complexity of various types of grammars that can be used to describe context-free languages. For example, context-free grammars and their normalform restrictions such as λ-free normal form, Chomsky normalform, Greibach normalform, position restricted grammars, etc. Transformations of context-free grammars into normalforms may change their complexity with respect to the measures under consideration. In a series of papers these questions were addressed [19, 44, 52, 80, 81, 117, 118]. Some of the most interesting results are presented next. We start with some results based on restricted context-free grammars. A context-free grammar is *restricted* if it is λ-free and does not contain any unit productions.

Theorem 1.75 (Bounded complexity for restricted CFGs). *Let G be a context-free grammar. Then there is an equivalent restricted context-free grammar G' such that*

(1) $\mathsf{var}(G') \leq \mathsf{var}(G)$,
(2) $\mathsf{prod}(G') \leq \frac{1}{2} \cdot \mathsf{prod}^2(G)$,
(3) $\mathsf{symb}(G') \leq \frac{1}{2} \cdot \mathsf{symb}^2(G)$.

The latter two bounds cannot be improved by more than a constant.

If each two equivalent minimal descriptors from different descriptional systems have the same complexity with respect to a measure, this measure is called *dense*. So, the measure var is dense for context-free and restricted context-free grammars. In the next theorem we will see that this is not true in general, because the var is *not* dense for (restricted) context-free and context-free grammars in Greibach normalform.

A context-free grammar $G = (N, T, P, S)$ is in *Greibach normalform* [43] if every production in P is of the form $A \to a\alpha$, for $A \in N$, $a \in T$, and $\alpha \in N^*$.

Theorem 1.76 (Bounded complexity for CFGs in normalforms). *Let G be a restricted context-free grammar. Then*

(1) there is an equivalent context-free grammar G' in Chomsky normalform such that $\mathsf{symb}(G') \leq 7 \cdot \mathsf{symb}(G)$,
(2) there is an equivalent context-free grammar G' in Greibach normalform such that $\mathsf{var}(G') \leq 2 \cdot \mathsf{var}(G)$ and this bound is the best possible, $\mathsf{prod}(G') \in O(\mathsf{prod}^3(G)) \cap \Omega(\mathsf{prod}^2(G))$, and $\mathsf{symb}(G') \in O(\mathsf{symb}^3(G)) \cap \Omega(\mathsf{symb}^2(G))$.

In the remainder, we turn our attention to the relation between context-free grammars and other grammars from the Chomsky hierarchy such as monotone grammars, that is, grammars with productions whose right-hand side is not shorter than the left-hand side, or arbitrary phrase structure grammars with respect to the measures under consideration. The next theorem shows that the gap between the number of nonterminals for a context-free grammar and a monotone grammar representation can be arbitrarily large. Consider the languages $L_n = \bigcup_{i=1}^{n-1} b(a^i b)^+$, for all integers $n \geq 3$. By standard arguments one can show that every context-free grammar generating L_n needs at least n nonterminals. On the other hand, the monotone grammar $G_n = (\{S, A\}, \{a, b\}, P_n, S)$ with

$$P_n = \{S \to ba^i b, S \to Aa^i b, Aa^i b \to Aa^i ba^i b, A \to b \mid 1 \leq i \leq n-1\}$$

generates the L_n with two nonterminals only. This result can be slightly strengthened as follows:

Theorem 1.77. *Let $n \geq 1$ be an integer. Then there exists a regular language L_n such that every context-free grammar generating L_n has at least n nonterminals, and there is an equivalent monotone grammar with* 2 *nonterminals.*

A similar situation appears if we consider the measure prod. Note that the witness language for the next theorem is a finite language. Further readings on the measure prod for finite languages can be found in [16, 18, 17].

Theorem 1.78. *Let $n \geq 1$ be an integer. Then there exists a* finite *language L_n such that every context-free grammar generating L_n has at least n productions, and there is an equivalent monotone grammar with* 5 *nonterminals.*

The argument for this statement is not too complicated [114]. We have already mentioned that every context-free grammar G generating the (finite) language $L_n = \{\, a^{2i} \mid 0 \leq i \leq n-1 \,\}$ has at least n productions. Now we consider the modified language

$$L'_n = \{\, b^{n-i-1}a^{2i}b^{i+1} \mid 0 \leq i \leq n-1 \,\}.$$

We have $L_n = h(L'_n)$ for the erasing homomorphism $h : \{a,b\}^* \to \{a\}^*$ defined by $h(a) = a$ and $h(b) = \lambda$. Moreover, every context-free grammar generating $h(L(G))$ clearly needs at most as many productions as G has. So, since $n \leq \mathsf{prod}(G)$ every context-free grammar G' generating L'_n has at least n productions. In order to conclude that n productions are enough we construct the grammar $G = (\{S\}, \{a,b\}, P, S)$ with the set of productions

$$P = \{\, S \to b^{n-i-1}a^{2i}b^{i+1} \mid 0 \leq i \leq n-1 \,\}.$$

Grammar G' generates L'_n and has n productions. Finally, it is easy to see that the monotone grammar $G' = (\{S,T,A\}, \{a,b\}, P', S)$ with productions

$$P' = \{S \to T^{n-1}ab, T \to A, T \to b, Aa \to aaA, Ab \to bb\}$$

also generates L'_n. Since P contains only five elements the statement follows.

1.5 Non-Recursive Trade-Offs

In order to motivate the main topic of this section we first deduce a property of any descriptional system $\mathcal{S}$ when it is measured by an s-measure c. Let $D \in \mathcal{S}$ be a descriptor. Since c is an s-measure there is a recursive function g such that $\mathsf{length}(D) \leq g(c(D), \mathrm{alph}(D))$. But this implies that with respect to $\mathrm{alph}(D)$ there are only finitely many descriptors in $\mathcal{S}$ having the same size as D. Otherwise, applying g to infinitely many descriptors would yield to the same result. But for any coding alphabet there are only finitely many descriptors whose length does not exceed $g(c(D), \mathrm{alph}(D))$. So, we know that for any size, $\mathcal{S}$ contains only finitely many descriptors over the same alphabet.

Assume now there are two descriptional systems $\mathcal{S}_1$ and $\mathcal{S}_2$, and two s-measures c_1 for $\mathcal{S}_1$ and c_2 for $\mathcal{S}_2$. Given a descriptor from $\mathcal{S}_1$ a natural question is for the maximal blow-up in complexity when this descriptor is converted into an equivalent one from $\mathcal{S}_2$. Clearly, if a general upper bound for the trade-off is known, the blow-up is given by that function.

Our question is somehow simpler. We are not interested in a general upper bound but in an upper bound that may use the given alphabet as an additional parameter, say $h(n, \Sigma)$. We can precisely determine h as follows. For all n and alphabets Σ, let $\mathcal{D}_{n,\Sigma}$ denote the finite subset of all descriptors from $\mathcal{S}_1$ over alphabet Σ whose complexity is n. For each $D_1 \in \mathcal{D}_{n,\Sigma}$ set $m(D_1) = \min\{\, c_2(D_2) \mid D_2 \in \mathcal{S}_2(L(D_1)) \,\}$. Then $h(n, \Sigma)$ is set to $\max\{\, m(D_1) \mid D_1 \in \mathcal{D}_{n,\Sigma} \,\}$. So, we have a function at hand that answers our question. Unfortunately, it may happen that this function is not effectively computable. What does this mean? This means that the size blow-up caused by such a conversion cannot be bounded above by *any* recursive function. With other words, one can choose an arbitrarily large recursive function but the gain in economy of description eventually exceeds it. This qualitatively new phenomenon, nowadays known as *non-recursive trade-off*, was first observed by Meyer and Fischer [109] between context-free grammars and finite automata.

In the sequel we often use the following second property of measures.

Definition 1.79. Let $\mathcal{S}$ be a descriptional system with s-measure c. If for any alphabet Σ, the set of descriptors in $\mathcal{S}$ describing languages over Σ is recursively enumerable in order of increasing size, then c is said to be an *sn-measure.*

In fact, the non-recursive trade-offs are independent of particular sn-measures. Any two sn-measures c_1 and c_2 for some descriptional system $\mathcal{S}$ are related by a function

$$h(n, \Sigma) = \max\{\, c_2(D) \mid D \in \mathcal{S} \text{ with } c_1(D) = n \text{ and } \mathrm{alph}(D) = \Sigma \,\}.$$

By the properties of sn-measures, h is recursive. So, a non-recursive trade-off exceeds any difference caused by applying two sn-measures.

1.5.1 *Proving Non-Recursive Trade-Offs*

Before we present examples of non-recursive trade-offs, we turn to the question of how to prove them. Roughly speaking, most of the proofs appearing in the literature are basically relying on one of two different schemes. One fundamental technique is due to Hartmanis [59]. In [60] a generalization is developed that relates semi-decidability to trade-offs. Next we present a slightly generalized and unified form of this technique [87].

Theorem 1.80. *Let $\mathcal{S}_1$ and $\mathcal{S}_2$ be two descriptional systems for recursive languages such that any descriptor D in $\mathcal{S}_1$ and $\mathcal{S}_2$ can effectively be con-*

verted into a Turing machine that decides $L(D)$, and let c_1 be a measure for $\mathcal{S}_1$ and c_2 be an sn-measure for $\mathcal{S}_2$. If there exists a descriptional system $\mathcal{S}_3$ and a property P that is not semi-decidable for descriptors from $\mathcal{S}_3$, such that, given an arbitrary $D_3 \in \mathcal{S}_3$, (i) there exists an effective procedure to construct a descriptor D_1 in $\mathcal{S}_1$, and (ii) D_1 has an equivalent descriptor in $\mathcal{S}_2$ if and only if D_3 does not have property P, then the trade-off between S_1 and S_2 is non-recursive.

Let us give evidence that the theorem is true. Assume contrarily that the trade-off is bounded by some recursive function f. Let D_1 be a descriptor in $\mathcal{S}_1$. If D_1 has an equivalent descriptor D_2 in $\mathcal{S}_2$, then $c_2(D_2) \leq f(c_1(D_1))$. Since f and c_1 are recursive, the value $f(c_1(D_1))$ can be computed. Next, we can recursively enumerate the finite number of descriptors in $\mathcal{S}_2$, whose underlying alphabet is $\mathrm{alph}(D_1)$ and whose size is at most $f(c_1(D_1))$. All these descriptors can effectively be converted into Turing machines that decide for any input whether it belongs to the described languages. The same holds for the descriptor D_1. By comparing the enumerated descriptors with D_1 on successive inputs over the alphabet $\mathrm{alph}(D_1)$, we can detect whether *none* of the descriptors is equivalent to D_1. As a result, a Turing machine is constructed that halts if and only if D_1 has no equivalent descriptor in $\mathcal{S}_2$. So, the set $R = \{\, D_1 \in \mathcal{S}_1 \mid D_1 \text{ has no equivalent descriptor in } \mathcal{S}_2 \,\}$ is recursively enumerable. Now the theorem follows due to the following contradiction. Given a descriptor $D_3 \in \mathcal{S}_3$ we construct the descriptor D_1 in $\mathcal{S}_1$, and semi-decide whether it belongs to the set R. If the answer is in the affirmative, there is no equivalent descriptor in $\mathcal{S}_2$ and, thus, D_3 has property P.

Example 1.81. Let $\mathcal{S}_1$ be the family of linear context-free grammars, and $\mathcal{S}_2$ be the set of deterministic finite automata. Clearly, both descriptional systems meet the preconditions of Theorem 1.80. Since the regularity of linear context-free grammars is not semi-decidable [13], we set $\mathcal{S}_3$ to be $\mathcal{S}_1$ and property P is to be a descriptor describing a non-regular language. So, any linear context-free grammar D_1 has an equivalent DFA if and only if language $L(D_1)$ is regular, that is, if it does not have property P. We conclude that the trade-off between linear context-free grammars and DFAs is non-recursive. □

Since we may apply Theorem 1.80 for any pairs of descriptional systems whose first component represents the linear context-free and whose second component represents the regular languages the following theorem is derived from Example 1.81.

Theorem 1.82. *The trade-offs between linear context-free grammars and deterministic finite automata, between one-turn pushdown automata and nondeterministic finite automata, etc., are non-recursive.*

On the one hand, the method presented can serve as a powerful tool. Several known proofs are simplified. Some more or less new non-recursive trade-offs follow immediately by known undecidability results. On the other hand, to apply Theorem 1.80 the crucial hard part is to find suitable descriptional systems $\mathcal{S}_3$ having the required properties. The example presented before considered linear context-free languages for which regularity is not semi-decidable. Another valuable descriptional system is the set of Turing machines for which only trivial problems are decidable and a lot of problems are not semi-decidable. When Theorem 1.80 is applied, one has to be a little bit careful about the negation of property P. For example, finiteness is not semi-decidable for Turing machines. Not finite means infinite, which is also not semi-decidable for Turing machines. On the other hand, emptiness is not semi-decidable, but its negation is, that is, whether the Turing machine accepts at least one input.

In order to utilize non-semi-decidable properties of Turing machines in [58] complex Turing machine computations have been encoded in small grammars. These encodings and variants thereof are of tangible advantage for our purposes. Basically, we consider *valid computations of Turing machines*. Roughly speaking, these are histories of accepting Turing machine computations. It suffices to consider deterministic Turing machines with one single tape and one single read-write head. Without loss of generality and for technical reasons, we assume that the Turing machines can halt only after an odd number of moves, accept by halting, make at least three moves, and cannot print blanks. A valid computation is a string built from a sequence of configurations passed through during an accepting computation.

Let Q be the state set of some Turing machine M, where q_0 is the initial state, $T \cap Q = \emptyset$ is the tape alphabet containing the blank symbol, and $\Sigma \subset T$ is the input alphabet. Then a configuration of M can be written as a word of the form T^*QT^* such that $t_1t_2 \cdots t_iqt_{i+1} \cdots t_n$ is used to express that M is in state q, scanning tape symbol t_{i+1}, and t_1, t_2 to t_n is the support of the tape inscription. For the purpose of the following, valid computations are now defined in three different forms:

(1) $\text{VALC}_A(M)$ is the set of strings of the form
$$\$w_1\$w_2^R\$w_3\$w_4^R\$\cdots\$w_{2n-1}\$w_{2n}^R\$.$$

(2) $\mathrm{VALC}_C(M)$ is the set of strings of the form

$$\$w_1\$w_2\$\cdots\$w_{2n}\$.$$

(3) $\mathrm{VALC}_R(M)$ is the set of strings of the form

$$\$w_1\$w_3\$\cdots\$w_{2n-1}\#w_{2n}^R\$\cdots\$w_4^R\$w_2^R\$.$$

In all three cases, $\$, \# \notin T \cup Q$, $w_i \in T^*QT^*$ are configurations of M, w_1 is an initial configuration of the form $q_0\Sigma^*$, w_{2n} is an halting, that is, accepting configuration, and w_{i+1} is the successor configuration of w_i.

The set of *invalid computations* $\mathrm{INVALC}_i(M)$, for $i \in \{A, C, R\}$, is the complement of $\mathrm{VALC}_i(M)$ with respect to the alphabet $\{\#, \$\} \cup T \cup Q$.

Later we exploit a result on the following decomposition of $\mathrm{VALC}_A(M)$: $\mathrm{VALC}_{A1}(M)$ is the set of strings of the form $\$w_1\$w_2^R\$\cdots\$w_{2n-1}\$w_{2n}^R\$$, where w_1 is an initial and w_{2n} is an accepting configuration, and w_{2i+1} is the successor configuration of w_{2i}, for $1 \leq i \leq n-1$. $\mathrm{VALC}_{A2}(M)$ is the set of strings of the form $\$w_1\$w_2^R\$\cdots\$w_{2n-1}\$w_{2n}^R\$$, where w_1 is an initial and w_{2n} is an accepting configuration, and w_{2i} is the successor configuration of w_{2i-1}, for $1 \leq i \leq n$.

The next lemma summarizes some of the important properties of valid computations.

Lemma 1.83. *Let M be a Turing machine and $i \in \{A, C, R\}$.*

(1) If $L(M)$ is finite, then $VALC_i(M)$ is finite.
(2) If $L(M)$ is finite, then $INVALC_i(M)$ is regular.
(3) If $L(M)$ is infinite, then $VALC_i(M)$ is not context free.
(4) If $L(M)$ is infinite, then $INVALC_i(M)$ is not regular.
(5) $VALC_R(M)$ can be represented by the intersection of two deterministic linear context-free languages, such that both deterministic pushdown automata and both linear context-free grammars can effectively be constructed from M.
(6) $VALC_A(M)$ can be represented by the intersection of two deterministic context-free languages, such that both deterministic pushdown automata can effectively be constructed from M.
(7) $VALC_{A1}(M)$ and $VALC_{A2}(M)$ are deterministic context-free languages, such that their deterministic pushdown automata can effectively be constructed from M.
(8) $INVALC_i(M)$ is a linear context-free language, such that its grammar can effectively be constructed from M.

Assertions (1), (2), (4), (6), and (7) are immediate observations. Assertion (3) is shown by pumping lemma [58]. The two deterministic linear context-free languages for (5) are constructed in [2]. For INVALC$_A$, assertion (8) has been shown in [58]. Similarly, it can be proved for INVALC$_R$. For INVALC$_C$ observe that the complement of the language $\{ w\$w \mid w \in \{a,b\}^* \}$ is linear context free [34, 123].

Before we discuss some more applications and results in detail, we turn to the generalized and unified form of the second technique that emerges from known proofs.

Theorem 1.84. *Let $\mathcal{S}_1$ and $\mathcal{S}_2$ be two descriptional systems, c_1 be a measure for $\mathcal{S}_1$ and c_2 be an sn-measure for $\mathcal{S}_2$. If there exists a recursive function $\varphi : \mathbb{N} \to \mathbb{N}$, such that given an arbitrary Turing machine M, (i) there exists an effective procedure to construct a descriptor D_1 in $\mathcal{S}_1$, (ii) if M halts on blank tape, then D_1 has an equivalent descriptor in $\mathcal{S}_2$, and for all equivalent descriptors D_2 in $\mathcal{S}_2$ it holds $\varphi(c_2(D_2)) \geq t$, where t is the number of tape cells used by M, then the trade-off between $\mathcal{S}_1$ and $\mathcal{S}_2$ is non-recursive.*

Again, let us give evidence that the theorem is true. Assume contrarily that the trade-off is bounded by some recursive function f. Given some Turing machine M, we first construct a descriptor D_1 in $\mathcal{S}_1$. Since c_1 and f are recursive, the value $f(c_1(D_1))$ can be computed. Next, we can recursively enumerate the finite number of descriptors D_2 in $\mathcal{S}_2$, whose underlying alphabet is $\text{alph}(D_1)$ and whose size is at most $f(c_1(D_1))$. Since c_2 and φ are recursive, their maximum value $t' = \varphi(c_2(D_2))$ can be computed. If Turing machine M halts on blank tape, then there is at least one descriptor D_2 in the list such that $\varphi(c_2(D_2)) \geq t$, where t is the number of tape cells used by M. Since $t' \geq \varphi(c_2(D_2)) \geq t$ it suffices to simulate M on blank tape until it exceeds the tape cell bound t', or it runs into a loop without exceeding the tape cell bound, or it halts, in order to decide whether M halts on blank tape at all. From the contradiction follows that the trade-off is non-recursive.

The preconditions of the previous theorem are different compared with Theorem 1.80. In particular, it is not requested that the descriptors can effectively be converted into Turing machines, and the descriptional systems need not to be for recursive languages.

The next trade-off exploits the following crucial lemma on the size of deterministic pushdown automata [133].

Lemma 1.85. *If for some deterministic pushdown automaton A with state set Q and set of stack symbols T the string w is the shortest string such that wa and wb are accepted, then there is a positive constant k such that $|Q| \cdot |T| \geq (\log |w|)^k$.*

Example 1.86. The trade-off between unambiguous context-free grammars and deterministic pushdown automata is non-recursive.

Let M be an arbitrary Turing machine with state set Q, tape alphabet T, and initial state q_0. From M two deterministic pushdown automata A_1 and A_2 for the languages $\text{VALC}_{A1}(M)$ and $\text{VALC}_{A2}(M)$ can effectively be constructed. Since the deterministic context-free languages are effectively closed under intersection with regular sets, the languages $L_1 = \$q_0\$(T^*QT^*\$)^* \cap \text{VALC}_{A1}(M)$ and $L_2 = \$q_0\$(T^*QT^*\$)^* \cap \text{VALC}_{A2}(M)$ are also effective deterministic context-free languages. Let a and b be new symbols. Then we can construct an unambiguous context-free grammar D_1 for the language $L = L_1a \cup L_2b$.

Now assume that M halts on empty input. Then the intersection $L_1 \cap L_2$ contains exactly one string v, which is the valid computation of M on empty input. Moreover, there does not exist a string of length $2|v|$ which is a prefix in both languages. So, by inspecting the first $2|v|$ input symbols a deterministic pushdown automaton can decide to which language the input may still possibly belong. We conclude that there exists a deterministic pushdown automaton D_2 for L. By Lemma 1.85 we obtain that the product of the number of states and the number of stack symbols of any equivalent deterministic pushdown automaton is greater than $(\log |v|)^k$, for some positive k. Therefore, the recursive function φ for the application of Theorem 1.84 can easily be determined. □

1.5.2 *A Compilation of Non-Recursive Trade-Offs*

Before we turn to collect some important results in a compilation of non-recursive trade-offs, we draw the attention to a general question in connection with descriptional complexity. We have seen that there is a non-recursive trade-off between linear context-free languages and finite automata. On the other hand, the trade-off between deterministic context-free languages and finite automata is recursive [129]. So, what makes the difference between linear and deterministic context-free languages? Though the answer might be the power of nondeterminism, a closer look at the problem might clarify descriptional complexity to be a finer apparatus compared with computational complexity. This observation is emphasized by showing that, for example, between two separated Turing machine space classes there is always a non-recursive trade-off (see also [60]).

Example 1.87. Denote the languages accepted by deterministic Turing machines obeying a space bound $s(n)$ by $\mathsf{DSPACE}(s(n))$. If $\mathsf{DSPACE}(s_1(n)) \supset \mathsf{DSPACE}(s_2(n))$ for a constructible bound s_1, then the trade-off between s_1-space bounded Turing machines and s_2-space bounded Turing machines is non-recursive. □

The example can be shown with the help of Theorem 1.80. It can be modified to work for several other Turing machine classes which are separated by bounding some resource. For example, $\mathsf{P} \neq \mathsf{NP}$ if and only if the trade-off between NP and P is non-recursive (see [60–62] for further relations between descriptional and computational complexity).

Now we turn to results that deal mainly with descriptional systems at the lower end of the Chomsky hierarchy. A cornerstone of descriptional complexity is the result of Meyer and Fischer [109] who showed for the first time a non-recursive trade-off. It appears between context-free grammars and finite automata. Nowadays, we can derive that result immediately from the non-recursive trade-off between linear context-free grammars and finite automata, but originally, the proof follows the scheme presented in Theorem 1.84.

Theorem 1.88. *The trade-off between context-free grammars and finite automata is non-recursive.*

Since, for example, the sizes of context-free grammars and pushdown automata are recursively related, there is a non-recursive trade-off between pushdown automata and finite automata, too. Similarly, this remark holds for several results below. Once a non-recursive trade-off has been shown, it is interesting to consider language families in between. We know already by Example 1.86 that there is a non-recursive trade-off between unambiguous context-free grammars and deterministic pushdown automata. Considering the remaining gap between general and unambiguous context-free grammars we encounter the problem that unambiguity is not semi-decidable for context-free grammars. Therefore, we cannot enumerate the unambiguous context-free grammars. This implies that there does not exist any sn-measure for them and, thus, neither Theorem 1.80 nor Theorem 1.84 can be applied directly. However, by a variant of the technique of Theorem 1.84 the gap has been closed in [126]. The proof uses Ogden's lemma [113] in order to solve the crucial part to show that the sizes of unambiguous grammars depend on the lengths of strings in certain witness languages.

Theorem 1.89. *The trade-off between context-free grammars and unambiguous context-free grammars is non-recursive.*

By measuring the amount of ambiguity and nondeterminism in pushdown automata in [12] and [64] infinite hierarchies in between the deterministic and nondeterministic context-free languages are obtained. The classes of pushdown automata with ambiguity and branching bounded by k are denoted by PDA($\alpha \leq k$) and PDA($\beta \leq k$), where branching is a measure of nondeterminism. If both resources are bounded at the same time, we write PDA($\alpha \leq k, \beta \leq k'$). Intuitively, the corresponding language families are close together. Nevertheless, there are non-recursive trade-offs between the levels of the hierarchies.

Theorem 1.90. *Let $k \geq 1$ be an integer. Then the following trade-offs are non-recursive:*

(1) between PDA($\alpha \leq k+1, \beta \leq k+1$) and PDA($\alpha \leq k$), and
(2) between PDA($\alpha \leq 1, \beta \leq k+1$) and PDA($\beta \leq k$).

The proofs of both theorems are similar generalizations of the proof of Theorem 1.89. They follow the scheme of Theorem 1.84.

In [59, 60] simple new proofs of some of the presented theorems have been given. The technique of these proofs follows the scheme of Theorem 1.80. Furthermore, Hartmanis [59] raised the question whether the trade-off between two descriptional systems is caused by the fact that in one system it can be proved what is accepted, but that no such proofs are possible in the other system. For example, consider descriptional systems for the deterministic context-free languages. It is easy to verify whether a given pushdown automaton is deterministic, but there is no uniform way to verify that a nondeterministic pushdown automaton accepts a deterministic context-free language. Sticking with this example, one may ask whether the trade-off is affected if so-called *verified nondeterministic pushdown automata* are considered which come with an attached proof that they accept deterministic languages. The following theorem summarizes the results.

Theorem 1.91. *The following trade-offs are non-recursive:*

(1) between verified and deterministic pushdown automata,
(2) between pushdown automata and verified pushdown automata, and

(3) between verified ambiguous and unambiguous context-free grammars.

So far, some of the presented results can be shown by using some variant of the valid computations. In fact, whenever the expressive capacities of two systems can be separated by valid computations, the proof of non-recursive trade-offs is more or less immediate by an application of Theorem 1.80. For example, consider the Boolean closure of context-free languages and the context-free languages. The two systems are separated by VALC_A. The same is true for context-sensitive and context-free grammars and many other pairs of systems. The situation changes if the weaker system also contains a descriptor for the valid computations. Consider Example 1.87 which induces a non-recursive trade-off between space bounded Turing machine classes and deterministic context-sensitive grammars ($\mathsf{DSPACE}(n)$).

Coming back to the observation that space might be a rough measure of complexity, it is interesting and natural to investigate infinite hierarchies of separated language classes where, intuitively, the classes are closer together. For example, $\text{LL}(k+1)$ grammars are known to describe strictly more languages than $\text{LL}(k)$ grammars, that is, the length of the lookahead induces an infinite hierarchy. Nevertheless, the trade-offs between the levels of the hierarchy are recursive [5]. On the other hand, we have seen that there are non-recursive trade-offs between the hierarchy levels of unambiguous and nondeterministic pushdown automata. In [101] the trade-offs between $(k+1)$-turn and k-turn pushdown automata are investigated. The results are summarized as follows:

Theorem 1.92. *Let $k \geq 1$ be an integer. Then the following trade-offs are non-recursive:*

(1) between nondeterministic 1-turn pushdown automata and finite automata,
(2) between nondeterministic $(k+1)$-turn pushdown automata and nondeterministic k-turn pushdown automata,
(3) between nondeterministic pushdown automata and nondeterministic finite-turn pushdown automata, and
(4) between nondeterministic k-turn and deterministic k-turn pushdown automata.

So, there are infinite hierarchies such that between each two levels there are non-recursive trade-offs. Other results of such flavor have been obtained in [87] where deterministic and nondeterministic one-way k-head finite automata (k-DFA, k-NFA) are considered.

Theorem 1.93. *Let $k \geq 2$ be an integer. The trade-off between k-DFA and nondeterministic pushdown automata is non-recursive.*

Theorem 1.94. *Let $k \geq 1$ be an integer. Then the following trade-offs are non-recursive:*

(1) between $(k+1)$-DFA and k-DFA,
(2) between $(k+1)$-NFA and k-NFA, and
(3) between $(k+1)$-DFA and k-NFA.

Theorem 1.95. *Let $k \geq 2$ be an integer. Then the following trade-offs are non-recursive:*

(1) between 2-NFA and k-DFA, and
(2) between k-DFA and nondeterministic pushdown automata.

In [76] the problem whether there are non-recursive trade-offs between the levels of the hierarchies defined by two-way k-head finite automata (cf. also [86]) has been answered in the affirmative.

Theorem 1.96. *Let $k \geq 1$ be an integer. The trade-off between (non)deterministic (unary) two-way $(k+1)$-head finite automata and (non)deterministic (unary) two-way k-head finite automata is non-recursive.*

Now we briefly consider non-classical descriptional systems. In [65] *deterministic restarting automata*, an automaton model inspired from linguistics are investigated. Variants of deterministic and monotone restarting automata build a strict hierarchy whose top is characterized by the *Church-Rosser languages* and whose bottom is characterized by the deterministic context-free languages. It is shown that between PDAs and any level of the hierarchy there are non-recursive trade-offs. Interestingly, the converse is also true for the Church-Rosser languages. Moreover, there are non-recursive trade-offs between the family of Church-Rosser languages and any other level of the hierarchy.

Theorem 1.97. *The following trade-offs are non-recursive:*

(1) between nondeterministic (one-turn) pushdown automata and Church-Rosser languages (deterministic R(R)WW*-automata),*
(2) between Church-Rosser languages and nondeterministic pushdown automata,

(3) between Church-Rosser languages and deterministic pushdown automata,
(4) between deterministic RWW*-automata (Church-Rosser languages) and deterministic monotone* RRWW*-automata,*
(5) between deterministic RWW*- and deterministic* RRW*-automata,*
(6) between deterministic RWW*- and deterministic* RW*-automata,*
(7) between deterministic RWW*- and deterministic* RR*-automata, and*
(8) between deterministic RWW*- and deterministic* R*-automata.*

Metalinear CD grammar systems [24] are context-free CD grammar systems where each component consists of metalinear productions. The maximal number of nonterminals in an initial production defines the width of the CD grammar system. In [131] it is proved that there are non-recursive trade-offs between CD grammar systems of width $m + 1$ and m. Furthermore, non-recursive trade-offs appear between CD grammar systems of width m and $(2m - 1)$-linear context-free grammars.

Further results are known for *(one-way) cellular automata ((O)CA)* and *iterative arrays (IA)* [99, 100].

Theorem 1.98. *The following trade-offs are non-recursive:*

(1) between real-time OCA and finite automata,
(2) between real-time OCA and pushdown automata,
(3) between real-time CA and real-time OCA, and
(4) between linear-time OCA and real-time OCA.
(5) Between real-time IA and finite automata,
(6) between real-time IA and pushdown automata,
(7) between linear-time IA and real-time IA,
(8) between real-time IA and real-time OCA, and
(9) between real-time OCA and real-time IA.

The proofs all follow the scheme of Theorem 1.80. It is worth mentioning that results (1) and (2) of Theorem 1.97 as well as results (8) and (9) of Theorem 1.98 say that there are non-recursive trade-offs between one system and another system and *vice versa*.

Finally, the phenomenon of non-recursive trade-offs between descriptional systems is investigated in an abstract and more axiomatic fashion in [49]. The aim is to categorize non-recursive trade-offs by bounds on their growth rate, and to show how to deduce such bounds in general. Also criteria are identified which, in the spirit of abstract language theory, allow to deduce non-recursive tradeoffs from effective closure properties of language families on the one hand, and differences in the decidability status of basic decision problems on the other. A qualitative classification of non-recursive trade-offs is developed in order to obtain a better understanding of this very fundamental behavior of descriptional systems.

References

1. Aho, A. V., Hopcroft, J. E. and Ullman, J. D. (1974). *The Design and Analysis of Computer Algorithms* (Addison-Wesley).
2. Baker, B. S. and Book, R. V. (1974). Reversal-bounded multipushdown machines, *J. Comput. System Sci.* **8**, pp. 315–332.
3. Berman, P. (1980). A note on sweeping automata, in *International Colloquium on Automata, Languages and Programming (ICALP 1980)*, *LNCS*, Vol. 85 (Springer), pp. 91–97.
4. Berman, P. and Lingas, A. (1977). On the complexity of regular languages in terms of finite automata, Tech. Rep. 304, Polish Academy of Sciences.
5. Bertsch, E. and Nederhof, M.-J. (2001). Size/lookahead tradeoff for LL(k)-grammars, *Inform. Process. Lett.* **80**, pp. 125–129.
6. Berwanger, D., Dawar, A., Hunter, P. and Kreutzer, S. (2006). Dag-width and parity games, in *Theoretical Aspects of Computer Science (STACS 2006)*, *LNCS*, Vol. 3884 (Springer), pp. 524–536.
7. Berwanger, D. and Grädel, E. (2005). Entanglement – a measure for the complexity of directed graphs with applications to logic and games, in *Logic for Programming, Artificial Intelligence, and Reasoning (LPAR 2004)*, *LNCS*, Vol. 3452 (Springer), pp. 209–223.
8. Birget, J.-C. (1992a). Intersection and union of regular languages and state complexity, *Inform. Process. Lett.* **43**, pp. 185–190.
9. Birget, J.-C. (1992b). Positional simulation of two-way automata: Proof of a conjecture of R. Kannan and generalizations, *J. Comput. System Sci.* **45**, pp. 154–179.
10. Birget, J.-C. (1993). State-complexity of finite-state devices, state compressibility and incompressibility, *Math. Systems Theory* **26**, pp. 237–269.
11. Birget, J.-C. (1996). Two-way automata and length-preserving homomorphisms, *Math. Systems Theory* **29**, pp. 191–226.
12. Borchardt, I. (1992). *Nonrecursive Tradeoffs between context-free grammars with different constant ambiguity*, Diploma thesis, Universität Frankfurt (in German).
13. Bordihn, H., Holzer, M. and Kutrib, M. (2005). Unsolvability levels of operation problems for subclasses of context-free languages, *Int. J. Found. Comput. Sci.* **16**, pp. 423–440.
14. Bordihn, H., Holzer, M. and Kutrib, M. (2009). Determinization of finite automata accepting subregular languages, *Theoret. Comput. Sci.* **410**, pp. 3209–3222.

15. Brzozowski, J. A. and Leiss, E. L. (1980). On equations for regular languages, finite automata, and sequential networks, *Theoret. Comput. Sci.* **10**, pp. 19–35.
16. Bucher, W. (1981). A note on a problem in the theory of grammatical complexity, *Theoret. Comput. Sci.* **14**, pp. 337–344.
17. Bucher, W., Maurer, H. A. and Čulik II, K. (1984). Context-free complexity of finite languages, *Theoret. Comput. Sci.* **28**, pp. 277–285.
18. Bucher, W., Maurer, H. A., Čulik II, K. and Wotschke, D. (1981). Concise description of finite languages, *Theoret. Comput. Sci.* **14**, pp. 227–246.
19. Cerný, A. (1977). Complexity and minimality of context-free grammars and languages, in *Mathematical Foundations of Computer Science (MFCS 1977)*, *LNCS*, Vol. 53 (Springer), pp. 263–271.
20. Champarnaud, J.-M., Ouardi, F. and Ziadi, D. (2007). Normalized expressions and finite automata, *Int. J. Algebra Comput.* **17**, pp. 141–154.
21. Chandra, A., Kozen, D. and Stockmeyer, L. (1981). Alternation, *J. ACM* **21**, pp. 114–133.
22. Chrobak, M. (1986). Finite automata and unary languages, *Theoret. Comput. Sci.* **47**, pp. 149–158.
23. Chrobak, M. (2003). Errata to "finite automata and unary languages", *Theoret. Comput. Sci.* **302**, pp. 497–498.
24. Csuhaj-Varjú, E., Dassow, J., Kelemen, J. and Păun, G. (1984). *Grammar Systems: A Grammatical Approach to Distribution and Cooperation* (Gordon and Breach).
25. Domaratzki, M., Pighizzini, G. and Shallit, J. (2002). Simulating finite automata with context-free grammars, *Inform. Process. Lett.* **84**, pp. 339–344.
26. Domaratzki, M. and Salomaa, K. (2006). Lower bounds for the transition complexity of NFAs, in *Mathematical Foundations of Computer Science (MFCS 2006)*, *LNCS*, Vol. 4162 (Springer), pp. 315–326.
27. Eggan, L. C. (1963). Transition graphs and the star height of regular events, *Michigan Math. J.* **10**, pp. 385–397.
28. Ehrenfeucht, A. and Zeiger, H. P. (1976). Complexity measures for regular expressions, *J. Comput. System Sci.* **12**, pp. 134–146.
29. Ellul, K., Krawetz, B., Shallit, J. and Wang, M.-W. (2005). Regular expressions: New results and open problems, *J. Autom., Lang. Comb.* **10**, pp. 407–437.
30. Fellah, A., Jürgensen, H. and Yu, S. (1990). Constructions for alternating finite automata, *Internat. J. Comput. Math.* **35**, pp. 117–132.
31. Geffert, V., Mereghetti, C. and Pighizzini, G. (2003). Converting two-way nondeterministic unary automata into simpler automata, *Theoret. Comput. Sci.* **295**, pp. 189–203.
32. Gelade, W. and Neven, F. (2008). Succinctness of the complement and intersection of regular expressions, in *Theoretical Aspects of Computer Science (STACS 2008)*, *Dagstuhl Seminar Proceedings*, Vol. 08001 (IBFI, Schloss Dagstuhl, Germany), pp. 325–336.
33. Gill, A. and Kou, L. T. (1974). Multiple-entry finite automata, *J. Comput. System Sci.* **9**, pp. 1–19.

34. Ginsburg, S. and Greibach, S. A. (1966). Deterministic context-free languages, *Inform. Control* **9**, pp. 620–648.
35. Ginsburg, S. and Rice, H. G. (1962). Two families of languages related to ALGOL, *J. ACM* **9**, pp. 350–371.
36. Glaister, I. and Shallit, J. (1996). A lower bound technique for the size of nondeterministic finite automata, *Inform. Process. Lett.* **59**, pp. 75–77.
37. Glushkov, V. M. (1961). The abstract theory of automata, *Russian Math. Surveys* **16**, pp. 1–53.
38. Goldstine, J., Kappes, M., Kintala, C. M. R., Leung, H., Malcher, A. and Wotschke, D. (2002). Descriptional complexity of machines with limited resources, *J. UCS* **8**, pp. 193–234.
39. Goldstine, J., Leung, H. and Wotschke, D. (1992). On the relation between ambiguity and nondeterminism in finite automata, *Inform. Comput.* **100**, pp. 261–270.
40. Goldstine, J., Price, J. K. and Wotschke, D. (1982a). On reducing the number of states in a PDA, *Math. Systems Theory* **15**, pp. 315–321.
41. Goldstine, J., Price, J. K. and Wotschke, D. (1982b). A pushdown automaton or a context-free grammar – which is more economical? *Theoret. Comput. Sci.* **18**, pp. 33–40.
42. Goldstine, J., Price, J. K. and Wotschke, D. (1993). On reducing the number of stack symbols in a PDA, *Math. Systems Theory* **26**, pp. 313–326.
43. Greibach, S. A. (1965). A new normal-form theorem for context-free phrase structure grammars, *J. ACM* **12**, pp. 42–52.
44. Greibach, S. A. (1973). The hardest context-free language, *SIAM J. Comput.* **2**, pp. 304–310.
45. Gruber, H. and Gulan, S. (2009). Simplifying regular expressions: A quantitative perspective, IFIG Research Report 0904, Institut für Informatik, Justus-Liebig-Universität, Gießen, Germany.
46. Gruber, H. and Holzer, M. (2005). A note on the number of transitions of nondeterministic finite automata, in *Theorietag Automaten und Formale Sprachen* (Universität Tübingen, Tübingen, Germany), pp. 24–25.
47. Gruber, H. and Holzer, M. (2006). Finding lower bounds for nondeterministic state complexity is hard, in *Developments in Language Theory (DLT 2006)*, *LNCS*, Vol. 4036 (Springer), pp. 363–374.
48. Gruber, H. and Holzer, M. (2008). Provably shorter regular expressions from deterministic finite automata, in *Developments in Language Theory (DLT 2008)*, *LNCS*, Vol. 5257 (Springer), pp. 383–395.
49. Gruber, H., Holzer, M. and Kutrib, M. (2009). On measuring non-recursive trade-offs, in *Descriptional Complexity of Formal Systems (DCFS 2009)* (Otto-von-Guericke-Universität Magdeburg, Germany), pp. 187–198.
50. Gruber, H. and Johannsen, J. (2008). Optimal lower bounds on regular expression size using communication complexity, in *Foundations of Software Science and Computational Structures (FoSSaCS 2008)*, *LNCS*, Vol. 4962 (Springer), pp. 273–286.
51. Gruska, J. (1967). On a classification of context-free languages, *Kybernetica* **3**, pp. 22–29.

52. Gruska, J. (1969). Some classifications of context-free languages, *Inform. Control* **14**, pp. 152–179.
53. Gruska, J. (1971a). A characterization of context-free languages, *J. Comput. System Sci.* **5**, pp. 353–364.
54. Gruska, J. (1971b). Complexity and unambiguity of context-free grammars and languages, *Inform. Control* **18**, pp. 502–519.
55. Gruska, J. (1976). Descriptional complexity (of languages) - a short survey, in *Mathematical Foundations of Computer Science (MFCS 1976)*, *LNCS*, Vol. 45 (Springer), pp. 65–80.
56. Gulan, S. and Fernau, H. (2008). An optimal construction of finite automata from regular expressions, in *Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2008)*, *Dagstuhl Seminar Proceedings*, Vol. 08002 (IBFI, Schloss Dagstuhl, Germany), pp. 211–222.
57. Harrison, M. A. (1978). *Introduction to Formal Language Theory* (Addison-Wesley).
58. Hartmanis, J. (1967). Context-free languages and Turing machine computations, *Proc. Symposia in Applied Mathematics* **19**, pp. 42–51.
59. Hartmanis, J. (1980). On the succinctness of different representations of languages, *SIAM J. Comput.* **9**, pp. 114–120.
60. Hartmanis, J. (1983). On Gödel speed-up and succinctness of language representations, *Theoret. Comput. Sci.* **26**, pp. 335–342.
61. Hartmanis, J. and Baker, T. P. (1979a). Relative succinctness of representations of languages and separation of complexity classes, in *Mathematical Foundations of Computer Science (MFCS 1979)*, *LNCS*, Vol. 74 (Springer), pp. 70–88.
62. Hartmanis, J. and Baker, T. P. (1979b). Succinctness, verifiability and determinism in representations of polynomial-time languages, in *Foundations and Computer Science (FOCS 1979)* (IEEE), pp. 392–396.
63. Hashiguchi, K. (1988). Algorithms for determining relative star height and star height, *Inform. Comput.* **78**, pp. 124–169.
64. Herzog, C. (1997). Pushdown automata with bounded nondeterminism and bounded ambiguity, *Theoret. Comput. Sci.* **181**, pp. 141–157.
65. Holzer, M., Kutrib, M. and Reimann, J. (2007). Non-recursive trade-offs for deterministic restarting automata, *J. Autom., Lang. Comb.* **12**, pp. 195–213.
66. Holzer, M., Salomaa, K. and Yu, S. (2001). On the state complexity of k-entry deterministic finite automata, *J. Autom., Lang. Comb.* **6**, pp. 453–466.
67. Hopcroft, J. E. and Ullman, J. D. (1979). *Introduction to Automata Theory, Languages, and Computation* (Addison-Wesley).
68. Hromkovič, J. (1997). *Communication Complexity and Parallel Computing* (Springer).
69. Hromkovič, J. and Schnitger, G. (2005). NFAs with and without ϵ-transitions, in *International Colloquium on Automata, Languages and Programming (ICALP 2005)*, *LNCS*, Vol. 3580 (Springer), pp. 385–396.
70. Hromkovič, J., Seibert, S. and Wilke, T. (2001). Translating regular expres-

sions into small ϵ-free nondeterministic finite automata, *J. Comput. System Sci.* **62**, pp. 565–588.

71. Hromkovič, J., Seibert, S., Karhumäki, J., Klauck, H. and Schnitger, G. (2002). Communication complexity method for measuring nondeterminism in finite automata, *Inform. Comput.* **172**, pp. 202–217.
72. Ilie, L. and Yu, S. (2003). Follow automata, *Inform. Comput.* **186**, pp. 140–162.
73. Iwama, K., Kambayashi, Y. and Takaki, K. (2000). Tight bounds on the number of states of DFAs that are equivalent to n-state NFAs, *Theoret. Comput. Sci.* **237**, pp. 485–494.
74. Johnson, T., Robertson, N., Seymour, P. D. and Thomas, R. (2001). Directed tree-width, *J. Combinatorial Theory* **82**, pp. 138–154.
75. Kannan, R. (1983). Alternation and the power of nondeterminism, in *Symposium on Theory of Computing (STOC 1983)* (ACM Press), pp. 344–346.
76. Kapoutsis, C. A. (2004). From $k + 1$ to k heads the descriptive trade-off is non-recursive, in *Descriptional Complexity of Formal Systems (DCFS 2004)*, pp. 213–224.
77. Kapoutsis, C. A. (2005). Removing bidirectionality from nondeterministic finite automata, in *Mathematical Foundations of Computer Science (MFCS 2005)*, *LNCS*, Vol. 3618 (Springer), pp. 544–555.
78. Kapoutsis, C. A. (2006). *Algorithms and Lower Bounds in Finite Automata Size Complexity*, Phd thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology.
79. Kappes, M. (2000). Descriptional complexity of deterministic finite automata with multiple initial states, *J. Autom., Lang. Comb.* **5**, pp. 269–278.
80. Kelemenová, A. (1981). Grammatical levels of the position restricted grammars, in *Mathematical Foundations of Computer Science (MFCS 1981)*, *LNCS*, Vol. 118 (Springer), pp. 347–359.
81. Kelemenová, A. (1984). Complexity of normal form grammars, *Theoret. Comput. Sci.* **28**, pp. 299–314.
82. Kintala, C. M. and Wotschke, D. (1980). Amounts of nondeterminism in finite automata, *Acta Inform.* **13**, pp. 199–204.
83. Kirsten, D. (2005). Distance desert automata and the star height problem, *RAIRO Inform. Théor.* **39**, pp. 455–509.
84. Klauck, H. (1998). Lower bounds for computation with limited nondeterminism, in *Conference on Computational Complexity* (IEEE), pp. 141–152.
85. Kleene, S. C. (1956). Representation of events in nerve nets and finite automata, in *Automata Studies* (Princeton University Press), pp. 3–42.
86. Kutrib, M. (2005a). On the descriptional power of heads, counters, and pebbles, *Theoret. Comput. Sci.* **330**, pp. 311–324.
87. Kutrib, M. (2005b). The phenomenon of non-recursive trade-offs, *Int. J. Found. Comput. Sci.* **16**, pp. 957–973.
88. Kutrib, M. and Reimann, J. (2008). Succinct description of regular languages by weak restarting automata, *Inform. Comput.* **206**, pp. 1152–1160.

89. Landau, E. (1903). Über die Maximalordnung der Permutationen gegebenen Grades, *Archiv der Math. und Phys.* **3**, pp. 92–103.
90. Landau, E. (1909). *Handbuch der Lehre von der Verteilung der Primzahlen* (Teubner).
91. Leiss, E. L. (1980). Constructing a finite automaton for a given regular expression, *Bull. EATCS* **10**, pp. 54–59.
92. Leiss, E. L. (1981). Succinct representation of regular languages by Boolean automata, *Theoret. Comput. Sci.* **13**, pp. 323–330.
93. Leiss, E. L. (1985). Succinct representation of regular languages by Boolean automata. II, *Theoret. Comput. Sci.* **38**, pp. 133–136.
94. Leung, H. (1998). Separating exponentially ambiguous finite automata from polynomially ambiguous finite automata, *SIAM J. Comput.* **27**, pp. 1073–1082.
95. Leung, H. (2001). Tight lower bounds on the size of sweeping automata, *J. Comput. System Sci.* **63**, pp. 384–393.
96. Leung, H. and Wotschke, D. (2000). On the size of parsers and LR(k)-grammars, *Theoret. Comput. Sci.* **242**, pp. 59–69.
97. Lifshits, Y. (2003). A lower bound on the size of ϵ-free NFA corresponding to a regular expression, *Inform. Process. Lett.* **85**, pp. 293–299.
98. Lupanov, O. B. (1963). A comparison of two types of finite sources (in Russian), *Problemy Kybernetiki* **9**, pp. 321–326, German translation (1966): Über den Vergleich zweier Typen endlicher Quellen, *Probleme der Kybernetik* **6**, pp. 328-335.
99. Malcher, A. (2002). Descriptional complexity of cellular automata and decidability questions, *J. Autom., Lang. Comb.* **7**, pp. 549–560.
100. Malcher, A. (2004). On the descriptional complexity of iterative arrays, *IEICE Trans. Inf. Syst.* **E87-D**, pp. 721–725.
101. Malcher, A. (2007). On recursive and non-recursive trade-offs between finite-turn pushdown automata, *J. Autom., Lang. Comb.* **12**, pp. 265–277.
102. Malcher, A. and Pighizzini, G. (2007). Descriptional complexity of bounded context-free languages, in *Developments in Language Theory (DLT 2007)*, *LNCS*, Vol. 4588 (Springer), pp. 312–323.
103. Mandl, R. (1973). Precise bounds associated with the subset construction on various classes of nondeterministic finite automata, in *Princeton Conference on Information Sciences and Systems (CISS 1973)*, pp. 263–267.
104. McNaughton, R. (1969). The loop complexity of regular events, *Inform. Sci.* **1**, pp. 305–328.
105. McNaughton, R. and Yamada, H. (1960). Regular expressions and state graphs for automata, *IRE Trans. Elect. Comput.* **EC-9**, pp. 39–47.
106. McWhirter, I. P. (1971). Substitution expressions, *J. Comput. System Sci.* **5**, pp. 629–637.
107. Mereghetti, C. and Pighizzini, G. (2000). Two-way automata simulations and unary languages, *J. Autom., Lang. Comb.* **5**, pp. 287–300.
108. Mereghetti, C. and Pighizzini, G. (2001). Optimal simulations between

unary automata, *SIAM J. Comput.* **30**, pp. 1976–1992.

109. Meyer, A. R. and Fischer, M. J. (1971). Economy of description by automata, grammars, and formal systems, in *Switching and Automata Theory (SWAT 1971)* (IEEE), pp. 188–191.
110. Micali, S. (1981). Two-way deterministic finite automata are exponentially more succinct than sweeping automata, *Inform. Process. Lett.* **12**, pp. 103–105.
111. Minsky, M. L. (1967). *Computation: Finite and Infinite Machines* (Prentice-Hall).
112. Moore, F. R. (1971). On the bounds for state-set size in the proofs of equivalence between deterministic, nondeterministic, and two-way finite automata, *IEEE Trans. Comput.* **20**, pp. 1211–1214.
113. Ogden, W. F. (1968). A helpful result for proving inherent ambiguity, *Math. Systems Theory* **2**, pp. 191–194.
114. Păun, G. (1984). *Probleme actuale în teoria limbajelor formale* (Editura ştiinţifică enciclopedică).
115. Pighizzini, G. (2008). Deterministic pushdown automata and unary languages, in *Implementation and Application of Automata (CIAA 2008)*, *LNCS*, Vol. 5148 (Springer), pp. 232–241.
116. Pighizzini, G., Shallit, J. and Wang, M.-W. (2002). Unary context-free grammars and pushdown automata, descriptional complexity and auxiliary space lower bounds, *J. Comput. System Sci.* **65**, pp. 393–414.
117. Pirická, A. (1974). Complexity and normal forms of contex-free languages, in *Mathematical Foundations of Computer Science (MFCS 1974)*, *LNCS*, Vol. 28 (Springer), pp. 292–297.
118. Pirická-Kelemenová, A. (1975). Greibach normal form complexity, in *Mathematical Foundations of Computer Science (MFCS 1975)*, *LNCS*, Vol. 32 (Springer), pp. 344–350.
119. Rabin, M. O. and Scott, D. (1959). Finite automata and their decision problems, *IBM J. Res. Dev.* **3**, pp. 114–125.
120. Ravikumar, B. and Ibarra, O. H. (1989). Relating the type of ambiguity of finite automata to the succinctness of their representation, *SIAM J. Comput.* **18**, pp. 1263–1282.
121. Sakarovitch, J. (2006). The language, the expression, and the (small) automaton, in *Implementation and Application of Automata (CIAA 2005)*, *LNCS*, Vol. 3845 (Springer), pp. 15–30.
122. Sakoda, W. J. and Sipser, M. (1978). Nondeterminism and the size of two way finite automata, in *Symposium on Theory of Computing (STOC 1978)* (ACM Press), pp. 275–286.
123. Salomaa, A. (1973). *Formal Languages* (Academic Press).
124. Salomaa, K. and Yu, S. (1997). NFA to DFA transformation for finite languages over arbitrary alphabets, *J. Autom., Lang. Comb.* **2**, pp. 177–186.
125. Schmidt, E. M. (1978). *Succinctness of Dscriptions of Context-Free, Regular and Finite Languages*, Ph.D. thesis, Cornell University, Ithaca, New York.
126. Schmidt, E. M. and Szymanski, T. G. (1977). Succinctness of descriptions

of unambiguous context-free languages, *SIAM J. Comput.* **6**, pp. 547–553.
127. Shepherdson, J. C. (1959). The reduction of two-way automata to one-way automata, *IBM J. Res. Dev.* **3**, pp. 198–200.
128. Sipser, M. (1980). Lower bounds on the size of sweeping automata, *J. Comput. System Sci.* **21**, pp. 195–202.
129. Stearns, R. E. (1967). A regularity test for pushdown machines, *Inform. Control* **11**, pp. 323–340.
130. Stearns, R. E. and Hunt III, H. B. (1985). On the equivalence and containment problems for unambiguous regular expressions, regular grammars, and finite automata, *SIAM J. Comput.* **14**, pp. 598–611.
131. Sunckel, B. (2005). On the descriptional complexity of metalinear CD grammar systems, *Int. J. Found. Comput. Sci.* **16**, pp. 1011–1025.
132. Valiant, L. G. (1975). Regularity and related problems for deterministic pushdown automata, *J. ACM* **22**, pp. 1–10.
133. Valiant, L. G. (1976). A note on the succinctness of descriptions of deterministic languages, *Inform. Control* **32**, pp. 139–145.
134. Veloso, P. A. S. and Gill, A. (1979). Some remarks on multiple-entry finite automata, *J. Comput. System Sci.* **18**, pp. 304–306.
135. Yntema, M. K. (1971). Cap expressions for context-free languages, *Inform. Control* **18**, pp. 311–318.
136. Yu, S. (1997). Regular languages, in *Handbook of Formal Languages*, Vol. 1, chap. 2 (Springer), pp. 41–110.
137. Yu, S. (2001). State complexity of regular languages, *J. Autom., Lang. Comb.* **6**, pp. 221–234.

Chapter 2

Classifying All Avoidable Sets of Partial Words of Size Two

Francine Blanchet-Sadri

Department of Computer Science,
University of North Carolina,
P.O. Box 26170, Greensboro, NC 27402–6170, USA,
E-mail: `blanchet@uncg.edu`

Brandon Blakeley and Josh Gunter

Department of Computer Sciences,
The University of Texas at Austin,
1 University Station C0500 Taylor Hall 2.124,
Austin, TX 78712–0233, USA

Sean Simmons

Department of Mathematics,
The University of Texas at Austin,
2515 Speedway Rm 8, Austin, TX 78712–0233, USA

Eric Weissenstein

Department of Mathematical Sciences,
Rensselaer Polytechnic Institute,
Amos Eaton 301, 110 8th Street, Troy, NY 12180, USA

Partial words are sequences that may have some undefined positions denoted by $\diamond$'s, where a $\diamond$ matches every letter of the alphabet. This chapter is related to the problem of characterizing the two-element unavoidable sets of partial words over an arbitrary alphabet, which reduces to the problem of characterizing the unavoidable sets of the form

$$X_{m_1,\ldots,m_k|n_1,\ldots,n_l} = \{a\diamond^{m_1}a\cdots a\diamond^{m_k}a, b\diamond^{n_1}b\cdots b\diamond^{n_l}b\}$$

where a, b are distinct letters of the alphabet and $m_1, \ldots, m_k, n_1, \ldots, n_l$ are nonnegative integers. We prove a conjecture regarding the case $k = 1, l = 2$, which was identified by Blanchet-Sadri *et al.* that, if proven, suffices to classify all sets $X_{m_1,\ldots,m_k|n_1,\ldots,n_l}$ such that $k \geq 2, l \geq 2$ as avoidable or unavoidable, and which involves six systems of inequalities on m, n_1 and n_2. We classify some of the previously unclassified sets by identifying common two-sided infinite avoiding words and exhibiting exactly which sets these words can avoid. In the process, we introduce a new technique to reduce a set if the period of an avoiding word is known. Finally using Cayley graphs, we are able to classify all remaining sets.

2.1 Introduction

A set of (full) words (ones without undefined positions) X over a finite alphabet A is *unavoidable* if no two-sided infinite word over A avoids X, that is, X is unavoidable if every two-sided infinite word over A has a factor in X. This concept was introduced in 1983 in an attempt to characterize which context-free languages are also rational [10]. For instance, the set $\{a, bbb\}$ is unavoidable, for if a two-sided infinite word w does not have a as a factor, then w consists only of b's. When X is finite, the following three statements are equivalent: (1) X is unavoidable; (2) there are only finitely many words in A^* with no member of X as a factor; and (3) no periodic two-sided infinite word avoids X. For other properties of unavoidable sets, we refer the reader to [14]. There, the main result is that there exists an explicit characterization of the unavoidable sets with a fixed cardinality. A vast literature exists on unavoidable sets, to mention a few [6, 9, 7, 11, 13, 15–17].

Partial words are sequences over a finite alphabet that may contain some undefined positions denoted by $\diamond$'s, where the $\diamond$ symbol is compatible with every symbol of the alphabet. Combinatorics on partial words was initiated by Berstel and Boasson [1], and has been investigated since then (see for example [3]). Unavoidable sets of partial words were introduced recently in [4]. In this context, a set of partial words X over A is unavoidable if every two-sided infinite full word over A has a factor compatible with an element of X. Partial words allow us to represent large sets of full words efficiently. This representation gives us new insights into the combinatorial structure of unavoidable sets of full words. Determining whether a finite set of partial words is avoidable is known to be NP-hard [5] in contrast with the known feasibility results for full words [8, 14, 15]. For other results on the complexity of deciding avoidability of sets of partial words, we refer the reader to [2].

In [4], the problem of characterizing unavoidable sets of partial words of cardinality two was initiated. If X is an unavoidable set, then every two-sided infinite unary word has a factor compatible with a member of X. In particular, X cannot have fewer elements than the alphabet. Thus if X has size two, then the alphabet is unary or binary. If the alphabet is unary, then X is unavoidable, so we will not consider the unary alphabet further. We hence assume that the alphabet is binary, say with distinct letters a and b. So one element of X is compatible with a factor of $a^{\mathbb{Z}}$ and the other element is compatible with a factor of $b^{\mathbb{Z}}$, since this is the only way to guarantee that both $a^{\mathbb{Z}}$ and $b^{\mathbb{Z}}$ will not avoid X. This shows that the classification of *all* the two-element unavoidable sets of partial words over an arbitrary alphabet reduces to the classification of *all* the two-element unavoidable sets over the binary alphabet $\{a, b\}$ of the form

$$X_{m_1,\dots,m_k|n_1,\dots,n_l} = \{a\diamond^{m_1}a\cdots a\diamond^{m_k}a, b\diamond^{n_1}b\cdots b\diamond^{n_l}b\} \tag{2.1}$$

where $m_1, \dots, m_k, n_1, \dots, n_l$ are nonnegative integers. Note that $\diamond$'s from the left and right ends of elements of unavoidable sets can be truncated. The problem is for which integers $m_1, \dots, m_k, n_1, \dots, n_l$ is $X_{m_1,\dots,m_k|n_1,\dots,n_l}$ unavoidable. Of course, the set $\{a, b\diamond^{n_1}b\cdots b\diamond^{n_l}b\}$ is always unavoidable, for if w is a two-sided infinite word which does not have a as a factor, then $w = b^{\mathbb{Z}}$. This handles the case where $k = 0$ (and symmetrically $l = 0$).

In [4], the authors gave a characterization of the special case of this problem when $k = 1$ and $l = 1$. They proposed a conjecture characterizing the case where $k = 1$ and $l = 2$, and proved one direction of the conjecture (the case $k = 2$ and $l = 1$ is symmetric). They then gave partial results towards the other direction, and in particular proved that the conjecture is easy to verify in a large number of cases. Finally, they showed that verifying this conjecture is sufficient for proving the avoidability of all sets where $k, l \geq 2$.

In this chapter, we complete the classification of all unavoidable sets of partial words of size two over any alphabet. Following the discussion above, we will do this by completing the classification of all the sets $X_{m_1,\dots,m_k|n_1,\dots,n_l}$. The contents of our chapter is as follows: In Section 2.2, we review basics on partial words and unavoidable sets. In Section 2.3, we identify a strict minimal period of an avoiding word for $X_{m|n}$. There, we reduce the conjecture of [4] (see Conjecture 2.9) regarding the sets $X_{m|n_1,n_2}$ to Conjecture 2.10 which involves six systems of inequalities on m, n_1 and n_2. In Section 2.4, we discuss a new technique for testing if a set is avoidable, the so-called canonical forms. In Section 2.5, we prove that Conjecture 2.10 is true using canonical forms and Cayley graphs. In Section 2.6, we show that all sets where $k = 1$ and $l \geq 4$ are avoidable, and we also classify all sets where $k = 1$ and $l = 3$. Finally in Section 2.7, we conclude with some remarks.

2.2 Preliminaries

We first review concepts on partial words. Let A be a fixed nonempty finite set called an *alphabet* whose elements we call *letters*. A *word* over A is a finite sequence of elements of A. We let A^* denote the set of words over A which, under the concatenation operation of words, forms a free monoid whose identity is the empty word, which we denote by ε.

A *partial word* u of length n (or $|u|$) over A can be defined as a function $u : [0..n-1] \to A_\diamond$, where $A_\diamond = A \cup \{\diamond\}$. For $0 \leq i < n$, if $u(i) \in A$, then i belongs to the *domain* of u, denoted $D(u)$, and if $u(i) = \diamond$, then i belongs to the *set of holes* of u, denoted $H(u)$. Whenever $H(u)$ is empty, u is a *full* word. We will refer to an occurrence of the symbol $\diamond$ as a *hole*. We let $A_\diamond^*$ denote the set of all partial words over A.

A *period* of a partial word u is a positive integer p such that $u(i) = u(j)$ whenever $i, j \in D(u)$ and $i \equiv j \pmod{p}$ (note that to simplify notation, we will often abbreviate the latter by $i \equiv j \bmod p$). In this case, we call u *p-periodic*. The smallest period of u is called the *minimal period* of u and is denoted by $p(u)$.

The partial word u is *contained* in the partial word v, denoted $u \subset v$, if $|u| = |v|$ and $u(i) = v(i)$ for all $i \in D(u)$. A nonempty partial word u is *primitive* if there exists no word v such that $u \subset v^n$ with $n \geq 2$. Two partial words u and v of equal length are *compatible*, denoted $u \uparrow v$, if $u(i) = v(i)$ whenever $i \in D(u) \cap D(v)$. In other words, u and v are compatible if there exists a partial word w such that $u \subset w$ and $v \subset w$, in which case we denote by $u \vee v$ the least upper bound of u and v ($u \subset (u \vee v)$ and $v \subset (u \vee v)$ and $D(u \vee v) = D(u) \cup D(v)$). For example, $u = aba\diamond\diamond$ and $v = a\diamond\diamond b\diamond$ are compatible, and $(u \vee v) = abab\diamond$.

Let $\sigma : A_\diamond^* \to A_\diamond^*$, where $\sigma(\varepsilon) = \varepsilon$ and $\sigma(au) = ua$ for all $u \in A_\diamond^*$

and $a \in A_\diamond$, be the *cyclic shift function.* The partial words u and v are *conjugates*, denoted $u \sim v$, if $v = \sigma^i(u)$ for some $i \geq 0$. The *conjugacy class* of a partial word u, denoted $[u]$, is the set of partial words $\{v \mid u \sim v\}$.

Lemma 2.1 ([18]**, p. 76).** *Let u be a full word of length n, and let $i \in [1..n-1]$. Then $\sigma^i(u) = u$ if and only if u is not primitive and $p(u)$ divides i.*

If u is a full word over a binary alphabet, then the *complement* of u, denoted $\bar{u}$, is such that $\bar{u}(i) \neq u(i)$ for all i.

Lemma 2.2. *Let w be a primitive full word over a binary alphabet. Then $\bar{w} \in [w]$ if and only if $w = u\bar{u}$ for some u.*

Proof. Suppose $w = u\bar{u}$ for some u. Observe that $\bar{w} = \bar{u}u$, and for $j = |u|$, $\sigma^j(w) = \bar{w}$. Therefore $\bar{w} \in [w]$. For the forward implication, suppose $\bar{w} \in [w]$. Since $w, \bar{w} \in [w]$ and every element in $[w]$ has the same alphabet letter distribution, there must be equal numbers of a's and b's in w for the distinct letters a, b in the alphabet. Consequently $|w|$ is even. We know that $\bar{w} = \sigma^j(w)$ for some j, $0 < j < |w|$, and so $w(i) \neq w(i+j)$ for all $i \in [0..j-1]$. Note that $\sigma^{2j}(w) = \sigma^j(\bar{w}) = w$, and by Lemma 2.1, $2j = |w|$ follows because w is primitive. As a result $j = \frac{|w|}{2}$. Therefore $w = u\bar{u}$ for some u. $\square$

We now review concepts on unavoidable sets. A *two-sided infinite word* over A is a function $w : \mathbb{Z} \to A$. A finite word u is a *factor* of w if u is a finite subsequence of w, that is, if there exists some $i \in \mathbb{Z}$ such that $u = w(i) \cdots w(i + |u| - 1)$. For a positive integer p, w has *period* p, or w is *p-periodic*, if $w(i) = w(i + p)$ for all $i \in \mathbb{Z}$. If w has period p for some p, then we call w *periodic*. If v is a nonempty finite word, then we denote by $v^{\mathbb{Z}}$ the unique two-sided infinite word w with period $|v|$ such that $v = w(0) \cdots w(|v| - 1)$.

Let x be a partial word over A, and let w be a two-sided infinite full word over A. Then w *avoids* x if there does not exist a factor u of w such that $x \subset u$. Moreover w avoids a set of partial words $X \subset A_\diamond^*$ if for every $x \in X$, w avoids x. The set X is *unavoidable* if there does not exist a two-sided infinite full word w over A such that w avoids X.

It was proved in [4] that if X is a set of partial words and Y is the resulting set from any of the so-called operations of factoring, prefix-suffix, hole truncation, and expansion, then X is avoidable if and only if Y is avoidable. For example, the hole truncation on $x\diamond^n$ preserves avoidability, that is, if $x\diamond^n \in X$ for some positive integer n, then $Y = (X\backslash\{x\diamond^n\}) \cup \{x\}$ has the same avoidability as X.

2.3 Unavoidable Sets of Partial Words of Size Two

In this section, we investigate the avoidability of the sets $X_{m_1,\ldots,m_k|n_1,\ldots,n_l}$. First, we recall a characterization for the case where $k = l = 1$.

Theorem 2.3 ([4])**.** *Write $m+1 = 2^s r_1$, $n+1 = 2^t r_2$ where r_1, r_2 are odd. Then $X_{m|n}$ is unavoidable if and only if $s \neq t$. Furthermore, if $s = t$, then $(a^{2^s} b^{2^s})^{\mathbb{Z}}$ avoids $X_{m|n}$.*

Since Theorem 2.3 classifies exactly which of the sets $X_{m|n}$ are avoidable or unavoidable, we instead focus on what types of words avoid these sets. In particular, we wish to give a lower bound on the length of a period of an avoiding word. As we show in Theorem 2.4, the avoiding word in Theorem 2.3 demonstrates the smallest period.

Theorem 2.4. *Let v be a full word over $\{a, b\}$. If $v^{\mathbb{Z}}$ avoids $X_{m|n}$, then 2^{s+1} divides $|v|$ where $m+1 = 2^s r_1$ and $n+1 = 2^s r_2$ for odd integers r_1, r_2. In addition, if v is primitive, then $v = u\bar{u}$ for some u.*

Proof. Let v be a full word over $\{a, b\}$ such that $v^{\mathbb{Z}}$ avoids $X_{m|n}$. By Theorem 2.3, $m+1 = 2^s r_1$ and $n+1 = 2^s r_2$ for odd integers r_1, r_2.

We first consider v primitive, in which case we show that $v = u\bar{u}$ for some u. Indeed, set $w = v^{\mathbb{Z}}$. We claim that $w(i) \neq w(i+m+1)$ for all i. Clearly if $w(i) = a$, then $w(i+m+1) \neq a$. If $w(i) = b$ and $w(i+m+1) = b$, then $w(i+n+1) = a$ and $w(i+m+1+n+1) = a$ to avoid $b\diamond^n b$; letting $j = i+n+1$ we get $w(j) = w(j+m+1) = a$, a contradiction. It follows by a symmetric argument that $w(i) \neq w(i+n+1)$ for all i. Observe that for some i, $w(i)\cdots w(i+|v|-1) = v$, and hence $\bar{v} = w(i+m+1)\cdots w(i+m+|v|)$. Therefore $\bar{v} \in [v]$, and so $v = u\bar{u}$ for some u follows as a direct consequence of Lemma 2.2.

Note one of the consequences of the fact that $w(i) \neq w(i+m+1)$ for all i, is that the binary word w has period $2(m+1)$ which provides the existence of an integer t such that $t|v| = 2(m+1)$ since v being primitive, $|v|$ is the smallest period of w. If t is even, then $t = 2t'$ for some t', and so $t'|v| = m+1$. Then $w(i) = w(i+|v|) = \cdots = w(i+t'|v|) = w(i+m+1)$, a contradiction. So t is odd, and since $t|v| = 2(m+1) = 2^{s+1}r_1$, we get that 2^{s+1} divides $|v|$.

The proof when v is not primitive then follows easily, since there exists a primitive factor v' such that $v = (v')^p$ for some $p > 1$. Since $v^{\mathbb{Z}}$ avoids $X_{m|n}$ by assumption, $(v')^{\mathbb{Z}}$ avoids $X_{m|n}$ and hence 2^{s+1} divides $|v'|$. Therefore 2^{s+1} divides $|v|$. □

Note that as a result, a lower bound on the length of v is 2^{s+1}. Since Theorem 2.3 provides such a v, this lower bound is optimal.

Now, we reduce Conjecture 2.9 of [4], regarding a complete characterization of when $X_{m|n_1,n_2}$ is avoidable, to Conjecture 2.10 that involves six systems of inequalities on m, n_1 and n_2. A fact from [4] that the set $X_{m_1 \dots m_k|n_1 \dots n_l}$ is unavoidable if and only if the set

$$\{a\diamond^{p(m_1+1)-1}a \cdots a\diamond^{p(m_k+1)-1}a, b\diamond^{p(n_1+1)-1}b \cdots b\diamond^{p(n_l+1)-1}b\}$$

is unavoidable, where p is a positive integer, will allow us to reduce the number of sets being considered. We may hence assume without loss of generality that $\gcd(m+1, n_1+1, n_2+1) = 1$. For $X_{m|n_1,n_2}$ to be avoidable it is sufficient that $X_{m|n_1}$, $X_{m|n_2}$ or $X_{m|n_1+n_2+1}$ be avoidable.

Here are unavoidability results for $k = 1$ and $l = 2$.

Proposition 2.5 ([4])**.** *Suppose either* $m = 2n_1+n_2+2$ *or* $m = n_2-n_1-1$, *and* n_1+1 *divides* n_2+1. *Then* $X_{m|n_1,n_2}$ *is unavoidable if and only if* $X_{m|n_1}$ *is unavoidable.*

Theorem 2.6. *Suppose either* $m = 2n_1 + n_2 + 2$ *or* $m = n_2 - n_1 - 1$. *Let* $m + 1 = 2^s r_0$ *and* $n_1 + 1 = 2^{t_1} r_1$ *for nonnegative integers* s, t_1 *and odd* r_0, r_1. *Then* $X_{m|n_1,n_2}$ *is unavoidable if and only if* $t_1 < s$.

Proof. Set $n_2 + 1 = 2^{t_2} r_2$ for some nonnegative integer t_2 and some odd integer r_2. If $t_1 < s$, then $X_{m|n_1,n_2}$ is unavoidable by Theorem 4 of [4]. So suppose $t_1 \geq s$. Then we claim that $X_{m|n_1,n_2}$ is avoidable by Theorem 2.3. Observe that if $t_1 = s$, then by Theorem 2.3, $X_{m|n_1}$ is avoidable, so $X_{m|n_1,n_2}$ is avoidable. Similarly if $t_2 = s$, then $X_{m|n_1,n_2}$ is avoidable. Assume $t_1 > s$. Note that this forces $n_1 + 1$ to be even and consequently n_1 to be odd. Suppose $m + 1$ is odd and $m = 2n_1 + n_2 + 2$. In this case $n_2 + 1$ is odd. Therefore $s = t_2 = 0$, so $X_{m|n_1,n_2}$ is avoidable. Similarly, if $m = n_2 - n_1 - 1$, then n_2 must be even since n_1 is odd by hypothesis. Thus $s = t_2 = 0$, so $X_{m|n_1,n_2}$ is avoidable. Suppose $m + 1$ is even, and consider $m = n_2 - n_1 - 1$. Then $m + 1 = (n_2 + 1) - (n_1 + 1)$, equivalently $2^s r_0 = 2^{t_2} r_2 - 2^{t_1} r_1$ or $2^s(r_0 + 2^{t_1-s} r_1) = 2^{t_2} r_2$. The latter implies $2^s = 2^{t_2}$ and thus $s = t_2$. Note that this follows because $t_1 > s$, so $r_0 + 2^{t_1 - s} r_1$ and r_2 are odd. Therefore $X_{m|n_1,n_2}$ is avoidable. The case when $m+1$ is even and $m = 2n_1 + n_2 + 2$ follows similarly. □

Corollary 2.7. *If $X_{m|n_1,n_2}$ is unavoidable by Proposition 2.5, then $X_{m|n_1,n_2}$ is unavoidable by Theorem 2.6. In other words, writing $m+1 = 2^s r_0$, $n_1 + 1 = 2^t r_1$ where r_0, r_1 are odd, if m, n_1, n_2 fulfill the conditions of Proposition 2.5 and $X_{m|n_1}$ is avoidable, then $t < s$.*

Proof. Suppose $X_{m|n_1,n_2}$ is unavoidable by Proposition 2.5. Then either $m = 2n_1+n_2+2$ or $m = n_2-n_1-1$, which fit the conditions of Theorem 2.6. In addition, $X_{m|n_1}$ is unavoidable. Let $m+1 = 2^s r_0$ and $n_1 + 1 = 2^t r_1$, where s, t are nonnegative integers and r_0, r_1 are odd. By Theorem 2.3, $s \neq t$. Referring to the proof of Theorem 2.6, $t < s$. So $X_{m|n_1,n_2}$ is unavoidable by Theorem 2.6. □

Theorem 2.8. *Let m, n_1, and n_2 be nonnegative integers such that $2m = n_1 + n_2$. Then $X_{m|n_1,n_2}$ is unavoidable if and only if $n_1 \neq n_2$ and $d = |m - n_1|$ divides $m+1$. Moreover, if $d \nmid m+1$, then $X_{m|n_1,n_2}$ is avoided by a two-sided infinite word whose period is polynomial in the size of $X_{m|n_1,n_2}$ (with the extra condition that if $n_1 \neq n_2$, then $X_{m|n_1,n_2}$ is more precisely avoided by a two-sided infinite word whose period is at most d).*

Proof. Let $x_a = a\diamond^m a$ and $x_b = b\diamond^{n_1} b \diamond^{n_2} b$. Proposition 2 of [4], which states that if $n_1 < n_2$, $2m = n_1 + n_2$ and $|m - n_1|$ divides $m+1$, then $X_{m|n_1,n_2}$ is unavoidable, shows that the conditions on m, n_1 and n_2 are sufficient. It remains to be shown that the conditions are necessary.

Assume $n_1 = n_2$. Then $m = n_1 = n_2$. Consequently by Theorem 2.3, $X_{m|n_1}$ is avoidable, so $X_{m|n_1,n_2}$ is avoidable. In fact, by Theorem 2.3, $(a^{2^s} b^{2^s})^{\mathbb{Z}}$ avoids $X_{m|n_1,n_2}$, where $m+1 = 2^s r$ for some odd integer r. Note that the period 2^{s+1} is polynomial in the size of $X_{m|n_1,n_2}$.

Assume $n_1 \neq n_2$. We show that if $d \nmid m+1$, then $X_{m|n_1,n_2}$ is avoided by the two-sided infinite word w, constructed by the following algorithm on d, m and n_1, with w's period at most d. The algorithm initializes w with $\diamond^{\mathbb{Z}}$.

For $0 \leq i \leq d-1$ *do: If* $w(i)$ *is not defined, then define*

$$w(j) = \begin{cases} a, & \textit{if } j \equiv i \bmod d; \\ b, & \textit{if } j \equiv i \pm (m+1) \bmod d. \end{cases} \tag{2.2}$$

In order to prove the correctness of the algorithm, we need a means of speaking about whether w avoids $X_{m|n_1,n_2}$ even if w is not fully defined.

We introduce the concept of *weakly avoidable set* and say w *weakly avoids* $X_{m|n_1,n_2}$ if there does not exist a factor u of w such that $x \subset u$ for every $x \in X_{m|n_1,n_2}$. Note that this is simply an extension of the definition of avoiding full words to avoiding partial words.

We prove the algorithm by induction, showing that the following invariant holds: w weakly avoids $X_{m|n_1,n_2}$ and w is consistent (that is, it is not the case that $w(j) = a$ and $w(j) = b$ for the same integer j). For the base case, observe that $w = \diamond^{\mathbb{Z}}$, so w weakly avoids $X_{m|n_1,n_2}$ trivially. Also because w is completely undefined, there does not exist an integer j such that $w(j) = a$ and $w(j) = b$, so w is consistent. This concludes the base case.

Now suppose that w is at some intermediate state of the algorithm and that our invariant holds. We show that at the end of the loop, the invariant still holds. Let $i \in [0..d-1]$ be the integer selected by the algorithm at this step. Note a condition on i is that $w(i)$ is not defined. We show that Equation (2.2) does not alter the invariant. Suppose for sake of contradiction that the invariant is altered. Then either w no longer avoids $X_{m|n_1,n_2}$, or w is inconsistent.

Suppose that w is not consistent. Then for some integer j, $w(j) = a$ and $w(j) = b$. By Equation (2.2), there exist integers $i_0, i_1 \in [0..d-1]$ such that both $j \equiv i_0 \bmod d$ and $j \equiv i_1 \pm (m+1) \bmod d$. It follows then that

$$i_0 \equiv i_1 \pm (m+1) \bmod d \tag{2.3}$$

If $i_0 = i_1$, then $d \mid (m+1)$, which contradicts our original hypothesis. Suppose $i \neq i_0$ and $i \neq i_1$. This inconsistency was present in w before this step, contradicting our assumption that the invariant was valid. Thus $i_0 \neq i_1$ and either $i = i_0$ or $i = i_1$. If $i = i_0$, then by Equation (2.3), $w(i) = b$ was defined previously by i_1 according to Equation (2.2). Symmetrically, if $i = i_1$, then $w(i) = b$ was defined previously by i_0 via Equation (2.2). Therefore this step could not have introduced an inconsistency. Note that this was proved independent of the question of whether w weakly avoids $X_{m|n_1,n_2}$, and as such we will use this result in its proof.

Suppose w no longer weakly avoids $X_{m|n_1,n_2}$. Suppose for contradiction that w no longer weakly avoids x_a. Then there must exist some factor of w of the form $u = w(j) \cdots w(j+m+1)$ for some integer j, such that $u(0) = a$ and $u(m+1) = a$. Since Equation (2.2) defines the construction of w, it follows that for some integers $i_0, i_1 \in [0..d-1]$ with $w(i_0) = w(i_1) = a$, both $j \equiv i_0 \bmod d$ and $j + m + 1 \equiv i_1 \bmod d$. It then follows that $i_0 \equiv i_1 - (m+1) \bmod d$, which by Equation (2.2) means $w(j) = a$ and $w(j) = b$.

Thus an inconsistency was either introduced at this step or prior, which in either case is a contradiction. Therefore our assumption that w no longer weakly avoids x_a was incorrect.

Suppose for contradiction that w no longer weakly avoids x_b (the case where w no longer weakly avoids x_a is simpler). Then there must exist some factor of w of the form $u = w(j) \cdots w(j + 2m + 2)$ for some integer j, such that $u(0) = b$, $u(n_1 + 1) = b$, and $u(2(m + 1)) = b$. Since Equation (2.2) defines how w is constructed, it follows that for some integers $i_0, i_1, i_2 \in [0..d - 1]$ (not necessarily distinct) where $w(i_k) = a$ for $k \in [0..2]$, an equation defining j_0, j_1, j_2 from each column holds true:

$\boldsymbol{j \equiv j_0 \bmod d}$	$\boldsymbol{j + m + 1 \equiv j_1 \bmod d}$	$\boldsymbol{j + 2(m + 1) \equiv j_2 \bmod d}$
$j_0 = i_0 + (m + 1)$	$j_1 = i_1 + (m + 1)$	$j_2 = i_2 + (m + 1)$
$j_0 = i_0 - (m + 1)$	$j_1 = i_1 - (m + 1)$	$j_2 = i_2 - (m + 1)$

Note that the second equation is $j + m + 1 \equiv j_1 \bmod d$ instead of $j + n_1 + 1 \equiv j_1 \bmod d$. This is because $m + 1 - (m - n_1) = n_1 + 1$, and $d = |m - n_1|$, hence mod d the two equations are equivalent. It then follows that $j_1 = i_1 - (m + 1)$, for otherwise $j + m + 1 \equiv i_1 + m + 1 \bmod d$ that is $j \equiv i_1 \bmod d$, and $i_0 \pm (m + 1) \equiv i_1 \bmod d$, where the latter follows from the first column in the table. Thus by Equation (2.2) $w(i_1) = b$, but since we assumed $w(i_1) = a$, we have reached a state of inconsistency, which is a contradiction. Hence $j + m + 1 \equiv i_1 - (m + 1) \bmod d$, or equivalently, $j \equiv i_1 - 2(m+1) \bmod d$. Combining this with Column 3 from the table, we get $(i_1-2(m+1))+2(m+1) \equiv i_2 \pm (m+1) \bmod d$ and $i_1 \equiv i_2 \pm (m+1) \bmod d$. But by Equation (2.2), this implies $w(i_1) = b$. Yet we previously assumed $w(i_1) = a$, hence we have reached a state of inconsistency, which is a contradiction. Consequently Column 2 cannot be satisfied, so there cannot exist a factor u of w such that $u(0) = u(n_1+1) = u(2(m+1)) = b$, meaning our assumption that w no longer weakly avoids x_b was incorrect.

Therefore the invariant holds for the duration of the algorithm. Observe that the algorithm will eventually halt, for at each step an $i \in [0..d - 1]$ is chosen such that $w(i)$ is undefined and defines $w(i)$. Thus upon termination, w is defined for all $i \in [0..d-1]$, and as a result of Equation (2.2), w is fully defined. Furthermore by the invariant, w is consistent and weakly avoids $X_{m|n_1,n_2}$. Therefore we have constructed the word $w = (w(0) \cdots w(d-1))^{\mathbb{Z}}$ which avoids $X_{m|n_1,n_2}$. We conclude that $X_{m|n_1,n_2}$ is avoided by a two-sided infinite word whose period is polynomial in the size of $X_{m|n_1,n_2}$, or more precisely it is avoided by a two-sided infinite word whose period is at most $d = |m - n_1|$. □

It is thought that the following conjecture gives a complete characterization of when $X_{m|n_1,n_2}$ is unavoidable.

Conjecture 2.9 ([4])**.** *Let m, n_1, n_2 be nonnegative integers such that $n_1 \leq n_2$ and $\gcd(m+1, n_1+1, n_2+1) = 1$. Then $X_{m|n_1,n_2}$ is unavoidable if and only if one of the following conditions hold:*

- *The case where $X_{m|n_1}$ is unavoidable, $m = 2n_1+n_2+2$ or $m = n_2-n_1-1$, and n_1+1 divides n_2+1.*
- *The case where $m = 2n_1+n_2+2$ or $m = n_2-n_1-1$, and the highest power of 2 dividing n_1+1 is less than the highest power of 2 dividing $m+1$.*
- *The case where $n_1 < n_2$, $2m = n_1+n_2$ and $m-n_1$ divides $m+1$.*
- *The case where $(m, n_1, n_2) = (6, 1, 3)$.*

Using Theorems 2.6 and 2.8 as well as Corollary 2.7, Conjecture 2.9 can be equivalently stated as follows: "Let m, n_1, n_2 be nonnegative integers such that $n_1 \leq n_2$ and $\gcd(m+1, n_1+1, n_2+1) = 1$. Then $X_{m|n_1,n_2}$ is unavoidable if and only if $(m, n_1, n_2) = (6, 1, 3)$ or $X_{m|n_1,n_2}$ is unavoidable by Theorem 2.6 or Theorem 2.8."

Conjecture 2.10. *Let m, n_1, n_2 be nonnegative integers such that $n_1 \leq n_2$, $\gcd(m+1, n_1+1, n_2+1) = 1$, and $(m, n_1, n_2) \neq (6, 1, 3)$. Then $X_{m|n_1,n_2}$ is avoidable if one of (2.4)–(2.9) is satisfied:*

$$m < n_2 - n_1 - 1 \text{ and } 2m < n_1 + n_2 \tag{2.4}$$

$$m < n_2 - n_1 - 1 \text{ and } 2m > n_1 + n_2 \tag{2.5}$$

$$m > n_2 - n_1 - 1 \text{ and } 2m < n_1 + n_2 \tag{2.6}$$

$$m < 2n_1 + n_2 + 2 \text{ and } m > n_2 - n_1 - 1 \text{ and } 2m > n_1 + n_2 \tag{2.7}$$

$$m > 2n_1 + n_2 + 2 \text{ and } m < 2n_2 + n_1 + 2 \tag{2.8}$$

$$m > 2n_2 + n_1 + 2 \tag{2.9}$$

"Conjecture 2.9 is true" is equivalent to "Conjecture 2.10 is true." Indeed, by Theorems 2.6 and 2.8, the sets where $2m = n_1+n_2$, $m = n_2-n_1-1$, $m = 2n_1+n_2+2$ and $m = 2n_2+n_1+2$ are now classified. It is easy to verify that the only sets which remain to be classified are those determined by the six systems of inequalities (2.4)–(2.9), and by Conjecture 2.9 these sets must be avoidable. In order to classify the remaining sets, we will introduce in Section 2.4 the canonical forms which will be useful for solving the conjecture.

2.4 Canonical Forms

In an effort to prove Conjecture 2.10, we attempt to classify exactly which sets have particular patterns of avoiding words. In order to motivate our technique, we consider the following proposition which is useful for verifying if a word is an avoiding word.

Proposition 2.11. *Let X be a set of partial words, and let v be a primitive full word. Let $M = \max\{|x| \mid x \in X\}$, and let $k = \frac{M+|v|-1}{|v|}$. Then $v^{\mathbb{Z}}$ avoids X if and only if v^k avoids X.*

Proof. To see the correctness of this result, it is first easier to consider how one might go about showing a two-sided infinite word $w = v^{\mathbb{Z}}$ avoids a word in X. One could begin by aligning x with $w(0)\cdots w(|x|-1)$, and then with $w(1)\cdots w(|x|)$ and continuing in this manner. If it were ever the case that x aligned with a factor of w, say $w(n)\cdots w(|x|-1+n)$, and $x \subset w(n)\cdots w(|x|-1+n)$, then we can conclude w does not avoid x. But in order to claim w does avoid x, we need to determine a point at which we can stop these comparisons. In fact, we can stop as soon as we have checked x against all factors from $w(0)\cdots w(|x|-1)$ up to $w(|v|-1)\cdots w(|x|-1+|v|-1)$, since $w(|v|)\cdots w(|x|-1+|v|) = w(0)\cdots w(|x|-1)$ because w is $|v|$-periodic. Therefore we need only consider $|x|+|v|-1$ letters of w for any particular x. If we consider x where $|x| = M$, the maximal length of all words in X, then we need $M + |v| - 1$ letters of w. This is achieved by letting $k = \frac{M+|v|-1}{|v|}$ and considering v^k. □

As a result of this proposition, for sets with words significantly longer than the period of an avoiding word, a large number of repetitions must be considered in order to verify that the word avoids the set. However, if we could somehow reduce the words in the set such that they are not longer than the period of the avoiding word, then we would need to consider at most two repetitions of the avoiding word (rounded up for simplicity). In fact, such a reduction exists, and so we begin by defining a new concept which will allow us to restrict our attention to these particular types of sets.

The *n-partition* of a partial word u is the set

$$\{u[0..n-1], u[n..2n-1], \ldots, u[(k-1)n..kn-1], u[kn..|u|-1]\diamond^m\}$$

where $u[i..j]$ denotes $u(i)\cdots u(j)$, and where the nonnegative integers k and m are chosen such that $|u| - kn + m = n$, where $0 \leq m < n$. Informally, we are partitioning u into k partial words of length n, where the kth slice may need to be padded with holes to meet the length requirements. If u is n-periodic, then we define the *n-canonical form* of u, denoted by $c_n(u)$, to be $\vee_{x \in P} x$ where P is the n-partition of u and where $\diamond$'s from the left and right ends of $\vee_{x \in P} x$ are truncated (note that $|c_n(u)| \leq n$). The *n-canonical form* of a set of partial words X is $c_n(X) = \{c_n(x) \mid x \in X \text{ and } x \text{ is } n\text{-periodic}\}$.

As an example, let us compute $c_{39}(X_{6|11,32})$. The 39-*partition* of $u = b\diamond^{11}b\diamond^{32}b$ is the set

$$P = \{u[0..38], u[39..45]\diamond^{32}\} = \{b\diamond^{11}b\diamond^{26}, \diamond^{6}b\diamond^{32}\}$$

and $c_{39}(u) = b\diamond^{5}b\diamond^{5}b$. Thus

$$c_{39}(X_{6|11,32}) = \{c_{39}(a\diamond^{6}a), c_{39}(b\diamond^{11}b\diamond^{32}b)\} = \{a\diamond^{6}a, b\diamond^{5}b\diamond^{5}b\}$$

Lemma 2.12. *Let v be a full word and X be a set of partial words such that all elements in X are $|v|$-periodic. Then the two-sided infinite word $v^{\mathbb{Z}}$ avoids X if and only if $v^{\mathbb{Z}}$ avoids $c_{|v|}(X)$.*

The following results are defined over n-canonical sets for some positive integer n. Recall from Proposition 2.11 that for an avoiding word $v^{\mathbb{Z}}$ where $|v| = n$, we need only check an n-canonical set against at most two repetitions of v.

Before we continue, we give a high-level overview of the method which we employ for our results. First, we define some integer n, and let $X_{m|n_1,n_2} = \{x_a, x_b\}$ be the n-canonical form of some set. At the same time, we define some full word v in terms of some pattern, such as $v = a^p b^q$, where $|v| = n$. We then give an exact characterization of the possible values for p and q in relationship to m, n_1, and n_2. From the perspective of an implementation, one would see if the theorem can be satisfied for n from 1, 2, up to $M = \max\{|x_a|, |x_b|\}$. For $|v| > M$, note that m, n_1, n_2 are constant; that is, $c_n(X_{m|n_1,n_2}) = X_{m|n_1,n_2}$. If m, n_1, n_2 satisfy the constraints, then we need only find p and q which also satisfy the constraints.

Theorem 2.13. *Let $v = a^p b^q$ where $p > 0, q > 0$, and let $X_{m_1,\ldots,m_k|n_1,\ldots,n_l}$ where $k, l > 0$ be $|v|$-canonical. Set $x_a = a\diamond^{m_1}a \cdots \diamond^{m_k}a$ and $x_b = b\diamond^{n_1}b \cdots b\diamond^{n_l}b$. Then $v^{\mathbb{Z}}$ avoids $\{x_a, x_b\}$ if and only if the following conditions hold: (i) $|x_a| > p$, (ii) $m_i < q$ for all $i \in [1..k]$, (iii) $|x_b| > q$, and (iv) $n_i < p$ for all $i \in [1..l]$.*

Proof. We begin with the reverse implication. Assume the conditions hold and set $w = v^{\mathbb{Z}}$. Since x_a is $|v|$-canonical, $|x_a| \leq |v|$. Then by Condition (i), $p < |x_a|$, so v (the finite word) avoids x_a. Thus in order for w to not avoid x_a, every b in v must be matched up with a hole in x_a, which would mean $m_i \geq q$ for some $i \in [1..k]$. But this cannot happen by Condition (ii), and hence w avoids x_a. The proof for x_b is symmetric, and therefore w avoids $\{x_a, x_b\}$.

Now suppose at least one of the conditions does not hold. We consider Conditions (i) and (ii) only due to symmetry. If (i) does not hold, then v does not avoid x_a, and hence $v^{\mathbb{Z}}$ does not avoid x_a. If (ii) does not hold, then because $|x_a| \leq |v|$, for some $i \in [1..k]$ where $m_i \geq q$, straddling the b's in v with this m_i shows that $v^{\mathbb{Z}}$ does not avoid x_a. This can be seen more clearly in the table

a	$\cdots$	a	b	$\cdots$	b	a	$\cdots$	a	$b \cdots$	b
	$a\diamond^{m_1} \cdots \diamond^{m_{i-1}}$	a	$\diamond$	$\diamond^{q-2}$	$\diamond$	a	$\diamond^{m_{i+1}} \cdots \diamond^{m_k}\ a$			

that depicts the situation where $m_i = q$ for some $i \in [1..k]$. Because $|x_a| \leq |v|$, it is never the case that x_a must straddle more than one set of b's. Therefore if at least one condition does not hold, then $\{x_a, x_b\}$ is not avoided by $v^{\mathbb{Z}}$. □

Corollary 2.14. *If* $\max\{n_1, n_2\} < m < n_1 + n_2 + 2$*, then the conditions of Theorem 2.13 hold for* $c_{2m+1}(X_{m|n_1,n_2})$.

Proof. Let $v = a^m b^{m+1}$ and note that $|v| = 2m + 1$. Suppose $\max\{n_1, n_2\} < m < n_1 + n_2 + 2$. Then the word $w = v^{\mathbb{Z}}$ avoids $X_{m|n_1,n_2}$ (see Proposition 5 of [4]). Here $\max\{n_1, n_2\} < m < n_1 + n_2 + 2 \leq 2m + 1$. Therefore by Lemma 2.12, w avoids $c_{|v|}(X_{m|n_1,n_2})$. Since v is of the form required by Theorem 2.13, the conditions hold for $c_{|v|}(X_{m|n_1,n_2})$. □

Similarly to Theorem 2.13, we can prove the following.

Theorem 2.15. *Let* $v = a^{p_1} b^{q_1} \cdots a^{p_M} b^{q_M}$ *where* $p_j > 0, q_j > 0$ *for all* $j \in [1..M]$*. Let* $X_{m_1,\ldots,m_k|n_1,\ldots,n_l}$ *where* $k, l > 0$ *be* $|v|$*-canonical. Set* $x_a = a\diamond^{m_1} a \cdots \diamond^{m_k} a$ *and* $x_b = b\diamond^{n_1} b \cdots b\diamond^{n_l} b$*. Then* $v^{\mathbb{Z}}$ *avoids* $\{x_a, x_b\}$ *if the following conditions hold: (i)* $|x_a| > p_j$ *for all* $j \in [1..M]$*, (ii)* $m_i < q_j$ *for all* $i \in [1..k]$ *and all* $j \in [1..M]$*, (iii)* $|x_b| > q_j$ *for all* $j \in [1..M]$*, and (iv)* $n_i < p_j$ *for all* $i \in [1..l]$ *and all* $j \in [1..M]$.

Theorem 2.16. *Let* $v = (ab)^p b^q$ *where* $p > 0$ *and* $q > 0$*, and let* $X_{m|n_1,n_2}$ *be* $|v|$*-canonical. Set* $x_a = a\diamond^m a$ *and* $x_b = b\diamond^{n_1} b\diamond^{n_2} b$*. Then* $v^{\mathbb{Z}}$ *avoids* $\{x_a, x_b\}$ *if and only if* q *is odd and* m, n_1, n_2 *each fall into one of the following cases: (1)* m *is even and* $m < q$*, (2)* m *is odd and* $2(p-1) < m$*, and (3)* n_1, n_2 *are even,* $n_1, n_2 \geq q - 1$ *and* $n_1 + n_2 + 3 \leq |v| - (q - 2)$.

Proof. Set $w = v^{\mathbb{Z}}$, and assume that $v^{\mathbb{Z}}$ avoids $\{x_a, x_b\}$. We will show that the requirement q is odd is necessary by x_b, and so for now we assume q to be odd. Consider x_a. If m is even and q is odd, then $|v|$ is odd. Observe that x_a successfully straddles the b^{q+1} in w if and only if $m > q$. Since $|v|$ is odd, the last a in x_a lines up with an a after this last substring of b's, and hence w does not avoid x_a. If m is odd, then x_a is not avoided by w when $m \leq 2(p-1)$ due to the substring $(ab)^p$ in v. Observe that x_a can straddle b^{q+1} in w and match an a following that if and only if q is even and $m > q$; but by assumption q is odd, so this is a non-issue.

Now we consider x_b, and make no assumption regarding the parity of q. We claim that it is necessary that $n_1, n_2 \geq q-1$. Suppose for contradiction that $n_2 < q-1$; then $b\diamond^{n_2}b$ can line up with b^{q+1} in v in two distinct ways, and in at least one of these, the remaining b in x_b will line up with another b in v. The same argument holds for n_1, so $n_1, n_2 \geq q-1$.

We now show that neither n_1 nor n_2 can be odd. Suppose for contradiction both n_1, n_2 are odd. Then x_b is contained in the factor of w which begins at the last b in v. Now suppose n_1 is even and n_2 is odd. Then x_b is contained in the factor of w which begins at the second to last b in v. Finally suppose n_1 is odd and n_2 is even. As a result, x_b is contained in the factor of w where the last letter is the second b in b^{q+1}.

Now suppose n_1, n_2 are even and q is even. Then x_b is contained in the factor of w where the middle b in x_b lines up with the second b in b^{q+1}. This is because there are an odd number of b's on either side of this b, hence the even n_1, n_2 will allow the remaining b's in x_b to match b's in the factor. Observe that for n_1, n_2 even and q odd, in order for x_b to not avoid w, two b's must match in $b^q a$. We outline the last case when this can occur and x_b does not avoid w. Suppose $n_1 + n_2 + 3 > |v| - (q-2)$. Then x_b is contained in the factor of w beginning at the second to last b in v. The following table illustrates each of the cases outlined where w does not avoid x_b.

a	b	a	b	a	b	a	b	a	b	b	b	b	b	b	b	b
$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	b					b	$\diamond$	$\diamond$	$\diamond$	$\diamond$	b	$\diamond$
$\diamond$	b	$\diamond$	$\diamond$	$\diamond$	b											b
$\diamond$	$\diamond$	$\diamond$	b	$\diamond$	b										b	$\diamond$
	b	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	b	$\diamond$	$\diamond$	b						
$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	b	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	b			b	$\diamond$

Here $v = (ab)^5 b^7$, so $|v| = 17$. The first row of the table demonstrates when $n_1 = 4 < q-1 = 6$. The next three rows demonstrate that neither

n_1 and n_2 can be odd. In this example, n_1, n_2 are 1, 3; 4, 1; and 5, 2 respectively. The last row illustrates when $n_1 = n_2 = 6$, but $n_1 + n_2 + 3 = 15 > |v| - (q-2) = 12$. Note that in this example, $n_1 = 6 \geq q - 1 = 6$.

Finally, we show that if n_1, n_2 are even, $n_1, n_2 \geq q-1$ and $n_1+n_2+3 \leq |v| - (q-2)$, then x_b is avoided by w. Suppose that w does not avoid x_b. Since q is odd, the only way this can occur is if at least two b's in x_b line up with some b's in b^{q+1}. However, this can only occur by the two b's lining up with the first and last b in b^{q+1} because $n_1, n_2 \geq (q-1)$ and $n_1 + n_2 + 3 \leq |v| - (q-2)$. In this case, the third b necessarily lines up with an a because n_1, n_2 are even, and therefore w avoids x_b. Therefore if $v = (ab)^p b^q$, then $v^{\mathbb{Z}}$ avoids the $|v|$-canonical set $X_{m|n_1,n_2}$ if and only if q is odd and m, n_1, n_2 meet the necessary conditions. □

2.5 The Answer to the Conjectures

In this section, we prove that Conjectures 2.9 and 2.10 are true.

Let us define $Y_{m|n_1,\ldots,n_l} = \{a\diamond^m a, b\diamond^m b, b\diamond^{n_1} b \cdots b\diamond^{n_l} b, a\diamond^{n_1} a \cdots a\diamond^{n_l} a\}$. Then it is obvious that if $Y_{m|n_1,n_2}$ is avoidable then $X_{m|n_1,n_2}$ is avoidable, so it seems possible that studying $Y_{m|n_1,n_2}$ will help solve the conjectures.

In order to do this, we will first prove a theorem (Theorem 2.40 below) which will give us a tool to investigate Conjecture 2.10, and indeed we prove this theorem in the following family of results and proofs.

Lemma 2.17. *If the two-sided infinite word w avoids $Y_{m|n_1,\ldots,n_l}$, then there exists a word u such that $|u| = m+1$ and $w = (u\overline{u})^{\mathbb{Z}}$.*

Proof. Define $u(i) = w(i)$ for $0 \leq i < m+1$. Then note that if $m+1 \leq i < 2m+2$, then $w(i) = \overline{w(i-(m+1))}$, since w avoids $a\diamond^m a$ and $b\diamond^m b$. Therefore, if we define $u' = u\overline{u}$, then $w(i) = u'(i)$ for $0 \leq i < 2m+2$. Since w is $(2m+2)$-periodic, it follows that $w = (u')^{\mathbb{Z}} = (u\overline{u})^{\mathbb{Z}}$. □

Before continuing, we need to introduce a few tools that are helpful in studying $Y_{m|n_1,n_2}$.

A quick note: By $a \bmod b$, we mean the least nonnegative integer c such that $c \equiv a \pmod{b}$. For example, $11 \bmod 7$ is 4. We will often abbreviate $a \bmod (2m+2)$ by $a \bmod 2m+2$, or simply by a, when the context is clear. Similarly, we will often abbreviate $c \equiv a \pmod{(2m+2)}$ by $c \equiv a \bmod 2m+2$.

Definition 2.18. Let us denote $Z_{2m+2} = \mathbb{Z}/(2m+2)\mathbb{Z}$. If $a_1, \ldots, a_k \in Z_{2m+2}$, let $\langle a_1, \ldots, a_k \rangle$ be the subgroup of Z_{2m+2} generated by $a_1, \ldots, a_k$.

Then define the directed graph $G_{m|n_1,\ldots,n_l} = (V, E)$, where $V = \{0, \ldots, 2m+1\}$ and E is the set of edges $(u, v), u \neq v$, such that

$$u - v \equiv (\sum_{i=1}^{j} n_i) + j \bmod 2m+2 \quad \textit{for} \quad \textit{some} \quad j,\ 0 < j \leq l$$

Let $H_{m|n_1,\ldots,n_l} = (V_H, E_H)$ denote the connected component of $G_{m|n_1,\ldots,n_l}$ containing 0 as a vertex.

The following proposition follows from straightforward algebraic arguments.

Proposition 2.19. *The vertex set of $H_{m|n_1,\ldots,n_l}$ is equal to $\langle n_1+1, \ldots, n_1+n_2+\cdots+n_l+l\rangle$.*

Definition 2.20. Let $K = (V_K, E_K)$ be any subgraph of $G_{m|n_1,\ldots,n_l}$ such that, if u is a vertex in K and (u, v) is an edge in $G_{m|n_1,\ldots,n_l}$, then v is a vertex in K. Then if $c : V_K \to \{a, b\}$ is a coloring of the vertices of K, we say that c avoids $Y_{m|n_1,\ldots,n_l}$ if the following two conditions hold:

(1) If i and $i+m+1$ are vertices in K, then $c(i) \neq c(i+m+1)$.
(2) For every vertex u of K, there is an edge (u, v) of K so that $c(u) \neq c(v)$.

Furthermore, if $z \in Z_{2m+2}$, define the graph $z + K = (\{z + v \bmod 2m+2 \mid v \in V_K\}, \{(z+u \bmod 2m+2, z+v \bmod 2m+2) \mid (u, v) \in E_K\})$.

Proposition 2.21. *If K is a subgraph of $G_{m|n_1,\ldots,n_l}$, so is $z + K$. Furthermore, if K is a subgraph of $H_{m|n_1,\ldots,n_l}$ and z is a vertex of $H_{m|n_1,\ldots,n_l}$, then $z + K$ is a subgraph of $H_{m|n_1,\ldots,n_l}$.*

Lemma 2.22. *The set $Y_{m|n_1,\ldots,n_l}$ is avoidable if and only if there is a coloring of $G_{m|n_1,\ldots,n_l}$ that avoids $Y_{m|n_1,\ldots,n_l}$.*

Proof. Let $Y = Y_{m|n_1,\ldots,n_l}$ and $G = G_{m|n_1,\ldots,n_l}$. First, assume that there is an infinite two-sided word, w, that avoids Y. Then we know by Lemma 2.17 that there exists a word u so that $|u| = m+1$, and that $w = (u\overline{u})^{\mathbb{Z}}$. For i, $0 \leq i < 2m+2$, define $c(i) = w(i)$. Since $w(i+m+1) = \overline{w(i)}$ and w has a period of $2m+2$, it follows that $c(i) = \overline{c(i+m+1 \bmod 2m+2)} \neq c(i+m+1 \bmod 2m+2)$. Furthermore, assume there is an i, $0 \leq i < 2m+2$, so that $c(i) = c(i+n_1+1 \bmod 2m+2) = \cdots = c(i+n_1+\cdots+n_l+l \bmod 2m+2)$. Since w is $(2m+2)$-periodic, $w(i) = w(i+n_1+1) = \cdots = w(i+n_1+\cdots+n_l+l)$. This implies that

w does not avoid $b\diamond^{n_1}b\cdots b\diamond^{n_l}b$ or w does not avoid $a\diamond^{n_1}a\cdots a\diamond^{n_l}a$, a contradiction. Therefore there is no such i, so c is a coloring of G that avoids Y.

On the other hand, assume that c is a coloring of G that avoids Y. For i, $0 \leq i < 2m+2$, define $v(i) = c(i)$, and define $w = v^{\mathbb{Z}}$. Note that w is $(2m+2)$-periodic. Assume w does not avoid $a\diamond^m a$ or $b\diamond^m b$. Then there exists an i so that $w(i) = w(i+m+1)$, which implies $w(i) = w(i+m+1 \bmod 2m+2)$, so $c(i) = c(i+m+1 \bmod 2m+2)$. The last equality contradicts the claim that c is a coloring that avoids Y. Similarly if w does not avoid $b\diamond^{n_1}b\cdots b\diamond^{n_l}b$ or w does not avoid $a\diamond^{n_1}a\cdots a\diamond^{n_l}a$, then there exists i so that $w(i) = w(i+n_1+1) = \cdots = w(i+n_1+\cdots+n_l+l)$, which implies $c(i) = c(i+n_1+1 \bmod 2m+2) = \cdots = c(i+n_1+\cdots+n_l+l \bmod 2m+2)$. This is a contradiction, so w avoids Y. This proves the claim. $\square$

Proposition 2.23. *Let i be a vertex of $H_{m|n_1,\ldots,n_l}$. Then $i+m+1$ is a vertex of $H_{m|n_1,\ldots,n_l}$ if and only if $m+1$ is a vertex of $H_{m|n_1,\ldots,n_l}$.*

Lemma 2.24. *There is a coloring of $G_{m|n_1,\ldots,n_l}$ that avoids $Y_{m|n_1,\ldots,n_l}$ if and only if there is a coloring of $H_{m|n_1,\ldots,n_l}$ that avoids $Y_{m|n_1,\ldots,n_l}$.*

Proof. Let $H = H_{m|n_1,\ldots,n_l}$, $Y = Y_{m|n_1,\ldots,n_l}$ and $G = G_{m|n_1,\ldots,n_l}$. Note that if u is a vertex of H and (u,v) is an edge of G, then by definition v is a vertex in H. Therefore it makes sense to talk about colorings that avoid Y. First assume that c is a coloring of G that avoids Y. By restricting c to the set of vertices in H, we get a coloring of H that avoids Y.

On the other hand, assume that c is a coloring of H that avoids Y. Then let $H = H_0, H_1, \ldots, H_r$ be the connected components of G. Furthermore, let $a_0, a_1, \ldots, a_r$ be vertices of G, so that a_i is a vertex of H_i.

First, assume that $m+1$ is a vertex of H_0. Then given a vertex j of H_i, define $c'(j) = c(j - a_i \bmod 2m+2)$ (this is well defined, since $-a_i + H_i = H_0$). Note that if $c'(j) = c'(j+n_1+1 \bmod 2m+2) = \cdots = c'(j+n_1+\cdots+n_l+l \bmod 2m+2)$, it follows that $c(j-a_i \bmod 2m+2) = c(j-a_i+n_1+1 \bmod 2m+2) = \cdots = c(j-a_i+n_1+\cdots+n_l+l \bmod 2m+2)$, contradicting the fact that c is a coloring of H that avoids Y. Similarly, if $c'(j) = c'(j+m+1)$, then $c(j-a_i) = c(j-a_i+m+1)$, a contradiction. Therefore c' is a coloring of G that avoids Y.

Now, assume that $m+1$ is not a vertex of H_0. Define $(H_i)^{-1}$ to be the component of G containing $m+1+a_i$, then note that $(H_i)^{-1} \neq H_i$ and $((H_i)^{-1})^{-1} = H_i$, so $r+1$ is even and we can rearrange $H_0, \ldots, H_r$ as

$H_{i_0}, H_{k_0}, \ldots, H_{i_p}, H_{k_p}$ (where $p = \frac{r+1}{2} - 1$) so that $(H_{i_h})^{-1} = H_{k_h}$. Then let us define c' as follows. If $j \in H_{i_h}$ for some h, let $c'(j) = c(j - a_{i_h} \bmod 2m+2)$, and if $j \in H_{k_h}$ let $c'(j) = c'(j - (m+1) \bmod 2m+2)$. Then if $v \in H_{i_h}$ there exists a vertex u in H_0 so that $c(v - a_{i_h} \bmod 2m+2) \neq c(u)$, and so $c'(v) \neq c'(u + a_{i_h} \bmod 2m+2)$ and $(v, u + a_{i_h} \bmod 2m+2)$ is an edge of H_{i_h}. It follows from this that if $v \in H_{k_h}$, there exists a vertex u in H_{i_h} so that $c'(v - m - 1 \bmod 2m+2) \neq c'(u)$, so $c'(v) \neq c'(u + m + 1 \bmod 2m+2)$, where $(v, u + m + 1 \bmod 2m+2)$ is an edge of H_{k_h}. Furthermore, it follows by construction that $c'(j) \neq c'(j + m + 1)$. Thus c' is a coloring of G that avoids Y, proving the claim. □

Note that $H_{m|n_1,\ldots,n_l}$ is, from an algebraic point of view, the Cayley graph of $\langle n_1 + 1, \ldots, n_1 + \cdots + n_l + l\rangle$. Most of our arguments will take advantage of this fact. Stating our arguments in terms of graphs as opposed to groups provides simplification of some proofs, as well as a more intuitive way of looking at the results. For more information on Cayley graphs, we refer the reader to [12].

Proposition 2.25. *If* $2m \equiv n_1 + n_2 \bmod 2m+2$*, then* $Y_{m|n_1,n_2}$ *is avoidable if and only if* $X_{m|n_1}$ *is avoidable.*

Proof. First we want to show that w avoids $X_{m|n_1}$ if and only if it avoids $Y_{m|n_1}$. To see this, note that since $X_{m|n_1} \subset Y_{m|n_1}$, if w avoids $Y_{m|n_1}$ it avoids $X_{m|n_1}$. Conversely, if w avoids $X_{m|n_1}$ and does not avoid $Y_{m|n_1}$, then either it does not avoid $b\diamond^m b$ or $a\diamond^{n_1} a$. If w does not avoid $b\diamond^m b$, we can assume $w(0) = w(m+1) = b$. Since w avoids $b\diamond^{n_1} b$, it follows $w(n_1 + 1) = w(n_1 + 1 + m + 1) = a$. This, however, contradicts the claim that w avoids $a\diamond^m a$. Similarly, we can argue that if w avoids $X_{m|n_1}$, then w avoids $a\diamond^{n_1} a$. Therefore, if w avoids $X_{m|n_1}$, it avoids $Y_{m|n_1}$.

Next we want to show $G_{m|n_1,n_2} = G_{m|n_1}$. First, note that $G_{m|n_1,n_2}$ has the same vertex set of $G_{m|n_1}$. Note that $n_1 + n_2 + 2 \equiv 2m + 2 \equiv 0 \bmod 2m+2$. Furthermore, (u, v) is an edge of $G_{m|n_1,n_2}$ if and only if it is one of $G_{m|n_1}$ (since if $u - v \equiv n_1 + n_2 + 2 \bmod 2m+2$, it follows that $u \equiv v \bmod 2m+2$, so (u, v) is not an edge, and (u, v) is an edge if and only if $u - v \equiv n_1 + 1 \bmod 2m+2$ and $u \neq v$). Therefore there is a coloring of $G_{m|n_1,n_2}$ that avoids $Y_{m|n_1,n_2}$ if and only if there is a coloring of $G_{m|n_1}$ that avoids $Y_{m|n_1}$. Therefore, by Lemma 2.22, $Y_{m|n_1,n_2}$ is avoidable if and only if $Y_{m|n_1}$ is avoidable. Therefore, since $Y_{m|n_1}$ is avoidable if and only if $X_{m|n_1}$ is, the claim follows. □

The proof of the following proposition is almost identical to the above.

Proposition 2.26.

- *If $n_1+1 \equiv 0 \bmod 2m+2$, then $Y_{m|n_1,n_2}$ is avoidable if and only if $X_{m|n_2}$ is avoidable.*
- *If $n_2+1 \equiv 0 \bmod 2m+2$, then $Y_{m|n_1,n_2}$ is avoidable if and only if $X_{m|n_1}$ is avoidable.*

Due to Propositions 2.25 and 2.26, the avoidability of $Y_{m|n_1,n_2}$ is characterized when $2m \equiv n_1 + n_2 \bmod 2m+2$ or $0 \equiv n_1 + 1 \bmod 2m+2$ or $0 \equiv n_2 + 1 \bmod 2m+2$, equivalently, when $\langle n_1 + n_2 + 2\rangle = \{0\}$ or $\langle n_1 + 1\rangle = \{0\}$ or $\langle n_2 + 1\rangle = \{0\}$ respectively. So we will now restrict our attention to the case where $\langle n_1 + n_2 + 2\rangle \neq \{0\}$, $\langle n_1 + 1\rangle \neq \{0\}$, and $\langle n_2 + 1\rangle \neq \{0\}$ hold simultaneously.

For simplification of notation, from this point on whenever we write H, it will be implied that $H = H_{m|n_1,n_2}$.

Proposition 2.27. *Let K be the subgraph of H induced by the vertex set $\langle x\rangle$, where $x \in Z_{2m+2}$. Set $r = |\langle x\rangle|$. If $m+1$ is a vertex in K, then r is even and $\frac{r}{2}x = m+1$.*

Proof. Let k be the smallest positive integer so that $kx = m+1$. Consider j, $0 \leq j < k$, then note that $(k+j)x \equiv m+1+jx \bmod 2m+2$. Furthermore, $2kx \equiv 0 \bmod 2m+2$. If $0 \leq j_1 < j_2 < 2k$, it follows that $j_1x \neq j_2x$. Furthermore, every $y \in \langle x\rangle$ is of the form $y = qx$, where q is a nonnegative integer. Note that $(q \bmod 2k)x = qx$. It follows that there exists q, $0 \leq q < 2k$, so that $y = qx$. Therefore every $y \in \langle x\rangle$ can be written uniquely in the form $y = qx$ where $0 \leq q < 2k$, so it follows that $r = 2k$, proving the claim. □

Proposition 2.28. *Let x, y be vertices of H such that $\langle x, y\rangle$ equals the vertex set of H. Let H_0 be the subgraph of H with vertex set $\langle x\rangle$. Then $H = \bigcup_{i=0}^{\alpha-1} H_i$, where $H_i = iy + H_0$, and the H_i's are distinct, with $y + H_{\alpha-1} = H_0$. Furthermore, if $m+1$ is a vertex in H and $m+1$ is not a vertex in H_0, then α is even and $m+1$ is a vertex of $H_{\frac{\alpha}{2}}$.*

Proof. This proof is similar to the above. Let H_0 be as defined, $H_i = iy + H_0$, $0 \leq i < \alpha$, where α is the smallest positive integer so that $\alpha y \in H_0$. Then note that the vertex set of H_i is $iy + \langle x\rangle$, so $y + H_{\alpha-1} = H_0$, which represent all the cosets of $\langle x\rangle$ in $\langle x, y\rangle$ (since every element in H is of the form $q_1x + q_2y \bmod 2m+2$), so $H = \bigcup_{i=0}^{\alpha-1} H_i$ and the H_i's are distinct by an elementary abstract algebra argument. Then note that if $m+1$ is a vertex of H_j, then if $0 \leq i < j$, $H_{i+j} = m+1+H_i$, and $H_{2j} = H_0$. Furthermore, $H_{i+j} \neq H_0$, otherwise that would imply $H_j = m+1+H_0 = m+1+m+1+H_i = H_i$, $i < j$. So it follows by definition that $2j = \alpha$, proving the claim. □

For the following lemma, note that $H_{m|n_1}$ and $H_{m|n_1+n_2+1}$ are subgraphs of $H_{m|n_1,n_2}$. Also, both the following lemmas get a little convoluted, since they involve lots of inter-related subcases.

Lemma 2.29. *Assume $m+1$ is not a vertex of H, $\langle n_1+n_2+2\rangle \neq \{0\}$, $\langle n_1+1\rangle \neq \{0\}$ and $\langle n_2+1\rangle \neq \{0\}$. Then if either $\langle n_1+1\rangle \neq \langle n_1+1, n_1+n_2+2\rangle$ or $\langle n_1+n_2+2\rangle \neq \langle n_1+1, n_1+n_2+2\rangle$, then $Y_{m|n_1,n_2}$ is avoidable.*

Proof. Assume $m+1$ is not a vertex of H, $\langle n_1+n_2+2\rangle \neq \{0\}$, $\langle n_1+1\rangle \neq \{0\}$ and $\langle n_2+1\rangle \neq \{0\}$, and either $\langle n_1+1\rangle \neq \langle n_1+1, n_1+n_2+2\rangle$ or $\langle n_1+n_2+2\rangle \neq \langle n_1+1, n_1+n_2+2\rangle$. Note that if i is a vertex of H, $i+m+1$ is not by Proposition 2.23. There are two cases to consider: if $\langle n_1+1\rangle \neq \langle n_1+1, n_1+n_2+2\rangle$, then let $x = n_1+1$ and $y = n_1+n_2+2$, otherwise let $x = n_1+n_2+2$ and $y = n_1+1$.

The idea of the proof is to break the claim down into lots of subcases, but first we make a few comments that apply to the majority of these subcases. Let k be the smallest positive integer so that $ky \in \langle x\rangle$, then note that $k > 1$, since $\langle x\rangle \neq \langle x, y\rangle$ and therefore $y \notin \langle x\rangle$. Let $H_0 = H_{m|x-1}$, $H_1 = y+H_0, \ldots, H_{k-1} = y+H_{k-2}$, then note that H_i has the same number of vertices as H_0, and moreover that $|H_0| > 1$ (since the vertex set of H_0 is exactly $\langle x\rangle$, and $x \not\equiv 0 \bmod 2m+2$, so both 0 and x are elements of H_0). Furthermore, if j is a vertex of H, then j is a vertex of H_i for some unique i, since there is a bijection between cosets of $\langle x\rangle$ in $\langle x, y\rangle$ and the graphs $H_0, \ldots, H_{k-1}$, which takes cosets to graphs whose vertex set is the coset. We will denote $a_i = iy \bmod 2m+2$ (note that a_i is a vertex of H_i) and $a_k = ky \bmod 2m+2$.

As a final preparation, let us define a coloring c^* on H_0. If $j \in H_0$, let t_j be the unique integer so that $0 \leq t_j < 2m+2$ for which $j \equiv t_j x \bmod 2m+2$. Then define $c^*(j) = a$ if $j = 0$, $c^*(j) = a$ if t_j is odd, and $c^*(j) = b$ otherwise.

First, if $|H_0|$ is even, then for each $j \in H_i$ there is a unique s_j, $0 \leq s_j < |H_i| = |H_0|$, so that $j = a_i + s_j x$. If s_j is even, then define $c(j) = a$, else define $c(j) = b$. Then c is a coloring of H that avoids $Y_{m|n_1,n_2}$ (and by Lemma 2.24 a coloring of $G_{m|n_1,n_2}$ that avoids $Y_{m|n_1,n_2}$ exists, and so $Y_{m|n_1,n_2}$ is avoidable by Lemma 2.22). To see this, note that if j is a vertex of H_i and s_j is as defined above, then if $s_j < |H_0| - 1$, we have $j + x \in H_i$ and $c(j + x) = c(a_i + (s_j + 1)x) \neq c(a_i + s_j x) = c(j)$, and if $s_j = |H_0| - 1$, we have $c(j) = c(a_i + s_j x) = b \neq a = c(a_i + (s_j + 1)x) = c(j + x)$.

Second, if k is even, then let k' be the smallest positive integer so that $k'x \in \langle y \rangle$, and let us define $H'_0 = H_{m|y-1}$, $H'_1 = H'_0 + x, \ldots, H'_{k'-1} = H'_{k'-2} + x$. Then note that $|H'_0|$ is even (since $ly = 0$ implies k divides l, so l is even). Define $a'_i = ix \bmod 2m + 2$. Then for each $j \in H'_i$ there is a unique s_j, $0 \leq s_j < |H'_i| = |H'_0|$, so that $j = a'_i + s_j y$. If s_j is even, then define $c(j) = a$, else define $c(j) = b$. By the same logic as in the first case, c is a coloring of H that avoids $Y_{m|n_1,n_2}$.

Third, if k is odd, $|H_0|$ is odd, and $c^*(a_k \bmod 2m + 2) = b$, then let j be a vertex of H_i. If i is even, then define $c(j) = c^*(j - a_i)$ and so $c(a_i) = c^*(a_i - a_i) = c^*(0) = a$. Otherwise define $c(j) = \overline{c^*(j - a_i)}$, and so $c(a_i) = \overline{c^*(a_i - a_i)} = \overline{c^*(0)} = b$. Note that since $a_k \in H_0$, we have $c(a_k) = c^*(a_k - a_0) = c^*(a_k)$. To see that c is a coloring of H that avoids $Y_{m|n_1,n_2}$, let j be a vertex in H_i. Note that if $j \neq a_i$, then $c(j) \neq c(j + x)$. If $i \neq k - 1$ and $j = a_i$, then $c(j) \neq \overline{c(j)} = \overline{c(a_i)} = c(a_{i+1}) = c(j + y)$. Finally, if $i = k - 1$ and $j = a_i$ then $c(j) = a \neq b = c^*(a_k) = c(a_k) = c(a_{k-1} + y) = c(j + y)$.

Fourth, if k is odd, $|H_0|$ is odd, $a_k \not\equiv x \bmod 2m + 2$, and $c^*(a_k \bmod 2m + 2) = a$, then let j be a vertex of H_i. If $i < k - 1$, then if i is even define $c(j) = c^*(j - a_i)$, else define $c(j) = \overline{c^*(j - a_i)}$, while if $i = k - 1$, define $c(j) = c^*(j - (a_i - x))$. To show that c is a coloring of H that avoids $Y_{m|n_1,n_2}$, let j be a vertex in H_i. If $i < k - 1$ and $j \neq a_i$ then $c(j) \neq c(j + x)$, while if $i < k - 1$ and $j = a_i$ then $c(j) \neq c(j + y)$. If $i = k - 1$ and $j \neq a_i - x$ then $c(j) \neq c(j + x)$, while if $i = k - 1$ and $j = a_i - x$ then $c(j) = a \neq b = c(j + y)$. To see the latter, note that since $a_k \not\equiv x \bmod 2m + 2$, we have $a_k - x \neq 0$. Also since $c^*(a_k) = a$ and $a_k \not\equiv x \bmod 2m + 2$, $a_k = k'x$ with k' odd. So we get $c(j) = c^*(j - (a_i - x)) = c^*(0) = a$ and $c(j + y) = c(a_i + y - x) = c(a_{i+1} - x) = c(a_k - x) = c^*(a_k - x - a_0) = c^*(a_k - x) = c^*((k' - 1)x) = b$.

Fifth, if k is odd, $|H_0|$ is odd, $a_k \equiv x \bmod 2m + 2$, and $c^*(a_k \bmod 2m + 2) = a$, then note that $k > 2$, since k is odd and $k \neq 1$. Let j be a vertex of H_i, $i < k - 2$ (note that $k - 2 > 0$), then if i is even define $c(j) = c^*(j - a_i)$, else define $c(j) = \overline{c^*(j - a_i)}$. If $i = k - 2$, then define $c(j) = \overline{c^*(j - (a_i - x))}$. If $i = k - 1$, then define $c(j) = c^*(j - (a_i - 2x))$. We want to show that c is a coloring of H that avoids $Y_{m|n_1,n_2}$, so let j be a vertex in H_i. If $i < k - 2$ and $j \neq a_i$, then $c(j) \neq c(j + x)$, while if $i < k - 2$ and $j = a_i$ then $c(j) \neq c(j + y)$. If $i = k - 2$ and $j \neq a_i - x$

then $c(j) \neq c(j+x)$, while if $i = k-2$ and $j = a_i - x$ then $c(j) \neq c(j+y)$. To see the latter, $c(j) = c(a_i - x) = \overline{c^*(a_i - x - (a_i - x))} = \overline{c^*(0)} = b$ and $c(j+y) = c(a_{i+1} - x) = c(a_{k-1} - x) = c^*(a_{k-1} - x - (a_{k-1} - 2x)) = c^*(x) = a$ since $a_{k-1} - x$ is a vertex of H_{k-1}. If $i = k-1$ and $j \neq a_i - 2x$ then $c(j) \neq c(j+x)$, while if $i = k-1$ and $j = a_i - 2x$ then $c(j) \neq c(j+y)$. □

Lemma 2.30. *Assume that $m+1$ is a vertex of H, $\langle n_1 + n_2 + 2\rangle \neq \{0\}$, $\langle n_1 + 1\rangle \neq \{0\}$, $\langle n_2 + 1\rangle \neq \{0\}$, $2n_2 + n_1 + 2 \not\equiv m \bmod 2m+2$ and $n_2 - n_1 - 1 \not\equiv m \bmod 2m+2$. If either $\langle n_1 + 1\rangle \neq \langle n_1 + 1, n_1 + n_2 + 2\rangle$ or $\langle n_1 + n_2 + 2\rangle \neq \langle n_1 + 1, n_1 + n_2 + 2\rangle$, then $Y_{m|n_1,n_2}$ is avoidable.*

Proof. Let $x = n_1+1$ and $y = n_1+n_2+2$ if $\langle n_1+1\rangle \neq \langle n_1+1, n_1+n_2+2\rangle$, $x = n_1 + n_2 + 2$ and $y = n_1 + 1$ otherwise. Once again, we will proceed by breaking the claim down into lots of subcases, though first we mention a few properties common to most of the subcases. Let k be the smallest positive integer so that $ky \in \langle x\rangle$, then note that $k > 1$ because $\langle x\rangle \neq \langle x, y\rangle$. Let $H_0 = H_{m|x-1}$, $H_1 = y+H_0, \ldots, H_{k-1} = y+H_{k-2}$, then note that H_i has the same number of vertices as H_0, where $|H_0| > 1$ (since $x \not\equiv 0 \bmod 2m+2$). Furthermore, if j is a vertex of H, then j is a vertex of H_i for some unique i. Define $a_i = iy \bmod 2m+2$, and note that a_i is a vertex of H_i.

Consider the possibility that $m+1$ is not a vertex in H_0. Then there are a few cases to consider. Note that by Proposition 2.28 k is even and $m+1$ is a vertex of $H_{\frac{k}{2}}$. Define the coloring c^* of H_0 so that $c^*(0) = a$, $c^*(x) = a$, and $c^*(j+x) = \overline{c^*(j)}$ if j is a vertex of H_0, $j \not\equiv 0 \bmod 2m+2$.

First, if $|H_0|$ is even, then define a coloring c on H. If j is a vertex in H_0, then define k_j to be the smallest nonegative integer so that $j = k_j x$. If k_j is even, let $c(j) = a$, and let $c(j) = b$ otherwise. If $j \in H_i$ and $0 < i < \frac{k}{2}$, let $c(j) = c(j - a_i)$. If $i \geq \frac{k}{2}$ let $c(j) = \overline{c(j - (m+1))}$. Then c is a coloring of H that avoids $Y_{m|n_1,n_2}$. To see this, let j be a vertex in H_i. Note that $c(j+x) \neq c(j)$ if $j \not\equiv a_i + (|H_0| - 1)x \bmod 2m+2$ and, since $|H_0|$ is even, $c(j+x) = c(a_i) \neq c(j)$ if $j \equiv a_i + (|H_0| - 1)x \bmod 2m+2$. Furthermore, $j+m+1$ is a vertex of $H_{(i+\frac{k}{2}) \bmod k}$, and $c(j+m+1) \neq c(j)$ by construction.

Second, if $|H_0|$ is odd and $k = 2$, then define H_0' as the graph induced by $\langle y\rangle$, and define $H_i' = x + H_{i-1}'$. Let k' be the smallest positive integer so that $k'x \in \langle y\rangle$. Then $|H_0'|$ is even (since $ly = 0$ implies k divides l, so l is even), so either the first case above holds, or else $m+1$ is a vertex of H_0'. In the latter case, note that, since $2|H_0| = k|H_0| = k'|H_0'| = |H|$, $|H_0|$ is odd, and $|H_0'|$ is even, it follows that k' is odd and $\frac{|H_0'|}{2}$ is odd (since $k'|H_0'| = 2k'\frac{|H_0'|}{2}$ is only divisible by 2 once). Therefore either $k' > 1$, in which case $\frac{|H_0'|}{2}$ is odd and we can apply the first case below when $m+1$ is a vertex of H_0, or else $k' = 1$, in which case we can use the coloring so that given an integer q, $c(2qy) = a, c((2q+1)y) = b$, which is a coloring of H that avoids $Y_{m|n_1,n_2}$.

Third, if $|H_0|$ is odd, $k > 2$, and $\frac{k}{2} = \beta$ is odd, then define a coloring c of H as follows. If $\overline{c^*(\beta y - m - 1 \bmod 2m+2)} = a$, color so that if j is a vertex of H_0 then $c(j) = c^*(j)$, otherwise define recursively $c(j) = \overline{c(j-y)}$ (which we can do since if j is a vertex of H_i then $j - y$ is a vertex of H_{i-1}). We can check that c is a coloring of H that avoids $Y_{m|n_1,n_2}$. If $\overline{c^*(\beta y - m - 1 \bmod 2m+2)} = b$, then define $c'(j) = c(j)$ if j is a vertex of H_i with $i \notin \{\beta - 1, k-1\}$, and define $c'(j) = c(j-x)$ if j is a vertex of H_i with $i \in \{\beta - 1, k-1\}$. Then c' is a coloring of H that avoids $Y_{m|n_1,n_2}$.

Fourth, if $|H_0|$ is odd, $k > 2$, $\frac{k}{2} = \beta$ is even, and $k \neq 4$, then the possibilities are that either:

- $\overline{c^*(\beta y - m - 1 \bmod 2m+2)} = a$, in which case just define c as in the third case.
- $\beta y - m - 1 \bmod 2m+2 \neq x$, in which case define c' as in the third case.
- $\beta y - m - 1 \bmod 2m+2 = x$, in which case define c'' as $c''(j) = c'(j)$ when j is a vertex of H_i, $i \notin \{\beta-1, \beta-2, k-1, k-2\}$, $c''(j) = c'(j-x)$ otherwise.

Fifth, if $|H_0|$ is odd and $k = 4$, then set $\frac{k}{2} = \beta$. With the exception of when $x \equiv \beta y - m - 1 \equiv 2y - m - 1 \bmod 2m+2$, the arguments from the fourth case work in this case also. If $x \equiv \beta y - m - 1 \equiv 2y - m - 1 \bmod 2m+2$, then note that there are two subcases. If $x = n_1 + 1$ and $y = n_1 + n_2 + 2$, note that $x + m + 1 \equiv 2y \bmod 2m+2$. Rearranging we get $m \equiv 2n_2 + n_1 + 2 \bmod 2m+2$. If, on the other hand, $x = n_1 + n_2 + 2$ and $y = n_1 + 1$, it follows that $m \equiv n_1 - n_2 - 1 \equiv 2m+2+n_1-n_2-1 \bmod 2m+2$. Some simple algebra gives $m \equiv n_2 - n_1 - 1 \bmod 2m+2$.

Next, consider the possibility that $m+1$ is a vertex in H_0. Then $|H_0|$ is even by Proposition 2.27, so define $\alpha = \frac{|H_0|}{2}$. Proposition 2.27 also implies that $\alpha x = m + 1$. Define the coloring c^* of H_0 so that $c^*(0) = a$, $c^*(x) = a$, $c^*(tx) = \overline{c^*(tx-x)}$ when $1 < t < \alpha$, and else $c^*(tx) = \overline{c^*(tx - \alpha x)}$. Then there are a few cases to consider.

First, if α is odd, then we will define a coloring c on H. If j is a vertex of H_0, let k_j be the smallest positive integer so that $j = k_j x$. If k_j is even let $c(j) = a$, otherwise let $c(j) = b$. And if j is a vertex of H_i, $i \neq 0$, define

$c(j) = c(j - a_i)$. Note that $c(j) \neq c(j + x)$ and $c(j + m + 1) \neq c(j)$, so c is a coloring of H that avoids $Y_{m|n_1,n_2}$.

Second, if α is even and k is odd, then note that if $j \in H_0$, $j \not\equiv 0 \bmod 2m + 2$, and $j \not\equiv \alpha x \bmod 2m + 2$, then $c^*(j) \neq c^*(j + x)$. Moreover, since $\alpha x = m + 1$, $c^*(j) \neq c^*(j + m + 1) = c^*(j + \alpha x)$ for $j \in H_0$. There are three possibilities:

- If $c^*(ky) = b$, then define $c(j) = c^*(j)$ if $j \in H_0$, and recursively define $c(j) = c(j - y)$.
- If $ky \neq x$, then define $c'(j) = c(j)$ if $j \notin H_{k-1}$, $c'(j) = c(j - x)$ otherwise.
- If $ky = x$, then define $c''(j) = c'(j)$ if $j \notin H_{k-1}, H_{k-2}$, $c''(j) = c'(j - x)$ otherwise.

Third, if α is even, k is even, and $k \neq 2$, then we have three cases to consider:

- If $c^*(ky) = a$, then define $c(j) = c^*(j)$ if $j \in H_0$, and recursively define $c(j) = c(j - y)$.
- If $ky \neq (\alpha + 1)x$, then define $c'(j) = c(j)$ if $j \notin H_{k-1}$, $c'(j) = c(j - x)$ otherwise.
- If $ky = (\alpha + 1)x$, then define $c''(j) = c'(j)$ if $j \notin H_{k-1}, H_{k-2}$, $c''(j) = c'(j - x)$ otherwise.

Fourth, if α is even and $k = 2$, then note that except when $ky = (\alpha+1)x$ the same arguments work for this case as in the third case. So the only case to consider is when $2y \equiv (\alpha + 1)x \equiv m + 1 + x \bmod 2m + 2$, which implies $2y - x - 1 \equiv m \bmod 2m + 2$. If $x = n_1 + 1$ and $y = n_1 + n_2 + 2$, then $m \equiv 2n_2 + n_1 + 2 \bmod 2m + 2$. Otherwise $x = n_1 + n_2 + 2$ and $y = n_1 + 1$, $m \equiv n_1 - n_2 - 1 \equiv 2m + 2 + n_1 - n_2 - 1 \bmod 2m + 2$ which gives $m \equiv n_2 - n_1 - 1 \bmod 2m + 2$. The claim then follows. □

Note that Lemma 2.29 and Lemma 2.30 account for a large number of avoidable sets.

Corollary 2.31. *If all of the following conditions hold*

- $\langle n_1 + n_2 + 2\rangle \neq \{0\}$,
- $\langle n_1 + 1\rangle \neq \{0\}$,
- $\langle n_2 + 1\rangle \neq \{0\}$,
- *either* $\langle n_1+1\rangle \neq \langle n_1+1, n_1+n_2+2\rangle$ *or* $\langle n_1+n_2+2\rangle \neq \langle n_1+1, n_1+n_2+2\rangle$,
- $2n_2 + n_1 + 2 \not\equiv m \bmod 2m + 2$,
- $n_2 - n_1 - 1 \not\equiv m \bmod 2m + 2$,

then $Y_{m|n_1,n_2}$ *is avoidable.*

Proof. The claim follows from Lemma 2.29 and Lemma 2.30. □

In order to prove Theorem 2.40, from now on we only need to consider the case when $\langle n_1+1\rangle = \langle n_1+n_2+2\rangle = \langle n_2+1\rangle \neq \{0\}$. This implies that the vertex set of H is equal to $\langle n_1+1\rangle$, since $\langle n_1+1\rangle = \langle n_1+1, n_1+n_2+2\rangle$. We will henceforward assume that these conditions hold on n_1 and n_2.

Lemma 2.32. *There exist no nonnegative integers m, n_1, n_2 so that $\langle n_1+1\rangle = \langle n_1+n_2+2\rangle = \langle n_2+1\rangle \neq \{0\}$ and $m+1 \in \langle n_1+1\rangle$ (or $m+1$ is a vertex of H).*

Proof. Assume such m, n_1 and n_2 exist. Then note that $|\langle n_1+1\rangle| = r$ is even by Proposition 2.27. Then there exists an integer k, $1 < k < r$, so that $k(n_1+1) \equiv n_2+1 \bmod 2m+2$. Furthermore, note that $(k+1)(n_1+1) \equiv (n_1+n_2+2) \bmod 2m+2$. Obviously either k is even or $k+1$ is even. If k is even then, since $n_1+1 \in \langle n_2+1\rangle$, there exists a positive integer, q, so that $n_1+1 \equiv q(n_2+1) \equiv kq(n_1+1) \bmod 2m+2$. This implies that $qk \equiv 1 \bmod r$, so $1 = \alpha r + qk$ for some integer α. Since r and k are both even, however, this implies that 1 is even, a contradiction. If, on the other hand, $k+1$ is even, the same argument applies, again leading to a contradiction. Therefore the claim follows. □

All that is left to consider is the case when $m+1$ is not a vertex of H.

Lemma 2.33. *If $\langle n_1+1\rangle = \langle n_2+1\rangle = \langle n_1+n_2+2\rangle \neq \{0\}$, $m+1$ is not a vertex of H and $|H|$ is even, then $Y_{m|n_1,n_2}$ is avoidable.*

Proof. Color the vertices of H as follows: for any integer $x < \frac{|H|}{2}$ let $c((2x+1)(n_1+1)) = a$ and $c((2x)(n_1+1)) = b$. Then c is a coloring of H that avoids $Y_{m|n_1,n_2}$, since $c(i) \neq c(i+n_1+1)$. □

Lemma 2.34. *If $\langle n_1+1\rangle = \langle n_2+1\rangle = \langle n_1+n_2+2\rangle \neq \{0\}$, $m+1$ is not a vertex of H, then let k be the smallest positive integer so that $k(n_1+n_2+2) \equiv n_1+1 \bmod 2m+2$. If k is even, then $Y_{m|n_1,n_2}$ is avoidable.*

Proof. Consider the following coloring. Let $c(0) = a$, and, for any i, $0 < i < |H|$, if i is odd let $c(i(n_1+n_2+2)) = a$, and otherwise set $c(i(n_1+n_2+2)) = b$. Since for any $i \neq 0$, $c(i) \neq c(i+n_1+n_2+2)$, and since $c(0) = a \neq b = c(0+n_1+1)$, it follows that c is a coloring of H that avoids $Y_{m|n_1,n_2}$. Therefore the claim follows. □

Lemma 2.35. *If $\langle n_1+1\rangle = \langle n_2+1\rangle = \langle n_1+n_2+2\rangle \neq \{0\}$, $m+1$ is not a vertex of H, then let k_1 be the smallest positive integer so that*

$k_1(n_1+n_2+2) \equiv n_1+1 \bmod 2m+2$, and let k_2 be the smallest positive integer so that $k_2(n_1+n_2+2) \equiv n_2+1 \bmod 2m+2$. Then either $k_1 < \frac{|H|}{2}+1$ or $k_2 < \frac{|H|}{2}+1$.

Proof. We know that $(n_1+1)+(n_2+1) = n_1+n_2+2$, so $k_1+k_2 \equiv 1 \bmod |H|$. Furthermore, since $1 < k_1 < |H|$ and $1 < k_2 < |H|$ are minimal, $1 < k_1+k_2 < 2|H| < 2|H|+1$, so the only possibility is that $k_1+k_2 = |H|+1$. The claim easily follows. □

Lemma 2.36. *Let k_1, k_2 be as defined in Lemma 2.35. If*

- *$\langle n_1+1\rangle = \langle n_2+1\rangle = \langle n_1+n_2+2\rangle \neq \{0\}$,*
- *$m+1$ is not a vertex of H,*
- *$k_1 \leq 3$ or $k_2 \leq 3$,*
- *$|H| > (k_1)^2-1$ or $|H| > (k_2)^2-1$,*

then there is a coloring of H that avoids $Y_{m|n_1,n_2}$.

Proof. Assume that $k_1 = \min\{k_1,k_2\}$, and so $k_1 \leq 3$ and $r = |H| > (k_1)^2-1$. We want to show that there exist nonnegative integers x and y so that $x+y$ is even and $r = x(k_1-1)+yk_1$. To see this, by Lemma 2.34, we can assume that k_1 is odd. Note for any nonnegative integer q that (k_1+1+2q) is even, so $(k_1+1+2q)(k_1-1)$ can be written in the above form. If p is such that $k_1+1+2q \geq p \geq 0$, then $(k_1+1+2q)-p+p = k_1+1+2q$ is even and $(k_1+1+2q)(k_1-1)+p = (k_1+1+2q-p)(k_1-1)+pk_1$, so $(k_1+1+2q)(k_1-1)+p$ can also be written in the above form.

Furthermore, note that under our assumptions r falls into an interval of the form $[(k_1+1+2q)(k_1-1)..(k_1+1+2q)k_1]$, so there exist nonnegative integers x and y so that $x+y$ is even and $r = x(k_1-1)+yk_1$. In such case, we color H as follows. Since $\langle n_1+n_2+2\rangle$ equals the vertex set of H, if j is a vertex of H there exists a unique integer t, $0 \leq t < |H| = r = x(k_1-1)+yk_1$, so that $j = t(n_1+n_2+2)$. If $t < x(k_1-1)$, let β be the largest integer so that $t \geq \beta(k_1-1)$. Then if β is even define $c(j) = a$, otherwise let $c(j) = b$. If $t \geq x(k_1-1)$, let β be the largest integer so that $t \geq x(k_1-1)+\beta k_1$. If $x+\beta$ is even, let $c(j) = a$, otherwise let $c(j) = b$. Then c is a coloring of H that avoids $Y_{m|n_1,n_2}$, since $c(j) \neq c(j+n_1+1)$ or $c(j) \neq c(j+n_1+n_2+2)$, so the lemma follows. □

Lemma 2.37. *Let k_1, k_2 be as defined in Lemma 2.35. If $m+1$ is not a vertex of H, $\langle n_1+1\rangle = \langle n_2+1\rangle = \langle n_1+n_2+2\rangle \neq \{0\}$, $k_1 > 3$ and $k_2 > 3$, then $Y_{m|n_1,n_2}$ is avoidable.*

Proof. We can assume k_1 and k_2 are odd, since otherwise by Lemma 2.34 $Y_{m|n_1,n_2}$ is avoidable. Similarly, we can assume that $|H|$ is odd by Lemma 2.33.

Furthermore, note by Proposition 2.23 that if j is a vertex of H, $j+m+1$ is not.

By Lemma 2.35, either $k_1 < \frac{|H|}{2}+1$ or $k_2 < \frac{|H|}{2}+1$. Without loss of generality, we can assume $k_1 < \frac{|H|}{2}+1$. Then $k_1 - 1 < \frac{|H|}{2}$, so, if $\beta = \lfloor \frac{|H|}{k_1-1} \rfloor$, it follows that $\beta \geq 2$. We then break the problem into three cases where j is a vertex of H.

First, if β is even, then let us define the coloring of $\{0, n_1+n_2+2, \ldots, (k_1-2)(n_1+n_2+2)\}$ so that $c^*(0) = a$, $c^*(n_1+n_2+2) = a$, and for $1 < i < k_1 - 1$, define recursively $c^*(i(n_1+n_2+2)) = \overline{c^*((i-1)(n_1+n_2+2))}$. Then let t_j be the smallest nonnegative integer so that $t_j(n_1+n_2+2) = j$, and let $t_j = \alpha_j(k_1-1) + r_j$ so that $0 \leq r_j < k_1 - 1$. If α_j is even, then $c(j) = c^*(r_j(n_1+n_2+2))$, otherwise let $c(j) = \overline{c^*(r_j(n_1+n_2+2))}$. This coloring c of H avoids $Y_{m|n_1,n_2}$. To see this, note that if $0 \equiv k_1 - 1 \bmod |H|$, then $c(j) \neq c(j+n_1+1)$, and otherwise $c(j) \neq c(j+n_1+n_2+2)$ so the claim follows.

Second, if β is odd and $1 \neq |H| \bmod k_1 - 1$, then let c^*, α_j, r_j, t_j be as defined above. Then let c be as defined above. We define c' as follows: if $t_j < |H|+1-k_1$, then $c'(j) = c(j)$. Else, define recursively $c'(|H|+1-k_1) = b$, $c'(j) = c'(t_j(n_1+n_2+2)) = \overline{c'((t_j-1)(n_1+n_2+2))}$. Then c' is a coloring of H that avoids $Y_{m|n_1,n_2}$ by a similar argument to that in the first case, which is what we wanted.

Third, if β is odd and $1 = |H| \bmod k_1 - 1$, then let $c^*, \alpha_j, r_j, t_j, c, c'$ be as defined above. If $t_j \leq k_1 - 3$ then define $c''(j) = c'(j)$, if $k_1 - 3 < t_j < (|H|-2)(n_1+n_2+2)$ define $c''(j) = c'((t_j+2)(n_1+n_2+2))$, and also define $c''((|H|-1)(n_1+n_2+2)) = b$, and $c''((|H|-2)(n_1+n_2+2)) = a$. Then c'' is a coloring of H that avoids $Y_{m|n_1,n_2}$ by a similar argument to that in the first case, proving the claim. □

Lemma 2.38. *If $m+1$ is not a vertex of H, $\langle n_1+1\rangle = \langle n_2+1\rangle = \langle n_1+n_2+2\rangle \neq \{0\}$, and H is not isomorphic to any of $H_{6|n_1',n_2'}$, where $n_1' \neq n_2'$, $n_1', n_2' \in \{1,3,7\}$, then $Y_{m|n_1,n_2}$ is avoidable.*

Proof. Let k_1, k_2 be as defined in Lemma 2.35. As in the proof of Lemma 2.37, k_1, k_2, and $|H|$ are odd. If $k_1 > 3$ and $k_2 > 3$, then by Lemma 2.37 $Y_{m|n_1,n_2}$ is avoidable. Otherwise, if $|H| > 3^2 - 1 = 8$, it follows by Lemma 2.36 that $Y_{m|n_1,n_2}$ is avoidable. Finally it is just a matter of checking all possible graphs with at most eight vertices in order to verify the result. □

Lemma 2.39. *If* $\gcd(m+1, n_1+1, n_2+1) = 1$*,* $\langle n_1+1\rangle = \langle n_2+1\rangle = \langle n_1+n_2+2\rangle \neq \{0\}$*,* H *is isomorphic to* $H_{6|1,3}$*,* $H_{6|1,7}$*, or* $H_{6|3,7}$*, then* $m = 6$ *and*

$$c_{2m+2}(Y_{m|n_1,n_2}) \in \{Y_{6|1,3}, Y_{6|1,7}, Y_{6|3,7}, Y_{6|3,1}, Y_{6|7,1}, Y_{6|7,3}\}$$

Proof. Since $\gcd(m+1, n_1+1, n_2+1) = 1$, note that $K = \langle n_1+1\rangle \cup ((m+1) + \langle n_1+1\rangle)$ is a subgroup of Z_{2m+2}. Furthermore, $m+1, n_1+1, n_2+1 \in K$. From here it follows that $1 = \gcd(m+1, n_1+1, n_2+1) \in K$, so $K = Z_{2m+2}$. Since H is isomorphic to one of $H_{6|1,3}$, $H_{6|1,7}$, or $H_{6|3,7}$, the number of elements in the vertex set of H is seven. Since $\langle n_1+1\rangle = \langle n_1+1, n_1+n_2+2\rangle$ is the vertex set of H, $|\langle n_1+1\rangle| = 7$. This implies K has fourteen elements. Since $K = Z_{2m+2}$, it follows that $2m+2 = 14$, so $m = 6$. Therefore $c_{2m+2}(Y_{m|n_1,n_2}) = Y_{6|n_1',n_2'}$, where n_1' and n_2' are nonnegative integers. From this point it is easy to write a computer program to check all 14-canonical sets of the form $Y_{6|n_1',n_2'}$, and realizing the only ones with $H_{6|n_1',n_2'}$ isomorphic to one of $H_{6|1,3}$, $H_{6|1,7}$, or $H_{6|3,7}$ are $\{Y_{6|1,3}, Y_{6|1,7}, Y_{6|3,7}, Y_{6|3,1}, Y_{6|7,1}, Y_{6|7,3}\}$. The claim follows. □

Now it is time to put everything together.

Theorem 2.40. *Let* m, n_1, n_2 *be nonegative integers such that* $n_1 \leq n_2$ *and* $\gcd(m+1, n_1+1, n_2+1) = 1$*. In case* $m = 6$*, also assume that if* $n_1' \neq n_2'$ *and* $n_1', n_2' \in \{1, 3, 7\}$*, then* $c_{2m+2}(Y_{m|n_1,n_2}) \neq Y_{m|n_1',n_2'}$*. Then* $Y_{m|n_1,n_2}$ *is avoidable if all of (2.10)–(2.15) are satisfied:*

$$n_1 + n_2 \not\equiv 2m \bmod 2m+2 \tag{2.10}$$

$$n_1 + 1 \not\equiv 0 \bmod 2m+2 \tag{2.11}$$

$$n_2 + 1 \not\equiv 0 \bmod 2m+2 \tag{2.12}$$

$$2n_1 + n_2 + 2 \not\equiv m \bmod 2m+2 \tag{2.13}$$

$$2n_2 + n_1 + 2 \not\equiv m \bmod 2m+2 \tag{2.14}$$

$$n_2 - n_1 - 1 \not\equiv m \bmod 2m+2 \tag{2.15}$$

Proof. Let m, n_1 and n_2 satisfy all the conditions. Note the $m = 6$ condition that "if $n_1' \neq n_2'$ and $n_1', n_2' \in \{1, 3, 7\}$, then $c_{2m+2}(Y_{m|n_1,n_2}) \neq Y_{m|n_1',n_2'}$" implies that $(m, n_1, n_2) \notin \{(6,1,3), (6,3,7), (6,1,7)\}$. We need to consider a few cases. If $\langle n_1+1\rangle \neq \langle n_1+1, n_1+n_2+2\rangle$ or $\langle n_1+n_2+2\rangle \neq \langle n_1+1, n_1+n_2+2\rangle$, then Corollary 2.31 implies that $Y_{m|n_1,n_2}$

is avoidable. If $\langle n_1+1\rangle = \langle n_2+1\rangle = \langle n_1+n_2+2\rangle$ and $m+1$ is not a vertex of H, then by Lemma 2.39 we have that H is not isomorphic to any of $H_{6|1,3}, H_{6|1,7}, H_{6|3,7}, H_{6|3,1}, H_{6|7,1}$ or $H_{6|7,3}$, and by Lemma 2.38 we get that $Y_{m|n_1,n_2}$ is avoidable. The only case left is when $\langle n_1+1\rangle = \langle n_1+n_2+2\rangle = \langle n_2+1\rangle$ and $m+1$ is a vertex in H, but this case never occurs by Lemma 2.32, so the theorem follows. $\square$

Indeed, Theorem 2.40 will help us prove Conjectures 2.9 and 2.10. Therefore, in order to prove them we need only consider the cases:

- $n_2 - n_1 - 1 \equiv m \bmod 2m+2$,
- $2n_1 + n_2 + 2 \equiv m \bmod 2m+2$ or $2n_2 + n_1 + 2 \equiv m \bmod 2m+2$,
- $n_1 + n_2 + 2 \equiv 0 \bmod 2m+2$, $n_1 + 1 \equiv 0 \bmod 2m+2$, or $n_2 + 1 \equiv 0 \bmod 2m+2$,
- $m = 6$,

which will be considered in Sections 2.5.1–2.5.4 that follow. Note we can always assume that $m > 2$, since [4] proves Conjecture 2.9 when $0 \leq m \leq 2$. We state this claim on occasion without justification, so we put this justification here.

Before continuing, we need to prove a technical lemma that will be very useful in the next few subsections.

Lemma 2.41. *Let p be a positive odd integer, m a nonnegative integer, and x a positive integer. Assume $u = (ab)^{\frac{p-1}{2}}b$, $x-(m+1) = p$, $\gcd(x, m+1) = 1$. Then for every integer k, $0 \leq k < p(m+1)$, there exists a unique pair of integers, i_k and j_k, so that $0 \leq i_k < p$, $0 \leq j_k < m+1$, and $i_k(m+1) + j_k x \equiv k \bmod p(m+1)$. For every such k we define v of length $p(m+1)$ so that $v(k) = u(i_k + j_k \bmod p)$. Then*

- *If $v(k) = a$, then $v(k+m+1 \bmod p(m+1)) = b$.*
- *If $v(k) = b$ and z is an integer so that $1 < 2z+1 < p$, then either $v(k+x \bmod p(m+1)) = a$ or $v(k-(2z+1)(m+1)+2x \bmod p(m+1)) = a$.*
- *If $v(k) = b$ and z is an integer so that $0 < 2z < p$, then either $v(k+x \bmod p(m+1)) = a$ or $v(k+2z(m+1) \bmod p(m+1)) = a$.*

Proof. Let u, p, x, m be as defined above. First note that $\gcd(p, x) = 1$, and so $\gcd(p(m+1), x) = 1$. Similarly, $\gcd(m+1, p) = 1$. The first statement is trivially equivalent to the fact that any number k such that $0 \leq k < p(m+1)$ is uniquely determined by the pair $(k \bmod p, k \bmod m+1)$.

Therefore we can define i_k, j_k and v as above. Then assume that $0 \leq k < p(m+1)$. First consider the case that $v(k) = a$, then $u(i_k + j_k \bmod p) = a$. Furthermore, note that this implies $u(i_k + 1 + j_k \bmod p) = b$, and so $v((i_k + 1)(m + 1) + j_k x \bmod p(m + 1)) = v(k + m + 1 \bmod p(m + 1)) = b$.

Next consider the possibility that $v(k) = b$. We know that $u(i_k + j_k \bmod p) = b$. If $i_k + j_k \not\equiv p - 2 \bmod p$, then $v(k + x \bmod p(m + 1)) = u(i_k + j_k + 1 \bmod p) = a$. If $i_k + j_k \equiv p - 2 \bmod p$ then $v(k - (2z + 1)(m + 1) + 2x \bmod p(m + 1)) = u(i_k + j_k - (2z + 1) + 2 \bmod p) = u(p - 2z - 1) = a$ (since $0 \leq p - 2z - 1 < p - 1$, and $p - 2z - 1$ is even), and so the claim follows.

Finally, we simply need to show that if $i_k + j_k \equiv p - 2 \bmod p$ then $v(k + 2z(m + 1) \bmod p(m + 1)) = a$, but this simply implies $u(p - 2 + 2z \bmod p) = a$, which is easy to see. □

2.5.1 *The* $m \equiv n_2 - n_1 - 1 \bmod 2m + 2$ *case*

The following lemma takes care of the $m \equiv n_2 - n_1 - 1 \bmod 2m + 2$ case.

Lemma 2.42. *Let* m, n_1, n_2 *be nonnegative integers such that* $\gcd(m + 1, n_1 + 1, n_2 + 1) = 1$. *If for some integer* $z > 0$, $m + z(2m + 2) = n_2 - n_1 - 1$, *then* $X_{m|n_1,n_2}$ *is avoidable.*

Proof. As mentioned earlier, we can assume that $m > 2$. It suffices to prove the result for $X_{m|n_2,n_1}$. Begin by noting that $\gcd(m + 1, n_1 + 1) = 1$ and $\gcd(m+1, n_2+1) = 1$. To see this, assume this was not the case. Then note that $(2z+1)(m+1) = (n_2+1)-(n_1+1)$, so if s divides n_2+1 and $m+1$ or $n_1 + 1$ and $m + 1$, it also divides the remaining member, so it divides $\gcd(m+1, n_1+1, n_2+1) = 1$, and therefore $s = 1$. By definition, this means that $\gcd(m + 1, n_1 + 1) = 1$ and $\gcd(m + 1, n_2 + 1) = 1$. It follows that at most one of $n_1 + 1$, $n_2 + 1$ and $m + 1$ is even (since if $m + 1$ is even, $n_1 + 1$ and $n_2 + 1$ are odd because $\gcd(m + 1, n_1 + 1) = 1$ and $\gcd(m + 1, n_2 + 1) = 1$; if $n_1 + 1$ is even then $m + 1$ is odd, so $n_2 + 1 = (2z + 1)(m + 1) + n_1 + 1$ is odd, and a similar argument holds if $n_2 + 1$ is even). Furthermore, if $m + 1$ is odd, either $n_1 + 1$ is odd, implying $X_{m|n_1}$ and thus $X_{m|n_1,n_2}$ is avoidable, or else $n_2 + 1$ is odd, implying $X_{m|n_2}$ and thus $X_{m|n_1,n_2}$ is avoidable. Therefore, we can assume $m + 1$ is even, and that $n_1 + 1$ and $n_2 + 1$ are odd.

Let $p = n_2 - m = n_2 + 1 - (m + 1)$. It follows from the above argument that we can assume p is odd. Furthermore, note that $p > 2z + 1$, since $n_2 - m = (2z + 1)(m + 1) + n_1 - m = 2z(m + 1) + n_1 + 1 > 2z + 1$. Then let us define the word $u = (ab)^{\frac{p-1}{2}} b$. Note that $|u| = p$. Define v as in

Lemma 2.41. Then we can let $x = n_2+1$. We claim $w = v^{\mathbb{Z}}$ avoids $X_{m|n_2,n_1}$. To see this, first assume that $w(k) = v(k \bmod p(m+1)) = a$, then by Lemma 2.41 it follows that $w(k+m+1) = v(k+m+1 \bmod p(m+1)) = b$, and so w avoids $a\diamond^m a$. Furthermore, if $w(k) = v(k \bmod p(m+1)) = b$ then either $a = v(k+n_2+1 \bmod p(m+1)) = w(k+n_2+1)$ or $a = v(k-(2z+1)(m+1)+2x \bmod p(m+1)) = w(k-(2z+1)(m+1)+2x) = w(k+n_1+n_2+2)$. Therefore, assume that there exists an occurrence of $b\diamond^{n_2}b\diamond^{n_1}b$. Then there exists an integer i so that $w(i) = b$, $w(i+n_2+1) = b$, and $w(i+n_1+n_2+2) = b$, but by the above argument that is impossible, so w avoids $b\diamond^{n_2}b\diamond^{n_1}b$ and thus avoids $X_{m|n_2,n_1}$, proving the lemma. $\square$

2.5.2 *The* $m \equiv 2n_1 + n_2 + 2 \bmod 2m+2$ *and* $m \equiv 2n_2 + n_1 + 2 \bmod 2m+2$ *cases*

Here we discuss the $m \equiv 2n_1 + n_2 + 2 \bmod 2m+2$ case (the $m \equiv 2n_2 + n_1 + 2 \bmod 2m+2$ case is symmetric).

Lemma 2.43. *If* m, n_1, n_2, z *are nonnegative integers such that* $z > 0$, $\gcd(m+1, n_1+1, n_2+1) = 1$, *and* $z(2m+2)+m+1 = (2z+1)(m+1) = 2n_1+n_2+3$, *then* $X_{m|n_1,n_2}$ *is avoidable.*

Proof. We know this holds when $m = 0$, $m = 1$ or $m = 2$, so let us assume $m > 2$.

Begin by noting that $\gcd(m+1, n_1+n_2+2) = 1$. If this were not the case, then it would follow that $\gcd(m+1, n_1+n_2+2) = s > 1$. This, however, implies that s divides $(2z+1)(m+1)-(n_1+n_2+2) = n_2+1$, and, moreover, s divides $n_1+n_2+2-(n_2+1) = n_1+1$, so s divides $\gcd(m+1, n_1+1, n_2+1) = 1$, a contradiction. Similarly, we have that $\gcd(n_2+1, m+1) = 1$. This implies that at most one of $m+1$ and n_2+1 is even, but if $m+1$ is odd, $(2z+1)(m+1)-2(n_2+1) = n_1+1$ is odd, so $X_{m|n_1}$ is avoidable. Therefore we can assume $m+1$ is even, n_2+1 odd, which implies n_1+1 is even, so n_1+n_2+2 is odd.

Let $p = n_1+n_2-m+1 = (n_1+n_2+2)-(m+1)$. First note that p is odd. Let us assume that either $m > 5$, or else $m \geq 3$ and $z \geq 2$ (this excludes a finite number of cases that are easily checked). Then we have that $p > 2z+1$. To see this, first note that $n_2+1 < (z+\frac{1}{2})(m+1)$, so $p = (n_1+n_2+2)-(m+1) = 2n_1+n_2+3-m-1-n_2-1 = (2z+1)(m+1)-m-1-n_2-1 = 2z(m+1)-(n_2+1) > 2z(m+1)-(z+\frac{1}{2})(m+1) = (z-\frac{1}{2})(m+1)$. Then note when $m > 5$ that $p > (z-\frac{1}{2})(m+1) > (z-\frac{1}{2})6 = 2z+4z-3 \geq$

$2z+z \geq 2z+1$. Similarly when $m \geq 3$ and $z \geq 2$ that $p > (z-\frac{1}{2})(m+1) \geq (z-\frac{1}{2})4 = 4z-2 = 2z+2z-2 \geq 2z+4-2 = 2z+2 > 2z+1$, so $p > 2z+1$.

Then let us define a word of lenth p, $u = (ab)^{\frac{p-1}{2}}b$. Define v as in Lemma 2.41. Then we can let $x = n_1 + n_2 + 2$. We claim that $w = v^{\mathbb{Z}}$ avoids $X_{m|n_1,n_2}$. To see this, first assume that $w(k) = a$, then by Lemma 2.41 it follows that $w(k+m+1) = v(k+m+1 \bmod p(m+1)) = b$, and so w avoids $a\diamond^m a$. Furthermore, if $w(k) = v(k \bmod p(m+1)) = b$ then either $a = v(k+n_1+n_2+2 \bmod p(m+1)) = w(k+n_1+n_2+2)$ or $a = v(k-(2z+1)(m+1)+2x \bmod p(m+1)) = w(k-(2z+1)(m+1)+2x) = w(k-2n_2-n_1-3+2n_2+2n_1+4) = w(k+n_1+1)$. Therefore, assume that there exists an occurrence of $b\diamond^{n_1}b\diamond^{n_2}b$, then there exists an i so that $w(i) = b$, $w(i+n_1+1) = b$, and $w(i+n_1+n_2+2) = b$, but by the above argument that is impossible, so w avoids $b\diamond^{n_1}b\diamond^{n_2}b$ and thus $X_{m|n_1,n_2}$, proving the lemma. $\square$

2.5.3 *The* $n_1 + n_2 + 2 \equiv 0 \bmod 2m+2$, $n_1 + 1 \equiv 0 \bmod 2m+2$, *and* $n_2 + 1 \equiv 0 \bmod 2m+2$ *cases*

First, we treat the $n_1 + n_2 + 2 \equiv 0 \bmod 2m+2$ case.

Lemma 2.44. *Let* m, n_1, n_2 *be nonegative integers,* $n_1 \leq n_2$, $\gcd(m+1, n_1+1, n_2+1) = 1$, *$z$ an integer so that* $z > 1$, *and* $z(2m+2) = n_1+n_2+2$. *Then* $X_{m|n_1,n_2}$ *is avoidable.*

Proof. It suffices to prove the claim for $X_{m|n_2,n_1}$. We can assume that $m > 2$, since we already know it is true otherwise. Furthermore, we can easily check the claim when $m \in \{3,4\}$ and $z \in \{2,3,4\}$, so we can assume that $m \in \{3,4\}$ and $z > 4$ or $m > 4$ and $z > 1$. Then set $p = n_2 - m$. Then $p > 2z$, since $2z(m+1) = n_1+n_2+2 \geq 2(n_1+1)$, so $z(m+1) \geq n_1+1$ and $z(m+1) \leq n_2+1$. If $m > 4$ and $z > 1$, then $p = n_2 - m = 2z(m+1) - n_1 - m - 2 \geq z(m+1) - (m+1) = (z-1)(m+1) = 2z + z(m-1) - m - 1 \geq 2z + 2(m-1) - m - 2 = 2z + m - 4 > 2z$. If $m \in \{3,4\}$ and $z > 4$ then $p = n_2 - m = 2z(m+1) - n_1 - m - 2 \geq z(m+1) - (m+1) = (z-1)(m+1) = 2z + z(m-1) - m - 1 \geq 2z + 3(m-1) - m - 1 = 2z + 2m - 4 > 2z$. Furthermore, note that $\gcd(m+1, n_2+1) = 1$ and that p is odd, by a similar argument to those above.

We then define $u = (ab)^{\frac{p-1}{2}}b$. Define v as in Lemma 2.41. Then we can let $x = n_2 + 1$. We claim that $w = v^{\mathbb{Z}}$ avoids $X_{m|n_2,n_1}$. To see this, first assume that $w(k) = a$, then by Lemma 2.41 it follows that $w(k+m+1) =$

$v(k+m+1 \bmod p(m+1)) = b$, and so w avoids $a\diamond^m a$. Furthermore, if $w(k) = v(k \bmod p(m+1)) = b$ then either $a = v(k+x \bmod p(m+1)) = v(k+n_2+1 \bmod p(m+1)) = w(k+n_2+1)$ or $a = v(k+2z(m+1) \bmod p(m+1)) = w(k+2z(m+1)) = w(k+n_1+n_2+2)$. Therefore, assume that there exists an occurrence of $b\diamond^{n_2}b\diamond^{n_1}b$, then there exists an i so that $w(i) = b$, $w(i+n_2+1) = b$, and $w(i+n_1+n_2+2) = b$, but by the above argument that is impossible, so w avoids $b\diamond^{n_2}b\diamond^{n_1}b$ and thus $X_{m|n_2,n_1}$, proving the lemma. □

Now, we treat the $n_1+1 \equiv 0 \bmod 2m+2$ case (the $n_2+1 \equiv 0 \bmod 2m+2$ case is symmetric).

Lemma 2.45. *Let m, n_1, n_2, z be nonnegative integers so that $z > 0$, $n_1 + 1 = 2z(m+1)$, $\gcd(m+1, n_1+1, n_2+1) = 1$. Then $X_{m|n_1,n_2}$ is avoidable.*

Proof. This is true when $m = 0$, so we can assume $m > 0$. Note that $\gcd(m+1, n_1+n_2+2) = 1$. To see this, note that $\gcd(m+1, n_2+1) = \gcd(m+1, 2z(m+1), n_2+1) = \gcd(m+1, n_1+1, n_2+1) = 1$, and so $\gcd(m+1, n_1+n_2+2) = \gcd(m+1, n_2+1+2z(m+1)) = \gcd(m+1, n_2+1) = 1$. Furthermore, note that we can assume $p = n_1+n_2-m+1 = n_1+n_2+2-(m+1)$ is odd. To see this, first consider the case $m+1$ is odd, then if p is even, $p+(m+1)-(n_1+1) = p+(m+1)-2z(m+1) = n_2+1$ is odd, so $X_{m|n_2}$ is avoidable and thus $X_{m|n_1,n_2}$ is avoidable. If $m+1$ is even and p is even then $p+(m+1)-2z(m+1) = n_2+1$ is even, as is n_1+1, which contradicts the claim that $\gcd(m+1, n_1+1, n_2+1) = 1$. Therefore, we can assume p is odd. Furthermore, note that $p > 2z$, since $p = n_1+1+n_2+1-(m+1) = n_2+1+2z(m+1)-(m+1) = n_2+1+(2z-1)(m+1) > 2z$.

Let us define $u = (ab)^{\frac{p-1}{2}}b$. Define v as in Lemma 2.41. Then we can let $x = n_1+n_2+2$. We claim $w = v^{\mathbb{Z}}$ avoids $X_{m|n_1,n_2}$. To see this, first assume that $w(k) = a$, then by Lemma 2.41 it follows that $w(k+m+1) = v(k+m+1 \bmod p(m+1)) = b$, and so w avoids $a\diamond^m a$. Furthermore, if $w(k) = v(k \bmod p(m+1)) = b$ then either $a = v(k+n_1+n_2+2 \bmod p(m+1)) = w(k+n_1+n_2+2)$ or $a = v(k+2z(m+1) \bmod p(m+1)) = w(k+2z(m+1)) = w(k+n_1+1)$. Therefore, assume that there exists an occurrence of $b\diamond^{n_1}b\diamond^{n_2}b$, then there exists an i so that $w(i) = b$, $w(i+n_1+1) = b$, and $w(i+n_1+n_2+2) = b$, but by the above argument that is impossible, so w avoids $b\diamond^{n_1}b\diamond^{n_2}b$ and thus $X_{m|n_1,n_2}$, proving the lemma. □

2.5.4 *The $m = 6$ case*

We now discuss the $m = 6$ case.

Lemma 2.46. *If $n > 181$, then there exist nonnegative integers x_1, x_2 such that $n = 14x_1 + 15x_2$.*

Proof. It is easy to check that this claim holds for $n \in \{182, \ldots, 195\} = S$. Then note that S has fourteen consecutive elements, so every integer $n > 181$ can be written in the form $n = 14q + r$ for some $r \in S$. Then, however, if x_1, x_2 are the nonnegative integers so that $r = 14x_1 + 15x_2$, it follows that $n = 14(x_1 + q) + 15x_2$, and so the claim follows. $\square$

Referring to Lemma 2.46, the *Frobenius problem* gives $14 \times 15 - (14 + 15) = 181$. More generally, given a, b with $\gcd(a, b) = 1$, the largest integer not representable as $ax_1 + bx_2$ with $x_1, x_2 \geq 0$ is $x_1 x_2 - (x_1 + x_2)$.

Lemma 2.47. *If $n > 38$, then there exist nonnegative integers $x_1, \ldots, x_6$ such that $n = 13x_1 + 14x_2 + \cdots + 18x_6$.*

Proof. It is easy to check that this claim holds for $n \in \{39, \ldots, 51\} = S$. Then note that S has thirteen consecutive elements, so every integer $n > 38$ can be written in the form $n = 13q + r$ for some $r \in S$. Then, however, if $x_1, \ldots, x_6$ is the sequence of nonnegative integers such that $r = 13x_1 + 14x_2 + \cdots + 18x_6$, it follows that $n = 13(x_1 + q) + 14x_2 + \cdots + 18x_6$, and so the claim follows. $\square$

Lemma 2.48. *Let n_1, n_2 be nonnegative integers such that $n_1 \leq n_2$, $\gcd(n_1 + 1, n_2 + 1, 7) = 1$, and $(6, n_1, n_2) \neq (6, 1, 3)$. If $c_{14}(X_{6|n_1,n_2}) = X_{6|n_1',n_2'}$ where $n_1' \neq n_2'$ and $n_1', n_2' \in \{1, 3, 7\}$, then $X_{6|n_1,n_2}$ is avoidable.*

Proof. We will proceed by reducing the claim to testing a finite number of cases. Proposition 6 of [4] states that if m is even and $2m \leq \min\{n_1, n_2\}$, then $X_{m|n_1,n_2}$ is avoidable, while Proposition 8 of [4] states that for $s \in \mathbb{N}$, $s < m - 2$, and for $n > 2(m + 1)^2 + m - 1$, we have that $X_{m|m+s,n}$ is avoidable. First note we can assume that $n_1 < 12$, since if $12 \leq n_1 \leq n_2$ it follows by Proposition 6 that $X_{6|n_1,n_2}$ is avoidable. Similarly, we know that if $m = 6 \leq n_1 < 2m - 2 = 10$ and $n_2 > 2(m+1)^2 + m - 1 = 2(7)^2 + 5 = 103$ then $X_{m|n_1,n_2}$ is avoidable by Proposition 8.

Next, assume that $n_1 < 6 = m$ and that $n_2 > 187$. Then we know that $c_{n_2-6}(X_{m|n_1,n_2}) = \{a\diamond^6 a, b\diamond^{n_1} b\diamond^6 b\}$ (where $c_n(X)$ is as defined in Section 2.4). To see this is the case, first consider the fact that

$n_2 - 6 > 8$, so $c_{n_2-6}(a\diamond^6 a) = a\diamond^6 a$. Then note that $c_{n_2-6}(b\diamond^{n_1} b\diamond^{n_2} b) = c_{n_2-6}(b\diamond^{n_1} b\diamond^{n_2-n_1-8}\diamond^{n_1+6+2} b) = b\diamond^{n_1} b\diamond^6 b$, since $n_2 - n_1 - 8 > 6$, $|b\diamond^{n_1} b\diamond^6 b| = n_1 + 9 < 6 + 9 = 15 < n_2 - 6$ and $|b\diamond^{n_1} b\diamond^{n_2-n_1-8}| = n_2 - 6$. Then let x_1, x_2 be the nonnegative integers so that $14x_1 + 15x_2 = n_2 - 6$ (such x_1 and x_2 exist by Lemma 2.46). Therefore if $v = (a^7b^7)^{x_1}(a^7b^8)^{x_2}$ and $w = v^{\mathbb{Z}}$, then $|v| = x_1 14 + x_2 15 = n_2 - 6$ and w avoids $c_{n_2-6}(X_{m|n_1,n_2})$ by Theorem 2.15. Furthermore, w avoids $X_{m|n_1,n_2}$ by Lemma 2.12.

Now, note that if $n_1 = 2m - 2 = 10$, then since $X_{6|10}$ is avoidable, so is $X_{6|10,n_2}$. Finally consider the case that $n_1 = 2m - 1 = 11$. Assume that $n_2 > 31$. Then note that $c_{n_2+7}(X_{6|11,n_2}) = \{a\diamond^6 a, b\diamond^5 b\diamond^5 b\}$. To see this, first note that since $n_2+7 > 8$ it follows that $c_{n_2+7}(a\diamond^6 a) = a\diamond^6 a$. Similarly, $c_{n_2+7}(b\diamond^{n_1} b\diamond^{n_2} b) = c_{n_2+7}(b\diamond^{11} b\diamond^{n_2} b) = c_{n_2+7}(b\diamond^{11} b\diamond^{n_2-6}\diamond^6 b) = b\diamond^5 b\diamond^5 b$, where the last equality holds since $13 < n_2 + 7$ and $11 + 2 + n_2 - 6 = n_2 + 7$. Then since $n_2 + 7 > 38$, by Lemma 2.47 there exist nonnegative integers $x_1, \ldots, x_6$ so that $n_2 + 7 = 13x_1 + 14x_2 + \cdots + 18x_6$. Then set $v = (a^6b^7)^{x_1}(a^6b^8)^{x_2} \cdots (a^6b^{12})^{x_6}$. It follows that $|v| = n_2+7$. Furthermore $w = v^{\mathbb{Z}}$ avoids $c_{n_2+7}(X_{6|11,n_2})$ by Theorem 2.15, and it follows by Lemma 2.12 that w avoids $X_{6|11,n_2}$. Therefore, all we have to do at this point is verify the claim when $n_1 < 12$ and $n_2 < 188$, which can be done using brute force to find avoiding words (despite the fact that some of these values of n_2 are large, this calculation only takes a short time). $\square$

We now have the tools to prove Conjectures 2.9 and 2.10.

Theorem 2.49. *Conjectures 2.9 and 2.10 are true.*

Proof. Let m, n_1, n_2 be nonnegative integers such that $n_1 \leq n_2$, $\gcd(m+1, n_1+1, n_2+1) = 1$, and $(m, n_1, n_2) \neq (6, 1, 3)$. If $n_1 + 1 \equiv 0 \bmod 2m + 2$ or $n_2 + 1 \equiv 0 \bmod 2m + 2$ then by Lemma 2.45 $X_{m|n_1,n_2}$ is avoidable. If $2m \equiv n_1 + n_2 \bmod 2m + 2$ and $2m \neq n_1 + n_2$, then by Lemma 2.44, $X_{m|n_1,n_2}$ is avoidable. If $m \equiv n_2 - n_1 - 1 \bmod 2m + 2$ and $m \neq n_2 - n_1 - 1$, then by Lemma 2.42 $X_{m|n_1,n_2}$ is avoidable. If ($m \equiv 2n_1 + n_2 + 2 \bmod 2m + 2$ and $m \neq 2n_1 + n_2 + 2$) or ($m \equiv 2n_2 + n_1 + 2 \bmod 2m + 2$ and $m \neq 2n_2 + n_1 + 2$), then by Lemma 2.43 $X_{m|n_1,n_2}$ is avoidable. If $m = 6$ and $c_{14}(X_{6|n_1,n_2}) = X_{6|n_1',n_2'}$ where $n_1' \neq n_2'$ and $n_1', n_2' \in \{1, 3, 7\}$, then by Lemma 2.48, $X_{m|n_1,n_2}$ is avoidable. Finally, in all other cases, it follows by Theorem 2.40 that $Y_{m|n_1,n_2}$, and thus $X_{m|n_1,n_2}$, is avoidable. Therefore, the claim follows. $\square$

2.6 The Classification

In this section, we complete the characterization of all two-element unavoidable sets of partial words over any arbitrary alphabet.

Proposition 4 in [4] states that if Conjecture 2.9 is true (or equivalently Conjecture 2.10 is true, which we showed true in Section 2.5), then all $X_{m_1,\dots,m_k|n_1,\dots,n_l}$ where $k = 1$ and $l \geq 3$ are avoidable. The proof is based on at least one of the four sets $X_{m|n_1,n_2}$, $X_{m|n_2,n_3}$, $X_{m|n_1+n_2+1,n_3}$, $X_{m|n_1,n_2+n_3+1}$ being avoidable. The proof claims that it is not possible for all four sets to meet even the length requirements of Conjecture 2.10, and hence at least one must be avoidable. However, using a computer algebra system, we have identified exactly twelve cases, when $k = 1$ and $l = 3$, where the length requirements are satisfied for all four sets. In most of these cases, further investigation has determined that while the length requirements of Conjecture 2.10 are satisfied, the other constraints of Conjecture 2.10 fail, and hence at least one of the four sets is avoidable. Theorem 2.51 will identify the cases where all four sets are unavoidable for $k = 1$ and $l = 3$.

Theorem 2.50. *Let m, n_1, n_2, n_3 be nonnegative integers such that $n_3 > m$. Define $d_3 = n_3 - m$ and $d_2 = d_3 + n_2 + 1$ and note that $0 < d_3 < d_2$. If the conditions (i) $2(m+2) - 1 = n_1 + n_2 + n_3 + 4$, (ii) $d_2 \mid m+1$, and (iii) $d_2 = 2d_3$ hold, then $X_{m|n_1,n_2,n_3}$ is unavoidable. Note that Condition (iii) implies $n_3 - m = n_2 + 1$, so $d_2 = 2(n_2 + 1)$.*

Proof. Let $x_a = a\diamond^m a$ and $x_b = b\diamond^{n_1} b\diamond^{n_2} b\diamond^{n_3} b$, hence $X_{m|n_1,n_2,n_3} = \{x_a, x_b\}$. We wish to attempt to construct a two-sided infinite word w such that w avoids x_a and x_b and reach a contradiction, thus proving $X_{m|n_1,n_2,n_3}$ is unavoidable. Note that by Conditions (ii) and (iii), $d_3 \mid d_2$ and $d_3 \mid m+1$.

If for any index i, $w(i) = a$, then $w(i+m+1) = b$ and $w(i-m-1) = b$ to avoid x_a. This situation is depicted in the table

$w(i-m-1)$	$\cdots$	$w(i-d_2)$	$w(i-d_3)$	$w(i)$	$\cdots$	$w(i+m+1)$
b		x	x	a		b

where these b's line up exactly with x_b. By Condition (i), in order to avoid x_b, at least one of $w(i-d_3) = a$ or $w(i-d_2) = a$. Here x's are placeholders denoting this. Note that adjacent letters in the table are distance d_3 apart. Observe that for $i \equiv k \bmod d_3$, $i - d_3 \equiv k \bmod d_3$ and $i - d_2 \equiv k \bmod d_3$, hence inductively for all integers $j < i$ where $j \equiv k \bmod d_3$, $w(j) = a$ or $w(j - d_3) = a$.

Suppose for any index i where $w(i) = a$ and $w(i+m+1) = w(i-(m+1)) = b$, we let $w(i-d_2) = a$. Then inductively $w(i-kd_2) = a$ for all integers $k > 0$. But $d_2 \mid m+1$ by Condition (ii), so $w(i-(m+1)) = a$, a contradiction. Therefore $w(i) = a$ and $w(i-d_3) = a$ for some i, and as a result $w(i-(m+1)) = b$ and $w(i-(m+1)-d_3) = b$. Consider $i = 0$, so $w(0) = a$ and $w(-d_3) = a$, and consequently $w(-(m+1)) = b$ and $w(-(m+1)-d_3) = b$. By our previous observation, for all $j < 0$ where $j \equiv 0 \bmod d_3$, $w(j) = a$ or $w(j-d_3) = a$. But $d_3 \mid -(m+1)$, and neither $w(-(m+1)) = a$ nor $w(-(m+1)-d_3) = a$. Therefore we have reached a contradiction, thereby proving $X_{m|n_1,n_2,n_3}$ is unavoidable. □

Theorem 2.51. *The set $X_{m|n_1,n_2,n_3}$ is unavoidable by Theorem 2.50 if and only if the sets $X_{m|n_1,n_2}$, $X_{m|n_2,n_3}$, $X_{m|n_1+n_2+1,n_3}$, and $X_{m|n_1,n_2+n_3+1}$ are unavoidable.*

Proof. Let m, n_1, n_2, and n_3 be nonnegative integers. We first show that the restrictions on these variables in Theorem 2.50 are equivalent to $m = (2k+4)(n_2+1)-1$, $n_1 = (2k+2)(n_2+1)-1$ and $n_3 = (2k+5)(n_2+1)-1$, for any integer $k \geq 0$. Suppose m, n_1 and n_3 are as defined in the set of equations above. Observe that $n_3 > m$ and

$$2m = (4k+8)(n_2+1) - 2 = n_1 + n_2 + n_3 + 1$$
$$d_3 = n_3 - m = n_2 + 1$$
$$d_2 = d_3 + n_2 + 1 = 2(n_2+1) = 2d_3 \mid m+1 = 2(k+2)(n_2+1)$$

Thus the equations satisfy the restrictions of Theorem 2.50. Now suppose m, n_1, n_2, and n_3 meet the conditions of Theorem 2.50. Then $2(m+2)-1 = n_1+n_2+n_3+4$, $d_3 = n_3-m$ and $d_2 = d_3+n_2+1$. Moreover $d_2 = d_3+n_2+1 = 2d_3$, so $n_3 - m = n_2 + 1$. Lastly, $d_2 = 2(n_3-m) = 2(n_2+1) \mid m+1$, hence $2k(n_2+1) = m+1$ for some integer k. Solving these equations for m:

$$2m = n_1 + n_2 + n_3 + 1 \tag{2.16}$$
$$m = n_3 - n_2 - 1 \tag{2.17}$$
$$m = 2k(n_2+1) - 1 \tag{2.18}$$

We claim that $k \geq 2$. Note that if $k = 0$, then $m = -1$ by Equation (2.18), a contradiction. If $k = 1$, then $m = 2n_2 + 1$ by Equation (2.18). But $3n_2 = n_1 + n_3 - 1$ by Equation (2.17), and $3n_2 = n_3 - 2$ by Equation (2.17). Consequently, $n_1 = -1$, which is a contradiction.

Thus by Equations (2.17) and (2.18), $n_3 = 2k(n_2+1)-1+n_2+1 = (2k+1)(n_2+1)-1$. As a result, by Equations (2.16) and (2.18),

$$\begin{aligned} n_1 &= 2(2k(n_2+1)-1)-(n_2+1)-n_3 \\ &= 4k(n_2+1)-2-(n_2+1)-((2k+1)(n_2+1)-1) \\ &= (4k-1)(n_2+1)-1-(2k+1)(n_2+1) \\ &= (2k-2)(n_2+1)-1 \end{aligned}$$

Substituting $k = k_0+2$ where $k_0 \geq 0$,

$$\begin{aligned} m &= (2k_0+4)(n_2+1)-1 \\ n_1 &= (2k_0+2)(n_2+1)-1 \\ n_3 &= (2k_0+5)(n_2+1)-1 \end{aligned}$$

Therefore the set of equations outlined in the hypothesis are equivalent to the restrictions of Theorem 2.50.

We now show that if $X_{m|n_1,n_2,n_3}$ is unavoidable by Theorem 2.50, then $X_{m|n_1,n_2}$, $X_{m|n_2,n_3}$, $X_{m|n_1+n_2+1,n_3}$, and $X_{m|n_1,n_2+n_3+1}$ are unavoidable. We claim $X_{m|n_2,n_1}$ is unavoidable by Proposition 2.5, and hence $X_{m|n_1,n_2}$ is unavoidable by symmetry. Observe that

$$2n_2+n_1+2 = 2n_2+(2k+2)(n_2+1)-1+2 = (2k+4)(n_2+1)-1 = m$$

Note that $n_2+1 \mid n_1+1$, and because $m+1 = 2(k+2)(n_2+1)$, by Theorem 2.3 $X_{m|n_2}$ is unavoidable. This suffices to show that by Proposition 2.5, $X_{m|n_2,n_1}$ is unavoidable.

We claim $X_{m|n_2,n_3}$ is unavoidable by Proposition 2.5. Observe that

$$n_3-n_2-1 = (2k+5)(n_2+1)-(n_2+1)-1 = (2k+4)(n_2+1)-1 = m$$

Since $n_2+1 \mid n_3+1$, and $X_{m|n_2}$ is unavoidable by Theorem 2.3, this suffices to show that by Proposition 2.5 $X_{m|n_2,n_3}$ is unavoidable.

Next we show that $X_{m|n_1+n_2+1,n_3}$ is unavoidable by Theorem 2.8. Here

$$\begin{aligned} n_1+n_2+n_3+1 &= (2k+2)(n_2+1)+(2k+5)(n_2+1)+(n_2+1)-2 \\ &= 2((2k+4)(n_2+1)-1) = 2m \end{aligned} \tag{2.19}$$

Let $d = m-(n_1+n_2+1) = n_2+1$, and note that $d \mid m+1$, hence by Theorem 2.8 $X_{m|n_1+n_2+1,n_3}$ is unavoidable.

We claim $X_{m|n_1,n_2+n_3+1}$ is unavoidable by Theorem 2.8. Note that Equation (2.19) satisfies one condition, so the only thing to be shown is that $m-n_1 \mid m+1$. But $m-n_1 = 2(n_2+1)$, and $m+1 = 2(k+2)(n_2+1)$, so indeed Theorem 2.8 applies. Thus $X_{m|n_1,n_2+n_3+1}$ is unavoidable.

Therefore if $X_{m|n_1,n_2,n_3}$ is unavoidable by Theorem 2.50, then the sets $X_{m|n_1,n_2}$, $X_{m|n_2,n_3}$, $X_{m|n_1+n_2+1,n_3}$ and $X_{m|n_1,n_2+n_3+1}$ are unavoidable.

We show that if $X_{m|n_1,n_2}$, $X_{m|n_2,n_3}$, $X_{m|n_1+n_2+1,n_3}$ and $X_{m|n_1,n_2+n_3+1}$ are all unavoidable, then $X_{m|n_1,n_2,n_3}$ is unavoidable by Theorem 2.50. We have determined by a computer algebra system that there are exactly 4 cases where all four resulting sets are unavoidable. However, we only have to consider 2 of the 4 cases, for the others are symmetric (swap n_1, n_3). One case is when $m = 4n_2 + 3, n_1 = 2n_2 + 1, n_3 = 5n_2 + 4$.

Let $m = 4n_2 + 3, n_1 = 2n_2 + 1, n_3 = 5n_2 + 4$, and define $d_3 = n_3 - m$ and $d_2 = d_3 + n_2 + 1$. Observe that

$$2(m + 2) - 1 = 8n_2 + 9 = n_1 + n_2 + n_3 + 4$$

and also that $d_3 = n_3 - m = n_2 + 1$, $d_2 = d_3 + n_2 + 1 = 2d_3$, and $d_2 = 2n_2 + 2$ divides $m + 1 = 4n_2 + 4$. It then follows directly from Theorem 2.50 that $X_{m|n_1,n_2,n_3}$ is unavoidable.

The other case is when $m \geq 2$, $n_1 = m - 2n_2 - 2$, and $n_3 = m + n_2 + 1$. To see the latter case, let $d_3 = n_3 - m$ and $d_2 = d_3 + n_2 + 1$. Observe that

$$n_1 + n_2 + n_3 + 1 = 2m \tag{2.20}$$

Recall that we are considering when $X_{m|n_1,n_2}$, $X_{m|n_2,n_3}$, $X_{m|n_1,n_2+n_3+1}$ and $X_{m|n_1+n_2+1,n_3}$ are all unavoidable. It follows from Equation (2.20) and Theorem 2.8 that $X_{m|n_1,n_2+n_3+1}$ and $X_{m|n_1+n_2+1,n_3}$ are unavoidable if and only if

$$m - n_1 = 2(n_2 + 1) \mid m + 1 \tag{2.21}$$

$$m - (n_1 + n_2 + 1) = n_2 + 1 \mid m + 1 \tag{2.22}$$

These equations, combined with Equation (2.20), satisfy the conditions of Theorem 2.50. Thus if Equations (2.21) and (2.22) hold, then $X_{m|n_1,n_2,n_3}$ is unavoidable. Therefore $X_{m|n_1,n_2,n_3}$ is unavoidable by Theorem 2.50 when $X_{m|n_1,n_2}$, $X_{m|n_2,n_3}$, $X_{m|n_1+n_2+1,n_3}$ and $X_{m|n_1,n_2+n_3+1}$ are all unavoidable. $\square$

The following theorem completes, as well as summarizes, the classification of all the sets $X_{m_1,\ldots,m_k|n_1,\ldots,n_l}$.

Theorem 2.52. *Let* $m_1, \ldots, m_k, n_1, \ldots, n_l$ *be nonnegative integers such that* $gcd(m_1 + 1, \ldots, m_k + 1, n_1 + 1, \ldots, n_l + 1) = 1$. *Assume that* $k \leq l$ *(the case* $l \leq k$ *is symmetric). Then* $X_{m_1,\ldots,m_k|n_1,\ldots,n_l}$ *is avoidable if and only if one of the following conditions hold:*

(1) $k = 1, l = 1$*: all such sets that are avoidable by Theorem 2.3.*

(2) $k = 1, l = 2$*: all such sets that satisfy* $n_1 \leq n_2$, $(m, n_1, n_2) \neq (6, 1, 3)$*, and one of (2.4)–(2.9) (the case* $n_1 \geq n_2$, $(m, n_1, n_2) \neq (6, 3, 1)$ *is symmetric).*

(3) $k = 1, l = 3$*: all such sets that are avoidable by Theorem 2.50.*

(4) $k = 1, l \geq 4$*: all such sets are avoidable.*

(5) $k \geq 2, l \geq 2$*: all such sets are avoidable.*

Proof. For Statement 1, see Theorem 2.3, while for Statement 2, see Theorem 2.49.

For Statement 3, let $X_{m|n_1,n_2,n_3}$ be unavoidable. Then the sets $X_{m|n_1,n_2}$, $X_{m|n_2,n_3}$, $X_{m|n_1+n_2+1,n_3}$ and $X_{m|n_1,n_2+n_3+1}$ are also unavoidable, and therefore by Theorem 2.51 $X_{m|n_1,n_2,n_3}$ is unavoidable by Theorem 2.50.

For Statement 4, it is enough to prove that $X_{m|n_1,n_2,n_3,n_4}$ is avoidable for all nonnegative integers m, n_1, n_2, n_3, n_4. Suppose for contradiction that $X_{m|n_1,n_2,n_3,n_4}$ is unavoidable. Then the sets $X_{m|n_1,n_2,n_3}$ and $X_{m|n_1+n_2+1,n_3,n_4}$ are unavoidable. By Statement 3, these sets must be unavoidable by Theorem 2.50, hence

$$\begin{aligned} 2(m+2) - 1 &= n_1 + n_2 + n_3 + 4 \\ 2(m+2) - 1 &= n_1 + n_2 + n_3 + 4 + n_4 + 1 \end{aligned}$$

Consequently $n_4 = -1$, a contradiction, so $X_{m|n_1,n_2,n_3,n_4}$ must be avoidable.

Statement 5 was shown in [4]. □

We have thus completed the classification of all two-element sets over any arbitrary alphabet.

2.7 Conclusion

A World Wide Web server interface has been established at

`www.uncg.edu/cmp/research/unavoidablesets3`

for automated use of a program that given as input a finite set of partial words over a given alphabet will classify the set as avoidable or unavoidable, and will output the shortest period of an infinite avoiding word in case the set is avoidable.

A World Wide Web server interface has also been established at

`www.uncg.edu/cmp/research/unavoidablesets4` for automated use of a program that provides a means of investigating the properties of $Y_{m|n_1,n_2}$. Given values of m, n_1, n_2 as input, the program determines whether $Y_{m|n_1,n_2}$ is avoidable or not, and then outputs the graph $H_{m|n_1,n_2}$. Furthermore if $Y_{m|n_1,n_2}$ is avoidable, then the program colors the graph in such a way that it avoids $Y_{m|n_1,n_2}$, using the colors red and blue to represent a and b (or vice versa).

Acknowledgments

This material is based upon work supported by the National Science Foundation under Grant No. DMS–0754154. The Department of Defense is also gratefully acknowledged. We thank the referees of a preliminary version of this chapter for their very valuable comments and suggestions.

References

1. Berstel, J. and Boasson, L. (1999). Partial words and a theorem of Fine and Wilf, *Theoretical Computer Science* **218**, pp. 135–141.
2. Blakeley, B., Blanchet-Sadri, F., Gunter, J. and Rampersad, N. (2009). On the complexity of deciding avoidability of sets of partial words, in V. Diekert and D. Nowotka (eds.), *DLT 2009, 13th International Conference on Developments in Language Theory, Stuttgart, Germany*, *Lecture Notes in Computer Science*, Vol. 5583 (Springer-Verlag, Berlin, Heidelberg), pp. 113–124, `www.uncg.edu/cmp/research/unavoidablesets3`.
3. Blanchet-Sadri, F. (2008). *Algorithmic Combinatorics on Partial Words* (Chapman & Hall/CRC Press, Boca Raton, FL).
4. Blanchet-Sadri, F., Brownstein, N. C., Kalcic, A., Palumbo, J. and Weyand, T. (2009a). Unavoidable sets of partial words, *Theory of Computing Systems* **45**, 2, pp. 381–406, `www.uncg.edu/cmp/research/unavoidablesets2`.
5. Blanchet-Sadri, F., Jungers, R. M. and Palumbo, J. (2009b). Testing avoidability on sets of partial words is hard, *Theoretical Computer Science* **410**, pp. 968–972.
6. Champarnaud, J. M., Hansel, G. and Perrin, D. (2004). Unavoidable sets of constant length, *International Journal of Algebra and Computation* **14**, pp. 241–251.
7. Choffrut, C. and Culik II, K. (1984). On extendibility of unavoidable sets, *Discrete Applied Mathematics* **9**, pp. 125–137.
8. Choffrut, C. and Karhumäki, J. (1997). Combinatorics of Words, in G. Rozenberg and A. Salomaa (eds.), *Handbook of Formal Languages*, Vol. 1 (Springer-Verlag, Berlin), pp. 329–438.
9. Crochemore, M., Le Rest, M. and Wender, P. (1983). An optimal test on

finite unavoidable sets of words, *Information Processing Letters* **16**, pp. 179–180.

10. Ehrenfeucht, A., Haussler, D. and Rozenberg, G. (1983). On regularity of context-free languages, *Theoretical Computer Science* **27**, pp. 311–332.
11. Evdokimov, A. and Kitaev, S. (2004). Crucial words and the complexity of some extremal problems for sets of prohibited words, *Journal of Combinatorial Theory, Series A* **105**, pp. 273–289.
12. Fraleigh, J. B. (2003). *A First Course in Abstract Algebra* (Addison-Wesley, Reading, MA).
13. Higgins, P. M. and Saker, C. J. (2006). Unavoidable sets, *Theoretical Computer Science* **359**, pp. 231–238.
14. Lothaire, M. (2002). *Algebraic Combinatorics on Words* (Cambridge University Press, Cambridge).
15. Rosaz, L. (1995). Unavoidable languages, cuts and innocent sets of words, *RAIRO-Theoretical Informatics and Applications* **29**, pp. 339–382.
16. Rosaz, L. (1998). Inventories of unavoidable languages and the word-extension conjecture, *Theoretical Computer Science* **201**, pp. 151–170.
17. Saker, C. J. and Higgins, P. M. (2002). Unavoidable sets of words of uniform length, *Information and Computation* **173**, pp. 222–226.
18. Smyth, W. F. (2003). *Computing Patterns in Strings* (Pearson Addison-Wesley).

This page is intentionally left blank

Chapter 3

On Glushkov $\mathbb{K}$-graphs

Pascal Caron
LITIS, Université de Rouen, 76801 Saint Étienne du Rouvray, France,
E-mail: `pascal.caron@univ-rouen.fr`

Marianne Flouret
LITIS, Université du Havre, 76058 Le Havre Cedex, France

3.1 Introduction

The extension of boolean algorithms (over languages) to multiplicities (over series) has always been a central point in theoretical research. First, Schützenberger [16] has given an equivalence between rational and recognizable series extending the classical result of Kleene [12]. Recent contributions have been done in this area, an overview of knowledge of these domains is presented by Sakarovitch in [15]. Many research works have focused on producing a small WFA . For example, Caron and Flouret have extended the Glushkov construction to WFAs [4]. Champarnaud *et al* have designed a quadratic algorithm [7] for computing the equation WFA of a $\mathbb{K}$-expression. This equation WFA has been introduced by Lombardy and Sakarovitch as an extension of Antimirov's algorithm [13] based on partial derivatives.

Moreover, the Glushkov WFA of a $\mathbb{K}$-expression with n occurrences of symbol (we say that its alphabetic width is equal to n) has only $n+1$ states; the equation $\mathbb{K}$-automaton (that is a quotient of the Glushkov automaton) has at most $n+1$ states.

On the opposite, classical algorithms compute $\mathbb{K}$-expressions the size of which is exponential with respect to the number of states of the WFA . For example, let us cite the block decomposition algorithm proven in [1].

In this paper, we also address the problem of computing short $\mathbb{K}$-expressions, and we focus on a specific kind of conversion based on Glushkov automata. Actually the particularity of Glushkov automata is the following: any regular expression of width n can be turned into its Glushkov $(n+1)$-state automaton; if a $(n+1)$-state automaton is a Glushkov one, then it can be turned into an expression of width n. The latter property is based on the characterization of the family of Glushkov automata in terms of graph properties presented in [6]. These properties are stability, transversality and reducibility. Brüggemann-Klein defines regular expressions in *Star Normal Form* (SNF) [2]. These expressions are characterized by underlying Glushkov automata where each edge is generated exactly one time. This definition is extended to multiplicities. The study of the SNF case would not be necessary if all $\mathbb{K}$-expressions were equivalent to some in SNF with the same litteral length, as it is the case for the boolean semiring $\mathbb{B}$.

The aim of this paper is to extend the characterization of Glushkov automata to the multiplicity case in order to compute a $\mathbb{K}$-expression of width n from a $(n+1)$-state WFA . This extension requires to restrict the work to factorial semirings as well as Star Normal Form $\mathbb{K}$-expressions.

We exhibit a procedure that, given a WFA M on $\mathbb{K}$ a factorial semiring, outputs the following: either M is obtained by the Glushkov algorithm from a proper $\mathbb{K}$-expression E in Star Normal Form and the procedure computes a $\mathbb{K}$-expression F equivalent to E, or M is not obtained in that way and the procedure says no.

The following section recalls fundamental notions concerning automata, expressions and Glushkov conversion for both boolean and multiplicity cases. An error in the paper by Caron and Ziadi [6] is pointed out and corrected. The section 3 is devoted to the reduction rules for acyclic $\mathbb{K}$-graphs. Their efficiency is provided by the confluence of $\mathbb{K}$-rules. The next section gives orbit properties for Glushkov $\mathbb{K}$-graphs. The section 5 presents the algorithms computing a $\mathbb{K}$-expression from a Glushkov $\mathbb{K}$-graph and details an example.

3.2 Definitions

3.2.1 *Classical Notions*

Let Σ be a finite set of letters (alphabet), ε the empty word and $\emptyset$ the empty set. Let $(\mathbb{K}, \oplus, \otimes)$ be a zero-divisor free semiring where $\overline{0}$ is the neutral element of $(\mathbb{K}, \oplus)$ and $\overline{1}$ the one of $(\mathbb{K}, \otimes)$. The semiring $\mathbb{K}$ is said to be zero-divisor free [10] if $\overline{0} \neq \overline{1}$ and if $\forall x, y \in \mathbb{K}$, $x \otimes y = \overline{0} \Rightarrow x = \overline{0}$ or $y = \overline{0}$.

A *formal series* [1] is a mapping S from Σ^* into $\mathbb{K}$ usually denoted by $S = \sum_{w \in \Sigma^*} S(w)w$ where $S(w) \in \mathbb{K}$ is the coefficient of w in S. The *support* of S is the language $Supp(S) = \{w \in \Sigma^* | S(w) \neq \overline{0}\}$.

In [13], Lombardy and Sakarovitch explain in details the computation of $\mathbb{K}$- expressions. We have followed their model of grammar. Our constant symbols are ε the empty word and $\emptyset$. Binary rational operations are still $+$ and $\cdot$, the unary ones are Kleene closure $*$, positive closure $^+$ and for every $k \in \mathbb{K}$, the multiplication to the left or to the right of an expression $\times$. For an easier reading, we will write kE (respectively Ek) for $k \times E$ (respectively $E \times k$). Notice that our definition of $\mathbb{K}$-expressions, which set is denoted $E_{\mathbb{K}}$, introduces the operator of positive closure. This operator preserves rationality with the same conditions (see below) that the Kleene closure's one.

$\mathbb{K}$-expressions are then given by the following grammar:

$$E \to a \in \Sigma \mid \emptyset \mid \varepsilon \mid (E+E) \mid (E \cdot E) \mid (E^*) \mid (E^+) \mid (kE), k \in \mathbb{K} \mid (Ek), k \in \mathbb{K}$$

Notice that parenthesis will be omitted when not necessary. The expressions E^+ and E^* are called *closure expressions.* If a series S is represented by a $\mathbb{K}$-expression E, then we denote by $c(S)$ (or $c(E)$) the coefficient of the empty word of S. A $\mathbb{K}$-expression E is *valid* [15] if for each closure subexpression F^* and F^+ of E, $\sum_{i=0}^{+\infty} c(F) \in \mathbb{K}$.

A $\mathbb{K}$-expression E is *proper* if for each closure subexpression F^* and F^+ of E, $c(F) = \overline{0}$.

We denote by $\mathcal{E}_{\mathbb{K}}$ the set of proper $\mathbb{K}$-expressions. Rational series can then be defined as formal series expressed by *proper* $\mathbb{K}$-*expressions.* For E in $\mathcal{E}_{\mathbb{K}}$, $Supp(E)$ is the support of the rational series defined by E.

The length of a $\mathbb{K}$-expression E, denoted by $||E||$, is the number of occurences of letters and of ε appearing in E. By opposition, the litteral length, denoted by $|E|$ is the number of occurences of letters in E. For example, the expression $E = (a+3)(b+2)+(-1)$ as a length of 5 and a litteral length of 2.

A *weighted finite automaton* (*WFA*) on a zero-divisor free semiring $\mathbb{K}$ over an alphabet Σ [8] is a 5-tuple $(\Sigma, Q, I, F, \delta)$ where Q is a finite set of states and the sets I, F and δ are mappings $I : Q \to \mathbb{K}$ (input weights), $F : Q \to \mathbb{K}$ (output weights), and $\delta : Q \times \Sigma \times Q \to \mathbb{K}$ (transition weights). The set of WFAs on $\mathbb{K}$ is denoted by $\mathcal{M}_{\mathbb{K}}$. A WFA is *homogeneous* if all vertices reaching a same state are labeled by the same letter.

A $\mathbb{K}$-*graph* is a graph $G = (X, U)$ labeled with coefficients in $\mathbb{K}$ where X is the set of vertices and $U : X \times X \to \mathbb{K}$ is the function that associates each edge with its label in $\mathbb{K}$. When there is no edge from p to q, we have $U(p, q) = \overline{0}$. In case $\mathbb{K} = \mathbb{B}$, the boolean semiring, $E_{\mathbb{B}}$ is the set of *regular expressions* and, as the only element of $\mathbb{K} \setminus \overline{0}$ is $\overline{1}$, we omit the use of coefficient and of the external product ($\overline{1}a = a\overline{1} = a$). For a rational series S represented by $E \in E_{\mathbb{B}}$, $Supp(E)$ is usually called the language of E, denoted by $L(E)$ and $S = Supp(S) = L(E)$. A boolean automaton (automaton in the sequel) M over an alphabet Σ is usually defined [8, 11] as a 5-tuple $(\Sigma, Q, I, F, \delta)$ where Q is a finite set of states, $I \subseteq Q$ the set of initial states, $F \subseteq Q$ the set of final states, and $\delta \subseteq Q \times \Sigma \times Q$ the set of edges. We denote by $L(M)$ the language recognized by the automaton M. A graph $G = (X, U)$ is a $\mathbb{B}$-graph for which labels of edges are not written.

3.2.2 *Extended Glushkov Construction*

An algorithm given by Glushkov [9] for computing an automaton with $n+1$ states from a regular expression of litteral length n has been extended to semirings $\mathbb{K}$ by the authors [4]. Informally, the principle is to associate exactly one state in the computed automaton to each occurrence of letters in the expression. Then, we link by a transition two states of the automaton if the two occurences of the corresponding letters in the expression can be read successively.

In order to recall the extended Glushkov construction, we have to first define the ordered pairs and the supported operations. An ordered pair (l, i) consists of a coefficient $l \in \mathbb{K} \setminus \{\overline{0}\}$ and a position $i \in \mathbb{N}$. We also define the functions $\mathcal{I}_H : H \to \mathbb{K}$ such that $\mathcal{I}_H(i)$ is equal to $\overline{1}$ if $i \in H$ and $\overline{0}$ otherwise. We define $P : 2^{\mathbb{K}\setminus\{\overline{0}\}\times\mathbb{N}} \to 2^{\mathbb{N}}$ the function that extracts positions from a set of ordered pairs as follows: for Y a set of ordered pairs, $P(Y) = \{i_j, 1 \leq j \leq |Y| \mid \exists (l_j, i_j) \in Y\}$.

The function $Coeff_Y : P(Y) \to \mathbb{K} \setminus \{\overline{0}\}$ extracts the coefficient associated to a position i as follows: $\mathit{Coeff}_Y(i) = l$ for $(l,i) \in Y$.

Let $Y, Z \subset \mathbb{K} \setminus \{\overline{0}\} \times \mathbb{N}$ be two sets of ordered pairs. We define the product of $k \in \mathbb{K} \setminus \overline{0}$ and Y by $k\cdot Y = \{(k \otimes l, i) \mid (l,i) \in Y\}$ and $Y\cdot k = \{(l \otimes k, i) \mid (l,i) \in Y\}$, $\overline{0}\cdot Y = Y\cdot \overline{0} = \emptyset$. We define the operation $\uplus$ by $Y \uplus Z = \{(l,i) \mid$ either $(l,i) \in Y$ and $i \notin P(Z)$ or $(l,i) \in Z$ and $i \notin P(Y)$ or $(l_s,i) \in Y, (l_t,i) \in Z$ for some $l_s, l_t \in \mathbb{K}$ with $l = l_s \oplus l_t \neq \overline{0}\}$.

As in the original Glushkov construction [9, 14], and in order to specify their position in the expression, letters are subscripted following the order of reading. The resulting expression is denoted $\overline{E}$, defined over the alphabet of indexed symbols $\overline{\Sigma}$, each one appearing at most once in $\overline{E}$. The set of indices thus obtained is called positions and denoted by $\mathit{Pos}(E)$. For example, starting from $E = (2a+b)^*\cdot a\cdot 3b$, one obtains the indexed expression $\overline{E} = (2a_1 + b_2)^*\cdot a_3\cdot 3b_4$, $\overline{\Sigma} = \{a_1, b_2, a_3, b_4\}$ and $\mathit{Pos}(E) = \{1,2,3,4\}$. Four functions are defined in order to compute a WFA which needs not be deterministic. $\mathit{First}(E)$ represents the set of initial positions of words of $\mathit{Supp}(\overline{E})$ associated with their input weight, $\mathit{Last}(E)$ represents the set of final positions of words of $\mathit{Supp}(\overline{E})$ associated to their output weight and $\mathit{Follow}(E,i)$ is the set of positions of words of $\mathit{Supp}(\overline{E})$ which immediately follows position i in the expression $\overline{E}$, associated to their transition weight. In the boolean case, these sets are subsets of $\mathit{Pos}(E)$. The $\mathit{Null}(E)$ set represents the coefficient of the empty word. The way to compute these sets is completely formalized in table 3.1.

These functions allow us to define the WFA $\overline{M} = (\overline{\Sigma}, Q, \{s_I\}, F, \overline{\delta})$ where

(1) $\overline{\Sigma}$ is the indexed alphabet,
(2) s_I is the single initial state with no incoming edge with $\overline{1}$ as input weight,
(3) $Q = \mathit{Pos}(E) \cup \{s_I\}$
(4) $F : Q \to \mathbb{K}$ such that $F(i) = \begin{cases} \mathit{Null}(E) & \text{if } i = s_I \\ \mathit{Coeff}_{\mathit{Last}(E)}(i) & \text{otherwise} \end{cases}$
(5) $\delta : Q \times \overline{\Sigma} \times Q \to \mathbb{K}$ such that $\delta(i, a_j, h) = \overline{0}$ for every $h \neq j$, whereas $\delta(i, a_j, j) = \begin{cases} \mathit{Coeff}_{\mathit{First}(E)}(j) & i = s_I \\ \mathit{Coeff}_{\mathit{Follow}(E,i)}(j) & i \neq s_I \end{cases}$

The Glushkov WFA $M = (\Sigma, Q, \{s_I\}, F, \delta)$ of E is computed from $\overline{M}$ by replacing the indexed letters on edges by the corresponding letters in the expression E. We will denote $A_{\mathbb{K}} : \mathcal{E}_{\mathbb{K}} \to \mathcal{M}_{\mathbb{K}}$ the application such that $A_{\mathbb{K}}(E)$ is the Glushkov WFA obtained from E by this algorithm proved in [4].

Table 3.1 Extended Glushkov functions.

E	**Null(E)**	**First(E)**	**Last(E)**	**Follow(E,i)**
$\emptyset$	$\overline{0}$	$\emptyset$	$\emptyset$	$\emptyset$
ε	$\overline{1}$	$\emptyset$	$\emptyset$	$\emptyset$
a_j	$\overline{0}$	$\{(\overline{1}, j)\}$	$\{(\overline{1}, j)\}$	$\emptyset$
kF	$k \otimes \mathit{Null}(F)$	$k \cdot \mathit{First}(F)$	$\mathit{Last}(F)$	$\mathit{Follow}(F, i)$
Fk	$\mathit{Null}(F) \otimes k$	$\mathit{First}(F)$	$\mathit{Last}(F) \cdot k$	$\mathit{Follow}(F, i)$
$F + G$	$\mathit{Null}(F) \oplus \mathit{Null}(G)$	$\mathit{First}(F) \uplus \mathit{First}(G)$	$\mathit{Last}(F) \uplus \mathit{Last}(G)$	$\mathcal{I}_{Pos(F)}(i) \cdot \mathit{Follow}(F, i) \uplus \mathcal{I}_{Pos(G)}(i) \cdot \mathit{Follow}(G, i)$
$F \cdot G$	$\mathit{Null}(F) \otimes \mathit{Null}(G)$	$\mathit{First}(F) \uplus \mathit{Null}(F) \cdot \mathit{First}(G)$	$\mathit{Last}(F) \cdot \mathit{Null}(G) \uplus \mathit{Last}(G)$	$\mathcal{I}_{Pos(F)}(i) \cdot \mathit{Follow}(F, i) \uplus \mathcal{I}_{Pos(G)}(i) \cdot \mathit{Follow}(G, i) \uplus \mathit{Coeff}_{Last(F)}(i) \cdot \mathit{First}(G)$
F^+	$\overline{0}$	$\mathit{First}(F)$	$\mathit{Last}(F)$	$\mathit{Follow}(F, i) \uplus \mathit{Coeff}_{Last(F)}(i) \cdot \mathit{First}(F)$
F^*	$\overline{1}$	$\mathit{First}(F)$	$\mathit{Last}(F)$	$\mathit{Follow}(F, i) \uplus \mathit{Coeff}_{Last(F)}(i) \cdot \mathit{First}(F)$

In order to compute a $\mathbb{K}$-graph from an homogeneous WFA M, we have to

- add a new vertex $\{\Phi\}$. Then U, the set of edges, is obtained from transitions of M by removing labels and adding directed edges from every final state to $\{\Phi\}$. We label edges to Φ with output weights of final states.
- $\otimes$-multiplied by $I(i)$, the label of the edge $U(i, p)$ for each $i \in Q$ such that $I(i) \neq \overline{0}$, for $p \in Q \cup \{\Phi\}$.

In case M is a Glushkov WFA of a $\mathbb{K}$-expression E, the $\mathbb{K}$-graph obtained from M is called Glushkov $\mathbb{K}$-graph of E and is denoted by $G_{\mathbb{K}}(E)$.

3.2.3 *Normal Forms and Casting Operation*

Star Normal Form and Epsilon Normal Form

For the boolean case, Brüggemann-Klein defines regular expressions in *Star Normal Form* (SNF) [2] as expressions E for which, for each position i of $\mathit{Pos}(E)$, when computing the $\mathit{Follow}(E, i)$ function, the unions of sets are disjoint. This definition is given only for usual operators $+$, $\cdot$, $*$. We can extend this definition to the positive closure, $^+$ as follows:

Definition 3.1. A $\mathbb{B}$-expression E is in SNF if, for each closure $\mathbb{B}$-subexpression H^* or H^+, the SNF conditions (1) $\mathit{Follow}(H, \mathit{Last}(H)) \cap \mathit{First}(H) = \emptyset$ and (2) $\varepsilon \notin L(H)$ hold.

Then, the properties of the star normal form (defined with the positive closure) are preserved.

In the same paper, Brüggemann-Klein defines also the *epsilon normal form* for the boolean case. We extend this epsilon normal form to the positive closure operator.

Definition 3.2. The epsilon normal form for a $\mathbb{B}$-expression E is defined by induction in the following way:

- $[E = \varepsilon$ or $E = a]$ E is in epsilon normal form.
- $[E = F + G]$ E is in epsilon normal form if F and G are in epsilon normal form and if $\varepsilon \notin L(F) \cap L(G)$.
- $[E = FG]$ E is in epsilon normal form if F and G are in epsilon normal form.
- $[E = F^+$ or $E = F^*]$ E is in epsilon normal form if F is in epsilon normal form and $\varepsilon \notin L(F)$.

Theorem 3.3 ([2])**.** *For each regular expression E, there exists a regular expression $E^\bullet$ such that*

(1) $A_{\mathbb{B}}(E) = A_{\mathbb{B}}(E^\bullet)$,
(2) $E^\bullet$ is in SNF,
(3) $E^\bullet$ can be computed from E in linear time.

Brüggemann-Klein has given every step for the computation of $E^\bullet$. This computation remains. We just have to add for H^+ the same rules as for H^*. Main steps of the proof are similar.

We extend the star normal form to multiplicities in this way. Let E be a $\mathbb{K}$-expression. For every subexpression H^* or H^+ in E, for each x in $P(\mathit{Last}(H))$,

$$P(\mathit{Follow}(H, x)) \cap P(\mathit{First}(H)) = \emptyset$$

We do not have to consider the case of the empty word because H^+ and H^* are proper $\mathbb{K}$-expressions if $c(H) = 0$.

As an example, let $\overline{H} = 2a_1^+ + (3b_2)^+$ and $\overline{E} = (\overline{H})^*$. We can see that the expression $\overline{E} = (2a_1^+ + (3b_2)^+)^*$ is not in SNF, because $2 \in P(Last(H))$ and $2 \in$ *P(Follow(H,2))*$\cap P(First(H))$.

The Casting Operation $\sim$

We have to define the casting $\sim$: $\mathcal{M}_{\mathbb{K}} \to \mathcal{M}_{\mathbb{B}}$. This is similar to the way in which Buchsbaum et al. [3] define the topology of a graph. A WFA $M = (\Sigma, Q, I, F, \delta)$ is casted into an automaton $\widetilde{M} = (\Sigma, Q, \widetilde{I}, \widetilde{F}, \widetilde{\delta})$ in the following way: $\widetilde{I}, \widetilde{F} \subset Q$, $\widetilde{I} = \{q \in Q \mid I(q) \neq \overline{0}\}$, $\widetilde{F} = \{q \in Q \mid F(q) \neq \overline{0}\}$ and $\widetilde{\delta} = \{(p,a,q) \mid p,q \in Q,\ a \in \Sigma \text{ and } \delta((p,a,q)) \neq \overline{0}\}$. The casting operation can be extended to $\mathbb{K}$-expressions $\sim$: $\mathcal{E}_{\mathbb{K}} \to E_{\mathbb{B}}$. The regular expression $\widetilde{E}$ is obtained from E by replacing each $k \in \mathbb{K} \setminus \overline{0}$ by $\overline{1}$. The $\sim$ operation on E is an embedding of $\mathbb{K}$-expressions into regular ones. Nevertheless, the Glushkov $\mathbb{B}$-graph computed from a $\mathbb{K}$-expression E may be different whether the Glushkov construction is applied first or the casting operation $\sim$. This is due to properties of $\mathbb{K}$-expressions. For example, let $\mathbb{K} = \mathbb{Q}$, $E = 2a^* + (-2)b^*$ (E is not in epsilon normal form). We then have $\widetilde{E} = a^* + b^*$. We can notice that $\widetilde{A_{\mathbb{K}}}(E) \neq A_{\mathbb{B}}(\widetilde{E})$ (E does not recognize ε but $\widetilde{E}$ does).

Lemma 3.4. *Let E be a $\mathbb{K}$-expression. If E is in SNF and in epsilon normal form, then*

$$\widetilde{A_{\mathbb{K}}}(E) = A_{\mathbb{B}}(\widetilde{E}).$$

Proof. We have to show that the automaton obtained by the Glushkov construction for an expression E in $\mathcal{E}_{\mathbb{K}}$ has the same edges as the Glushkov automaton for $\widetilde{E}$. First, we have $Pos(E) = Pos(\widetilde{E})$, as $\widetilde{E}$ is obtained from E only by deleting coefficients. Let us show that $First(\widetilde{E}) = P(First(E))$ (states reached from the initial state) by induction on the length of E. If $E = \varepsilon$, $\widetilde{E} = \varepsilon$, $First(\widetilde{E}) = \emptyset = First(E) = P(First(E))$. If $E = a \in \Sigma$, $\overline{E} = a_1$ then $E = \widetilde{E}$, $First(E) = \{(\overline{1}, 1)\}$, $P(First(E)) = \{1\} = First(\widetilde{E})$. Let F satisfy the hypothesis, and $E = kF$,$k \in \mathbb{K} \setminus \overline{0}$. In this case, $\widetilde{E} = \widetilde{F}$, $P(First(E)) = P(k.First(F)) = P(First(F)) = First(\widetilde{F}) = First(\widetilde{E})$. If $E = Fk$, $k \in \mathbb{K}$, $\widetilde{E} = \widetilde{F}$, $P(First(E)) = P(First(F)) = First(\widetilde{F}) = First(\widetilde{E})$.

If $E = F + H$, and if F and H satisfy the induction hypothesis, and as the coefficient of the empty word is $\overline{0}$ for one of the two subexpression F or H (epsilon normal form), we have $\widetilde{E} = \widetilde{F} + \widetilde{H}$, $First(\widetilde{F} + \widetilde{H}) = First(\widetilde{F}) \cup$

$\mathit{First}(\widetilde{H}) = P(\mathit{First}(F)) \cup P(\mathit{First}(H))$ which is equal to $P(\mathit{First}(F+H))$ by induction. We obtain the same result concerning $F \cdot H$, F^+ and F^*. The equality $\mathit{Last}(\widetilde{E}) = P(\mathit{Last}(E))$ is obtained similarly.

The last function used to compute the Glushkov automaton is the *Follow* function. Let E be a $\mathbb{K}$-expression and $i \in \mathit{Pos}(E)$. If $E = \varepsilon$, $\widetilde{E} = \varepsilon$, $\mathit{Follow}(\widetilde{E}, i) = \emptyset = \mathit{Follow}(E, i) = P(\mathit{Follow}(E, i))$. If $E = a \in \mathbb{K}$, $E = \widetilde{E}$, $\mathit{Follow}(\widetilde{E}, i) = \emptyset$. Let F satisfy $\mathit{Follow}(\widetilde{F}, i) = P(\mathit{Follow}(F, i))$ for all $i \in \mathit{Pos}(F)$. If E is kF or Fk, $k \in \mathbb{K} \setminus \overline{0}$, $P(\mathit{Follow}(E, i)) = P(\mathit{Follow}(F, i)) = \mathit{Follow}(\widetilde{F}, i)$ by hypothesis. If F and H satisfy the induction hypothesis, and if $E = F + H$, (and $i \in \mathit{Pos}(F)$ without loss of generality), $\mathit{Follow}(F + H, i) = \mathit{Follow}(F, i)$, then $P(\mathit{Follow}(F, i)) = \mathit{Follow}(\widetilde{F}, i)$. We obtain similar results for $E = F.H$ as there is no intersection between positions of F and H. Concerning the star operation, let $E = F^*$, with $\mathit{Follow}(\widetilde{F}, i) = P(\mathit{Follow}(F, i))$ for all $i \in \mathit{Pos}(F)$. Then, $P(\mathit{Follow}(F^*, i)) = P(\mathit{Follow}(F, i) \cup \mathit{Coeff}_{\mathit{Last}(F)}(i) \cdot \mathit{First}(F))$. But by definition, as F is in SNF, we know that $\mathit{Follow}(F, i) \cap \mathit{First}(F) = \emptyset$, so $P(\mathit{Follow}(F^*, i)) = \mathit{Follow}(\widetilde{F^*}, i)$. In fact, it means that if there exists a couple $(\alpha, j) \in \mathit{Follow}(F, i)$, there cannot exist $(\beta, j) \in \mathit{First}(F)$. Otherwise, the expression would not be in SNF, and it would be possible that $\beta = \alpha$, which would make $j \notin \mathit{Pos}(F^*)$ and imply a deletion of an edge. A same reasonning can be done for the positive closure operator.

Hence, the casting operation $\sim$ and the Glushkov construction commute for the composition operation if we do not consider the empty word. $\square$

3.2.4 *Characterization of Glushkov Automata in the Boolean Case*

The aim of the paper by Caron and Ziadi [6] is to know how boolean Glushkov graphs can be characterized. We recall here the definitions which allow us to give the main theorem of their paper. These notions will be necessary to extend this characterization to Glushkov $\mathbb{K}$-graphs.

A *hammock* is a graph $G = (X, U)$ without a loop if $|X| = 1$, otherwise it has two distinct vertices i and t such that, for any vertex x of X, (1) there exists a path from i to t going through x, (2) there is no non-trivial path from t to x nor from x to i. Notice that every hammock with at least two vertices has a unique root (the vertex i) and anti-root (the vertex t).

Let $G = (X, U)$ be a hammock. We define $\mathcal{O} = (X_{\mathcal{O}}, U_{\mathcal{O}}) \subseteq G$ as an *orbit* of G if and only if for all x and x' in $X_{\mathcal{O}}$ there exists a non-trivial path from x to x'. The orbit $\mathcal{O}$ is *maximal* if, for each vertex $x \in X_{\mathcal{O}}$ and for each vertex $x' \in X \setminus X_{\mathcal{O}}$, there do not exist both a path from x to x' and a path from x' to x. Equivalently, $\mathcal{O} \subseteq G$ is a maximal orbit of G if and only if it is a strongly connected component with at least one edge.

Informally, in a Glushkov graph obtained from a regular expression E, the set of vertices of a maximal orbit corresponds exactly to the set of positions of a closure subexpression of E.

The set of direct successors (respectively direct predecessors) of $x \in X$ is denoted by $Q^+(x)$ (respectively $Q^-(x)$). Let $n_x = |Q^-(x)|$ and $m_x = |Q^+(x)|$. For an orbit $\mathcal{O} \subset G$, $\mathcal{O}^+(x)$ denotes $Q^+(x) \cap (X \setminus \mathcal{O})$ and $\mathcal{O}^-(x)$ denotes the set $Q^-(x) \cap (X \setminus \mathcal{O})$. In other words, $\mathcal{O}^+(x)$ is the set of vertices which are directly reached from x and which are not in $\mathcal{O}$. By extension, $\mathcal{O}^+ = \bigcup_{x\in\mathcal{O}} \mathcal{O}^+(x)$ and $\mathcal{O}^- = \bigcup_{x\in\mathcal{O}} \mathcal{O}^-(x)$. The sets $\mathit{In}(\mathcal{O}) = \{x \in X_{\mathcal{O}} \mid \mathcal{O}^-(x) \neq \emptyset\}$ and $\mathit{Out}(\mathcal{O}) = \{x \in X_{\mathcal{O}} \mid \mathcal{O}^+(x) \neq \emptyset\}$ denote the *input* and the *output* of the orbit $\mathcal{O}$. As G is a hammock, $\mathit{In}(\mathcal{O}) \neq \emptyset$ and $\mathit{Out}(\mathcal{O}) \neq \emptyset$. An orbit $\mathcal{O}$ is *stable* if $\mathit{Out}(\mathcal{O}) \times \mathit{In}(\mathcal{O}) \subset U$. An orbit $\mathcal{O}$ is *transverse* if, for all $x, y \in \mathit{Out}(\mathcal{O})$, $\mathcal{O}^+(x) = \mathcal{O}^+(y)$ and, for all $x, y \in \mathit{In}(\mathcal{O})$, $\mathcal{O}^-(x) = \mathcal{O}^-(y)$.

An orbit $\mathcal{O}$ is *strongly stable* (respectively *strongly transverse*) if it is stable (respectively transverse) and if after deleting the edges in $\mathit{Out}(\mathcal{O}) \times \mathit{In}(\mathcal{O})$ (1) there does not exist any suborbit $\mathcal{O}' \subset \mathcal{O}$ or (2) every maximal suborbit of $\mathcal{O}$ is strongly stable (respectively strongly transverse). The hammock G is stronly stable (respectively strongly transverse) if (1) it has no orbit or (2) every maximal orbit $\mathcal{O} \subset G$ is strongly stable (respectively strongly transverse).

If G is strongly stable, then we call *the graph without orbit* of G, denoted by $SO(G)$, the acyclic directed graph obtained by recursively deleting, for every maximal orbit $\mathcal{O}$ of G, the edges in $\mathit{Out}(\mathcal{O}) \times \mathit{In}(\mathcal{O})$. The graph $SO(G)$ is then reducible if it can be reduced to one vertex by iterated applications of the three following rules:

- **Rule $\mathbf{R_1}$:** If x and y are vertices such that $Q^-(y) = \{x\}$ and $Q^+(x) = \{y\}$, then delete y and define $Q^+(x) := Q^+(y)$.
- **Rule $\mathbf{R_2}$:** If x and y are vertices such that $Q^-(x) = Q^-(y)$ and $Q^+(x) = Q^+(y)$, then delete y and any edge connected to y.
- **Rule $\mathbf{R_3}$:** If x is a vertex such that for all $y \in Q^-(x)$, $Q^+(x) \subset Q^+(y)$, then delete edges in $Q^-(x) \times Q^+(x)$.

Theorem 3.5 ([6]). *$G = (X, U)$ is a Glushkov graph if and only if the three following conditions are satisfied:*

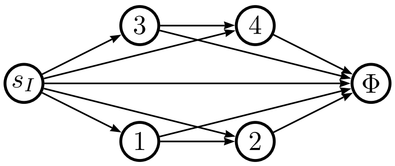
- *G is a hammock.*

- *Each maximal orbit in G is strongly stable and strongly transverse.*
- *The graph without orbit $SO(G)$ is reducible.*

3.2.5 *The Problem of Reduction Rules*

An Erroneous Statement in the Paper by Caron and Ziadi

In [6], the definition of the R_3 rules is wrong in some cases. Indeed, if we consider the regular expression $E = (x_1 + \varepsilon)(x_2 + \varepsilon) + (x_3 + \varepsilon)(x_4 + \varepsilon)$, the graph obtained from the Glushkov algorithm is as follows



Let us now try to reduce this graph with the reduction rules as they are defined in [6]. We can see that the sequel of applicable rules is R_3, R_3 and R_1. We can notice that there is a multiple choice for the application of the first R_3 rule, but after having chosen the vertex on which we will apply this first rule, the sequel of rules leads to a single graph (exept with the numerotation of vertices).

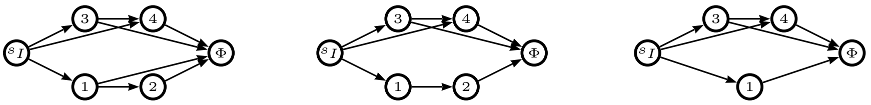


Fig. 3.1 Application of R_3 on 1, R_3 on 2 and R_1 on 1 and 2.

We can see that the graph obtained is no more reducible. This problem is a consequence of the multiple computation of the edge $(0, \Phi)$. In fact, this problem is solved when each edge of the acyclic Glushkov graph is computed only once. It is the case when E is in epsilon normal form.

A New R_3 Rule for the Boolean Case

Let $G = (X, U)$ be an acyclic graph. The rule R_3 is as follows:

- If $x \in X$ is a vertex such that for all $y \in Q^-(x)$, $Q^+(x) \subset Q^+(y)$, then delete the edge $(q^-, q^+) \in Q^-(x) \times Q^+(x)$ if there does not exist a vertex $z \in X \setminus \{x\}$ such that the following conditions are true:
  - there is neither a path from x to z nor a path from z to x,
  - $q^- \in Q^-(z)$ and $q^+ \in Q^+(z)$,
  - $|Q^-(z)| \times |Q^+(z)| \neq 1$.

The new rule R_3 check whether conditions of the old R_3 rules are verified and moreover deletes an edge only if it does not correspond to the ε of more than one subexpression. The validity of this rule is shown in Proposition 3.10.

3.3 Acyclic Glushkov WFA Properties

The definitions of section 3.2.4 related to graphs are extended to $\mathbb{K}$-graphs by considering that edges labeled $\overline{0}$ do not exist.

Let us consider M a WFA without orbit. Our aim here is to give conditions on weights in order to check whether M is a Glushkov WFA. Relying on the boolean characterization, we can deduce that M is homogeneous and that the Glushkov graph of $\widetilde{M}$ is reducible.

3.3.1 $\mathbb{K}$-*rules*

$\mathbb{K}$-rules can be seen as an extension of reduction rules. Each rule is divided into two parts: a graphic condition on edges, and a numerical condition (exept for the $\mathbb{K}R_1$-rule) on coefficients. The following definitions allow us to give numerical constraints for the application of $\mathbb{K}$-rules.

Let $G = (X, U)$ be a $\mathbb{K}$-graph and let $x, y \in X$. Let us now define the set of beginnings of the set $Q^-(x)$ as $B(Q^-(x)) \subseteq Q^-(x)$. A vertex x^- is in $B(Q^-(x))$ if for all q^- in $Q^-(x)$ there is not a non trivial path from q^- to x^-. In the same way, we define the set of terminations of $Q^+(x)$ as $T(Q^+(x)) \subseteq Q^+(x)$. A vertex x^+ is in $T(Q^+(x))$ if for all q^+ in $Q^+(x)$ there is not a non trivial path from x^+ to q^+.

We say that x and y are *backward equivalent* if $Q^-(x) = Q^-(y)$ and there exist $l_x, l_y \in \mathbb{K}$ such that for every $q^- \in Q^-(x)$, there exists $\alpha_{q^-} \in \mathbb{K}$ such that $U(q^-, x) = \alpha_{q^-} \otimes l_x$ and $U(q^-, y) = \alpha_{q^-} \otimes l_y$. Similarly, we say that x and y are *forward equivalent* if $Q^+(x) = Q^+(y)$ and there exist $r_x, r_y \in \mathbb{K}$ such that for every $q^+ \in Q^+(x)$, there exists $\beta_{q^+} \in \mathbb{K}$ such that $U(x, q^+) = r_x \otimes \beta_{q^+}$ and $U(y, q^+) = r_y \otimes \beta_{q^+}$. Moreover, if x and y

are both backward and forward equivalent, then we say that x and y are *bidirectionally equivalent.*

In the same way, we say that x is *ε-equivalent* if for all $(q^-, q^+) \in Q^-(x) \times Q^+(x)$ the edge (q^-, q^+) exists and if there exist $k, l, r \in \mathbb{K}$ such that for every $q^- \in Q^-(x)$ there exists $\alpha_{q^-} \in \mathbb{K}$ and for every $q^+ \in Q^+(x)$ there exist $\beta_{q^+} \in \mathbb{K}$, such that $U(q^-, x) = \alpha_{q^-} \otimes l$, $U(x, q^+) = r \otimes \beta_{q^+}$ and $U(q^-, q^+) = \alpha_{q^-} \otimes k \otimes \beta_{q^+}$.

Similarly, x is *quasi-ε-equivalent* if

- $B(Q^-(x)) \neq Q^-(x)$ or $T(Q^+(x)) \neq Q^+(x)$, and
- for all $(q^-, q^+) \in Q^-(x) \times Q^+(x) \setminus B(Q^-(x)) \times T(Q^+(x))$, the edge (q^-, q^+) exists, and
- there exist $k, l, r \in \mathbb{K}$ such that for every $q^- \in Q^-(x)$ there exist $\alpha_{q^-} \in \mathbb{K}$ and for every $q^+ \in Q^+(x)$, there exist $\beta_{q^+} \in \mathbb{K}$ such that $U(q^-, x) = \alpha_{q^-} \otimes l$, $U(x, q^+) = r \otimes \beta_{q^+}$, and
- if $q^- \notin B(Q^-(x))$ or $q^+ \notin T(Q^+(x))$
  - then $U(q^-, q^+) = \alpha_{q^-} \otimes k \otimes \beta_{q^+}$
  - else there exists $\gamma \in \mathbb{K}$ such that $U(q^-, q^+) = \gamma \oplus \alpha_{q^-} \otimes k \otimes \beta_{q^+}$ (Notice that if the edge from q^- to q^+ does not exist in the automaton, then $U(q^-, q^+) = \overline{0}$ and it is possible to have $\gamma \oplus \alpha_{q^-} \otimes k \otimes \beta_{q^+} = \overline{0}$).

In order to clarify our purpose, we have distinguished the case where (q^-, q^+) are superpositions of edges (quasi-ε-equivalence of x) to the case where they are not (ε-equivalence of x).

Rule $\mathbb{K}R_1$: If x and y are vertices such that $Q^-(y) = \{x\}$ and $Q^+(x) = \{y\}$, then delete y and define $Q^+(x) \leftarrow Q^+(y)$.

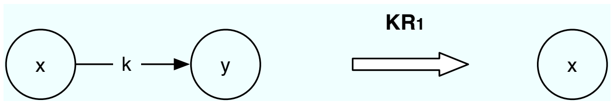


Fig. 3.2 $\mathbb{K}R_1$ reduction rule.

Rule $\mathbb{K}R_2$: If x and y are bidirectionally equivalent, with $l_x, l_y, r_x, r_y \in \mathbb{K}$ the constants satisfying such a definition, then

- delete y and any edge connected to y

- for every $q^- \in Q^-(x)$ and $q^+ \in Q^+(x)$ set $U'(q^-, x) = \alpha_{q^-}$ and $U'(x, q^+) = \beta_{q^+}$ where α_{q^-} and β_{q^+} are defined as in the bidirectional equivalence.

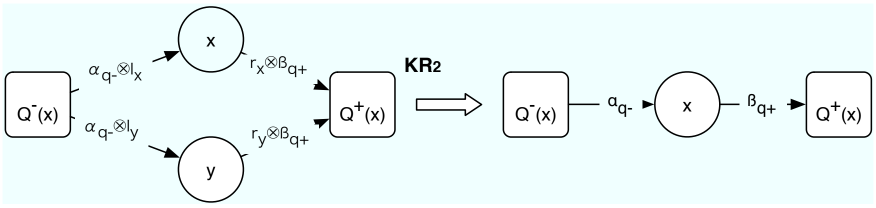


Fig. 3.3 $\mathbb{K}R_2$ reduction rule.

Rule $\mathbb{K}\boldsymbol{R_3}$: If x is ε-equivalent or x is quasi-ε-equivalent with $l, r, k, \gamma \in \mathbb{K}$ the constants satisfying such a definition, then

- if x is ε-equivalent
  - then delete every $(q^-, q^+) \in Q^-(x) \times Q^+(x)$,
  - else delete every $(q^-, q^+) \in Q^-(x) \times Q^+(x) \setminus B(Q^-(x)) \times T(Q^+(x))$.
- for every $q^- \in Q^-(x)$ and $q^+ \in Q^+(x)$ set $U'(q^-, x) = \alpha_{q^-}$ and $U'(x, q^+) = \beta_{q^+}$ where α_{q^-} and β_{q^+} are defined as in the ε-equivalence or quasi-ε-equivalence.
- If x is quasi-ε-equivalent then compute the new edges from $B(Q^-(x)) \times T(Q^+(x))$ labeled γ.

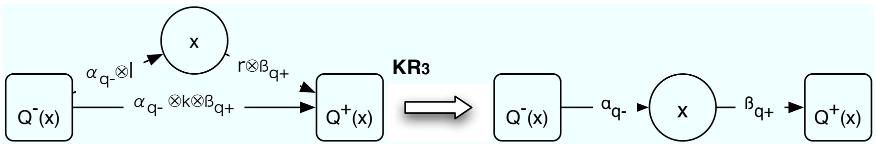


Fig. 3.4 $\mathbb{K}R_3$ reduction when x is ε-equivalent.

3.3.2 *Confluence for $\mathbb{K}$-rules*

In order to have an algorithm checking whether a $\mathbb{K}$-graph is a Glushkov $\mathbb{K}$-graph , we have to know (1) if it is decidable to apply a $\mathbb{K}$-rule on some vertices and (2) if the application of $\mathbb{K}$-rules ends. In order to ensure these

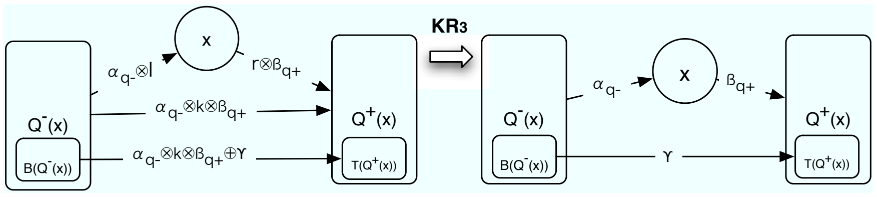


Fig. 3.5 $\mathbb{K}R_3$ reduction when x is quasi-ε-equivalent.

characteristics, we will specify some sufficient properties on the semiring $\mathbb{K}$. Let us define $\mathbb{K}$ as a field or as a factorial semiring. A factorial semiring $\mathbb{K}$ is a zero-divisor free semiring for which every non-zero, non-unit element x of $\mathbb{K}$ can be written as a product of irreducible elements of $\mathbb{K}$ $x = p_1 \cdots p_n$, and this representation is unique apart from the order of the irreducible elements. This notion is a slight adaptation of the factorial ring notion.

It is clear that, if $\mathbb{K}$ is a field, the application of $\mathbb{K}$-rules is decidable. Conditions of application of $\mathbb{K}$-rules are sufficient to define an algorithm. In the case of a factorial semiring, as the decomposition is unique, a *gcd* is defined[1] and it gives us a procedure allowing us to apply one rule ($\mathbb{K}R_2$ or $\mathbb{K}R_3$) on a $\mathbb{K}$-graph if it is possible. It ensures the decidability of $\mathbb{K}$-rules application for factorial semirings. For both cases (field and factorial semiring), we prove that $\mathbb{K}$-rules are confluent. It ensures the ending of the algorithm allowing us to know whether a $\mathbb{K}$-graph is a Glushkov one. Detailed algorithms can be found in [5].

For the $\mathbb{K}R_2$-rule, with notations of Figure 3.3, we check if there exists a value α_{q^-} for each $q^- \in Q^-(x)$ such that $U(q^-, x)$ and $U(q^-, y)$ can respectively be rewritten as $\alpha_{q^-} \otimes l_x$ and $\alpha_{q^-} \otimes l_y$ with $l_x \leftarrow \mathrm{gcd}_r(x)$ and $l_y \leftarrow \mathrm{gcd}_r(y)$ respectively the right gcd of values $\{U(q^-, x) \mid q^- \in Q^-(x)\}$ and $\{U(q^-, y) \mid q^- \in Q^-(y)\}$. Similar steps are applied to find a value β_{q^+} for each $q^+ \in Q^+(x)$ with $r_x \leftarrow \mathrm{gcd}_l(x)$ the left gcd of values $\{U(x, q^+) \mid q^+ \in Q^+(x)\}$.

For the $\mathbb{K}R_3$-rule, with notations of Figures 3.4 and 3.5, we check if there exists a value α_{q^-} for each $q^- \in Q^-(x)$ such that $U(q^-, x)$ can be rewritten as $\alpha_{q^-} \otimes l$ with $l \leftarrow \mathrm{gcd}_r(x)$ the right gcd of values $\{U(q^-, x) \mid q^- \in Q^-(x)\}$. A value β_{q^+} is searched for $q^+ \in Q^+(x)$ with $r \leftarrow \mathrm{gcd}_l(x)$ the left gcd of values $\{U(x, q^+) \mid q^+ \in Q^+(x)\}$. Then, we check if, for every edge from states of $Q^-(x)$ to states of $Q^+(x)$ unique values k and γ can be extracted

[1]In case $\mathbb{K}$ is not commutative, left *gcd* and right *gcd* are defined.

such that $U(q^-, q^+) = \begin{cases} \alpha_{q^-} \otimes k \otimes \beta_{q^+} & \text{if } x \text{ is } \varepsilon\text{-equivalent} \\ \alpha_{q^-} \otimes k \otimes \beta_{q^+} \oplus \gamma & \text{if } x \text{ is quasi-}\varepsilon\text{-equivalent} \end{cases}$

Definition 3.6 (Confluence). *Let G be a $\mathbb{K}$-graph and $\mathbb{I}_G$ the acyclic graph having only one vertex. Let R_1 be a sequence of $\mathbb{K}$-rules such that*

$$G \underset{R_1}{\longrightarrow} \mathbb{I}_G$$

$\mathbb{K}$-rules are confluent if for all $\mathbb{K}$-graph G_2 such that there exists R_2 a sequence of $\mathbb{K}$-rules with $G \underset{R_2}{\longrightarrow} G_2$ then there exists R_2' a sequence of $\mathbb{K}$-rules such that

$$G_2 \underset{R_2'}{\longrightarrow} \mathbb{I}_G$$

For the following, $\mathbb{K}$ is a field or a factorial semiring.

Proposition 3.7. *The $\mathbb{K}$-rules are confluent.*

Proof. In order to prove this result, we will show that if there exist two applicable $\mathbb{K}$-rules reducing a Glushkov $\mathbb{K}$-graph, then the order of application does not modify the resulting $\mathbb{K}$-graph.

Let us denote by $r_{x,y}(G)$ the application of a $\mathbb{K}R_1$, $\mathbb{K}R_2$ or $\mathbb{K}R_3$ rule on the vertices x and y with $y = \emptyset$ for a $\mathbb{K}R_3$ rule.

Let $G = (X, U)$ be a Glushkov $\mathbb{K}$-graph and let $r_{x,y}$ and $r_{z,t}$ be two applicable $\mathbb{K}$-rules on G such that $\{x, y\} \cap \{z, t\} = \emptyset$ and no edge can be deleted by both rules. Necessarily we have $r_{x,y}(r_{z,t}(G)) = r_{z,t}(r_{x,y}(G))$.

Suppose now that $\{x, y\} \cap \{z, t\} \neq \emptyset$ or one edge is deleted by both rules. We have to consider several cases depending on the rule $r_{x,y}$.

- $\mathbf{r_{x,y}}$ **is a** $\mathbb{K}\mathbf{R_1}$ **rule:**
  In this case $r_{z,t}$ can not delete the edge from x to y and $r_{z,t}$ is necessarily a $\mathbb{K}R_1$-rule with $\{x, y\} \cap \{z, t\} \neq \emptyset$. If $y = z$, as the coefficient does not act on the reduction rule, $r_{x,y}(r_{z,t}(G)) = r_{z,t}(r_{x,y}(G))$
- $\mathbf{r_{x,y}}$ **is a** $\mathbb{K}\mathbf{R_2}$ **rule:**
  Consider that $r_{z,t}$ is a $\mathbb{K}R_2$ rule with $y = z$. Using the notations of the $\mathbb{K}R_2$ rule, there exist α_{q^-}, β_{q^+}, l_x, l_y, r_x, r_y such that $U(q^-, x) = \alpha_{q^-} l_x$, $U(q^-, y) = \alpha_{q^-} l_y$, $U(x, q^+) = r_x \beta_{q^+}$ and $U(y, q^+) = r_y \beta_{q^+}$ with $q^- \in Q^-(x)$, $q^+ \in Q^+(x)$, and $l_x = \mathrm{gcd}_r(x)$, $l_y = \mathrm{gcd}_r(y)$ ($r_x = \mathrm{gcd}_l(x)$, $r_y = \mathrm{gcd}_l(y)$). By hypothesis, a $\mathbb{K}R_2$ rule can also be applied on the vertices y and t. There also exists α'_{q^-}, β'_{q^+}, l'_x, l'_t, r'_x, r'_t such that $\alpha_{q^-} = \alpha'_{q^-} l'_x$, $\beta_{q^+} = r'_x \beta'_{q^+}$, $U(q^-, t) = \alpha'_{q^-} l'_t$, $U(t, q^+) = r'_t \beta'_{q^+}$ ($Q^-(x) = Q^-(t)$ and $Q^+(x) = Q^+(t)$). By construction of $\mathrm{gcd}_r(x)$, the

left gcd of all α_{q^-} is $\overline{1}$. Then, whatever the order of application of $\mathbb{K}R_2$ rules, the same decomposition of edges values is obtained. Symetrically a same reasoning is applied for the right part.
Consider now that $r_{z,t} = r_{z,\emptyset}$ is a $\mathbb{K}R_3$ rule. Neither edges from x or y nor edges to x or y can be deleted by $r_{z,\emptyset}$. Then $z = x$ or $z = y$. Let $z = y$. If we successively apply $r_{x,y}$ and $r_{y,\emptyset}$ or $r_{y,\emptyset}$ and $r_{x,y}$ on G, we obtain the same $\mathbb{K}$-graph following the same method as the previous case. If we choose $z = x$, we have also the same $\mathbb{K}$-graph (commutativity property of the sum operator).

- **$\mathbf{r_{x,y}}$ is a $\mathbb{K}\mathbf{R_3}$ rule:**
The only case to consider now is $r_{z,t} = r_{z,\emptyset}$ a $\mathbb{K}R_3$ rule. Suppose that $r_{z,\emptyset}$ deletes an edge also deleted by $r_{x,\emptyset}$ (with $x \neq z$). Let (q^-, q^+) be this edge.
Using the notations of the $\mathbb{K}R_3$ rule, there exist α_{q^-}, β_{q^+}, l, r such that $U(q^-, x) = \alpha_{q^-} l$, $U(x, q^+) = r\beta_{q^+}$, $U(q^-, q^+) = \alpha_{q^-} k \beta_{q^+} \oplus \gamma$ with $q^- \in Q^-(x)$, $q^+ \in Q^+(x)$ and $l = \text{gcd}_r(x)$, $r = \text{gcd}_l(x)$. There also exists α'_{q^-}, β'_{q^+}, l', r' such that $U(q^-, z) = \alpha'_{q^-} l'$, $U(z, q^+) = r' \beta'_{q^+}$, $U(q^-, q^+) = \alpha'_{q^-} k' \beta'_{q^+} \oplus \gamma'$ with $l' = \text{gcd}_r(z)$, $r' = \text{gcd}_l(z)$. By construction, the computation of l and l' (r and r') are independant. A same reasoning is applied for the right part. Then we can choose γ'' such that $\gamma = \alpha'_{q^-} k' \beta'_{q^+} \oplus \gamma''$ and $\gamma' = \alpha_{q^-} k \beta_{q^+} \oplus \gamma''$. So $U(q^-, q^+) = \alpha_{q^-} k \beta_{q^+} \oplus \alpha'_{q^-} k' \beta'_{q^+} \oplus \gamma''$. It is easy to see that $r_{x,\emptyset}(r_{z,\emptyset}(G)) = r_{z,\emptyset}(r_{x,\emptyset}(G))$.

$\square$

3.3.3 $\mathbb{K}$-*reducibility*

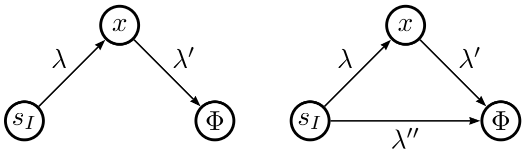
Definition 3.8. A $\mathbb{K}$-graph $G = (X, U)$ is said to be $\mathbb{K}$-*reducible* if it has no orbit and if it can be reduced to one vertex by iterated applications of any of the three rules $\mathbb{K}R_1$, $\mathbb{K}R_2$, $\mathbb{K}R_3$ described below.

Proposition 3.10 shows the existence of a sequel of $\mathbb{K}$-rules leading to the complete reduction of Glushkov $\mathbb{K}$-graphs. However, the existence of an algorithm allowing us to obtain this sequel of $\mathbb{K}$-rules depends on the semiring $\mathbb{K}$.

In order to show the $\mathbb{K}$-reducibility property of a Glushkov $\mathbb{K}$-graph G, we check (Lemma 3.9) that every sequence $\mathcal{R}$ of $\mathbb{K}$-rules leading to the $\mathbb{K}$-reduction of G contains necessarily two $\mathbb{K}R_1$ rules which will be denoted by $r_\circ$ and $r_\bullet$.

Lemma 3.9. *Let $G = (X, U)$ be a $\mathbb{K}$-reducible Glushkov $\mathbb{K}$-graph without orbit with $|X| \geq 3$, and let $\mathcal{R} = r_1 \cdots r_n$ be the sequence of $\mathbb{K}$-rules which can be applied on G and reduce it. Necessarily, $\mathcal{R}$ can be written $\mathcal{R}' r_\circ r_\bullet$ with $r_\circ$ and $r_\bullet$ two $\mathbb{K}R_1$-rules merging respectively s_I and Φ.*

Proof. We show this lemma by induction on the number of vertices of the graph. It is obvious that if $|X| = 3$ then, the only possible graphs are the following ones:



and then, for the first one $\mathcal{R} = r_\circ r_\bullet$ with $k = \lambda$ in $r_\circ$ and $k = \lambda'$ in $r_\bullet$. For the second one x is ε-equivalent and $\mathcal{R} = r r_\circ r_\bullet$ with r a $\mathbb{K}R_3$-rule such that $\alpha = \overline{1}$, $\beta = \overline{1}$, $l = \lambda$, $r = \lambda'$ and $k = \lambda''$. Then, $r_\circ$ and $r_\bullet$ are $\mathbb{K}R_1$ rules such that $k = \overline{1}$ for $r_\circ$ and $r_\bullet$. Suppose now that G has n vertices. As it is $\mathbb{K}$-reducible, there exists a sequence of $\mathbb{K}$-rules which leads to one of the two previous basic cases. □

For the reduction process, we associate each vertex of G to a subexpression. We define $E(x)$ to be the expression of the vertex x. At the beginning of the process, $E(x)$ is a, the only letter labelling edges reaching the vertex x (homogeneity of Glushkov automata). For the vertices s_I and Φ, we define $E(s_I) = E(\Phi) = \varepsilon$. When applying $\mathbb{K}$-rules, we associate a new expression to each new vertex. With notations of Figure 3.2, the $\mathbb{K}R_1$-rule induces $E(x) \leftarrow E(x) \cdot k \times E(y)$ with $k = U(x, y)$. With notations of Figure 3.3, the $\mathbb{K}R_2$-rule induces $E(x) \leftarrow l_x E(x) r_x + l_y E(y) r_y$. And with notations of Figures 3.4 and 3.5, the $\mathbb{K}R_3$-rule induces $E(x) \leftarrow lF(x)r + k$.

Proposition 3.10. *Let $G = (X, U)$ be a $\mathbb{K}$-graph without orbit. The graph G is a Glushkov $\mathbb{K}$-graph if and only if it is $\mathbb{K}$-reducible.*

Proof. ($\Rightarrow$) This proposition will be proved by recurrence on the length of the expression. First for $||E|| = 1$, we have only two proper $\mathbb{K}$-expressions which are $E = \lambda$ and $E = \lambda a \lambda'$, for $\lambda, \lambda' \in \mathbb{K}$. When $E = \lambda$, the Glushkov $\mathbb{K}$-graph has only two vertices which are s_I and Φ and the edge (s_I, Φ) is labeled with λ. Then the $\mathbb{K}R_1$ rule can be applied. Suppose now that $E = \lambda a \lambda'$, then the Glushkov $\mathbb{K}$-graph of E has three vertices and is $\mathbb{K}$-reducible. Indeed, the $\mathbb{K}R_1$-rule can be applied twice.

Suppose now that for each proper $\mathbb{K}$-expression E of length n, its Glushkov $\mathbb{K}$-graph is $\mathbb{K}$-reducible. We then have to show that the Glushkov $\mathbb{K}$-graph of $\mathbb{K}$-expressions $F = E + \lambda$, $F = E + \lambda a \lambda'$, $F = \lambda a \lambda' \cdot E$ and $F = E \cdot \lambda a \lambda'$ of length $n+1$ are $\mathbb{K}$-reducible. Let us denote by $\mathcal{R}$ (respectively $\mathcal{R}'$) the sequence of rules which can be applied on $A_{\mathbb{K}}(E)$ (respectively $A_{\mathbb{K}}(F)$). In case $|X| \geq 3$, $\mathcal{R} = \mathcal{R}_b r_\circ r_\bullet$ (respectively $\mathcal{R}' = \mathcal{R}'_b r'_\circ r'_\bullet$).

- case $F = E + \lambda$
  We have $Pos(F) = Pos(E)$, $First(F) = First(E)$, $Last(F) = Last(E)$, $Null(F) = Null(E) + \lambda$ and $\forall i \in Pos(E)$, $Follow(F, i) = Follow(E, i)$. Every rule which can be applied on $A_{\mathbb{K}}(E)$ and which does not modify the edge (s_I, Φ) can also be applied on $A_{\mathbb{K}}(F)$.
  If $A_{\mathbb{K}}(E)$ has only two states, then $\mathcal{R} = r$ a $\mathbb{K}R_1$-rule, and then $\mathcal{R}' = r'$ a $\mathbb{K}R_1$- rule where r' is such that $k = Null(E) + \lambda$. Elsewhere, the (s_I, Φ) edge can only be reduced by a $\mathbb{K}R_3$ rule.
  Suppose now that there is no $\mathbb{K}R_3$ rule modifying (s_I, Φ) which can be applied on $A_{\mathbb{K}}(E)$. Then there is a $\mathbb{K}R_3$ rule r' which can be applied on $A_{\mathbb{K}}(F)$ with $k = \lambda$ and then $A_{\mathbb{K}}(F)$ can be reduced by $\mathcal{R}' = \mathcal{R}_b r' r'_\circ r'_\bullet$.
  Let us now suppose that $r_1, r_2, \cdots r_n$ is the subsequence of $\mathbb{K}R_3$-rules of $\mathcal{R}$ which modify the (s_I, Φ) edge. Necessarily, r_n acts on a state x which is ε-equivalent. If $Q^-(x) \neq \{s_I\}$ or $Q^+(x) \neq \{\Phi\}$ then $\mathcal{R}'_b = \mathcal{R}_b r_{n+1}$ where r_n in $\mathcal{R}'_b$ is modified as follows: x is quasi-ε-equivalent with $\gamma = \lambda$ and the rule r_{n+1} is a $\mathbb{K}R_3$ rule on a state x which is ε-equivalent and $k = \lambda$. Elsewhere, there is two cases to distinguish. If $Null(E) \oplus \lambda = \overline{0}$ then the r_n rule is no more applicable on $A_{\mathbb{K}}(F)$ (no edge between s_I and Φ) and the r_{n-1} rule in $\mathcal{R}'$ now acts on an ε-equivalent vertex in $A_{\mathbb{K}}(F)$. If $Null(E) + \lambda \neq \overline{0}$ then r_n can be applied on $A_{\mathbb{K}}(F)$ with $k = k \oplus \lambda$.
- case $F = E + \lambda a \lambda'$
  If $|Pos(E)| = n$, we have, $Pos(F) = Pos(E) \cup \{n+1\}$, $First(F) = First(E) \uplus \{(\lambda, n+1)\}$, $Last(F) = Last(E) \uplus \{(\lambda', n+1)\}$, $Null(F) = Null(E)$ and $\forall i \in Pos(E)$, $Follow(F, i) = Follow(E, i)$ and $Follow(F, n+1) = \emptyset$. In this case, $\mathcal{R}' = \mathcal{R}_b r r'_\circ r'_\bullet$ where r is a $\mathbb{K}R_2$ rule with $\alpha_{s_I} = \beta_\Phi = \overline{1}$ and $l_y = \lambda$, $r_y = \lambda'$ and so $A_{\mathbb{K}}(F)$ is $\mathbb{K}$-reducible.
- case $F = E \cdot \lambda a \lambda'$
  If $|Pos(E)| = n$, we have, $Pos(F) = Pos(E) \cup \{n+1\}$, $First(F) = First(E)$, $Last(F) = \{(\lambda', n+1)\}$, $Null(F) = \emptyset$ and $\forall i \in Pos(E) \setminus P(Last(E))$, $Follow(F, i) = Follow(E, i)$ and $\forall i \in P(Last(E))$, $Follow(F, i) = Follow(E, i) \uplus \{(\lambda, n+1)\}$. Let $r_1, \cdots r_n$ be the subsequel of $\mathbb{K}$-rules modifying edges reaching Φ. Necessarily, $n = 1$ and

$r_1 = r_\bullet$ (Lemma 3.9). Indeed, let us suppose that $n > 1$ and that there exists $j \neq i$ such that r_j is a $\mathbb{K}R_1$, $\mathbb{K}R_2$, or $\mathbb{K}R_3$-rule. Necessarily $|Q^-(\Phi)| \geq 1$, which contradicts our hypothesis. Then we have $\mathcal{R}' = \mathcal{R}r_{n+1}$ where $r_\bullet$ the $\mathbb{K}R_1$-rule from a vertex x to Φ of the sequence $\mathcal{R}$ and labeled with k_i is modified in $\mathcal{R}'$ as follows: $k = k_i \otimes \lambda$. We have also $k = \lambda'$ for the rule r_{n+1}.

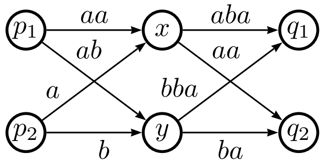
The case $F = \lambda a \lambda' \cdot E$ is proved similarily as the previous one considering the rules modifying edges from s_I (with $r_\circ$ instead of $r_\bullet$).

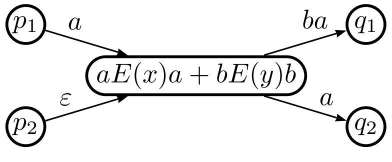
($\Leftarrow$) By induction on the number of states of the reducible $\mathbb{K}$-graph $G = (X, U)$. If $|X| = 2$, $X = \{s_I, \Phi\}$ and the only $\mathbb{K}$-expression E is λ with $\lambda \in \mathbb{K}$. Let $G' = (X', U')$ be the Glushkov $\mathbb{K}$-graph obtained from E. By construction $\lambda = U(s_I, \Phi) = E(s_I)$ and $\lambda = \mathit{Null}(E)$, necessarily $G' = G$.

We consider the property true for ranks bellow $n+1$ and G a $\mathbb{K}$-graph partially reduced. Three cases can occur according to the graphic form of the partially reduced graph. Either we will have to apply twice the $\mathbb{K}R_1$-rule or once the $\mathbb{K}R_3$-rule and twice the $\mathbb{K}R_1$-rule if $X = \{s_I, x, \Phi\}$, or we will have to apply once the $\mathbb{K}R_2$-rule and twice the $\mathbb{K}R_1$-rule if $X = \{s_I, x, y, \Phi\}$. For each case, we compute successively the new expressions of vertices, and we check that the Glushkov construction applied on the final $\mathbb{K}$-expression is G. □

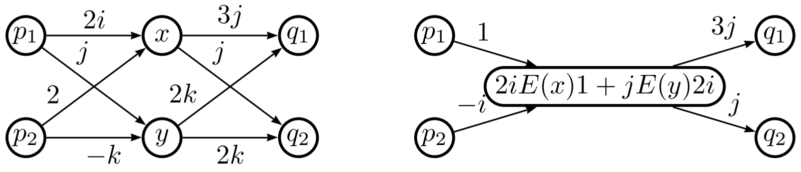
3.3.4 *Several Examples of Use for* $\mathbb{K}$*-rules*

For the $\mathbb{K}R_2$ rule, the first example is for transducers in $(\mathbb{K}, \oplus, \otimes) = (\Sigma^* \cup \emptyset, \cup, \cdot)$ where "$\cdot$" denotes the concatenation operator. In this case, we can express the $\mathbb{K}R_2$ rule conditions as follows. For all q^- in $Q^-(x)$, α_{q^-} is the common prefix of $U(q^-, x)$ and $U(q^-, y)$. Likewise, for all q^+ in $Q^+(x)$, β_{q^+} is the common suffix of $U(q^+, x)$ and $U(q^+, y)$.





The second one is in $(\mathbb{Z}/7\mathbb{Z}[i, j, k], \oplus, \otimes)$, where $\{i, j, k\}$ are elements of the quaternions and $\oplus$ is the sum and $\otimes$ the product. In this case, $\mathbb{K}$ is a field. Every factorization leads to the result.



We now give a complete example using the three rules on the $(\mathbb{N} \cup \{+\infty\}, min, +)$ semiring. This example enlightens the reader on the problem of the quasi-ε equivalence. For this example, we will identify the vertex with its label.

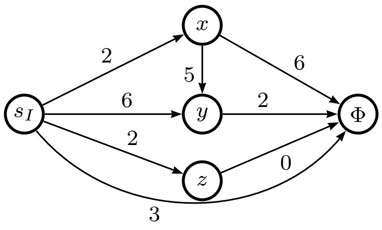


Fig. 3.6 A $(min, +)$-WFA.

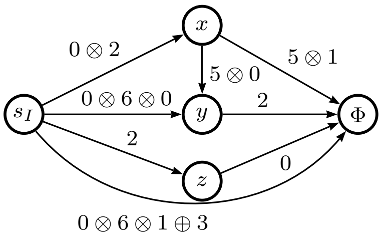


Fig. 3.7 $\mathbb{K}R_3$ rule can be applied on x with $l_x = 2$, $r_x = 5$ and $k = 6$.

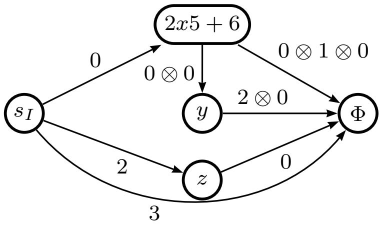


Fig. 3.8 $\mathbb{K}R_3$ rule can be applied on y with $l_y = 0$, $r_y = 2$ and $k = 1$.

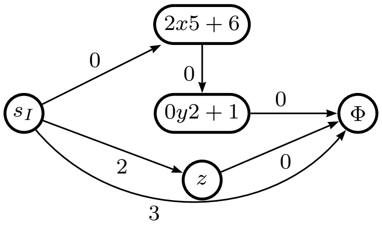


Fig. 3.9 $\mathbb{K}R_1$ rule can be applied on $(2x5 + 6)$ and on $(0y2 + 1)$.

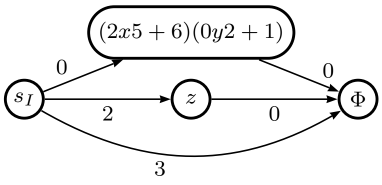


Fig. 3.10 $\mathbb{K}R_2$ rule can be applied on $(2x5 + 6)(0y2 + 1)$ and on z.

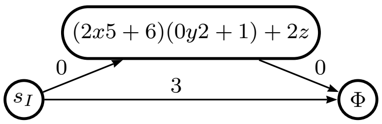


Fig. 3.11 A $\mathbb{K}R_3$ rule can be applied to end the process.

This example leads to a possible $\mathbb{K}$-expression such as

$$E = ((2x5 + 6)(0y2 + 1) + 2z) + 3$$

3.4 Glushkov $\mathbb{K}$-graph with Orbits

We will now consider a graph which has at least one maximal orbit $\mathcal{O}$. We extend the notions of strong stability and strong transversality to the $\mathbb{K}$-graphs obtained from $\mathbb{K}$-expressions in SNF. We have to give a characterization on coefficients only. The stability and transversality notions are rather linked. Indeed, if we consider the states of $In(\mathcal{O})$ as those of $\mathcal{O}^+$ then both notions amount to the transversality. Moreover, the extension of these notions to WFAs ($\mathbb{K}$-stability - definition 3.12 - and $\mathbb{K}$-transversality - definition 3.14), implies the manipulation of output and input vectors of $\mathcal{O}$ whose product is exactly the orbit matrix of $\mathcal{O}$ (Proposition 3.17).

Lemma 3.11. *Let E be a $\mathbb{K}$-expression and $G_{\mathbb{K}}(E)$ its Glushkov $\mathbb{K}$-graph. Let $\mathcal{O} = (X_{\mathcal{O}}, U_{\mathcal{O}})$ be a maximal orbit of $G_{\mathbb{K}}(E)$. Then E contains a closure subexpression F such that $X_{\mathcal{O}} = Pos(F)$.*

This lemma is a direct consequence of Lemma 4.5 in [6] and of Lemma 3.4.

Definition 3.12 ($\mathbb{K}$-stability). *A maximal orbit $\mathcal{O}$ of a $\mathbb{K}$-graph $G = (X, U)$ is $\mathbb{K}$-stable if*

- *$\widetilde{\mathcal{O}}$ is stable and*
- *the matrix $M_{\mathcal{O}} \in \mathbb{K}^{|Out(\mathcal{O})| \times |In(\mathcal{O})|}$ such that $M_{\mathcal{O}}(s, e) = U(s, e)$, for each (s, e) of $Out(\mathcal{O}) \times In(\mathcal{O})$, can be written as a product VW of two vectors such that $V \in \mathbb{K}^{|Out(\mathcal{O})| \times 1}$ and $W \in \mathbb{K}^{1 \times |In(\mathcal{O})|}$.*

The graph G is $\mathbb{K}$-stable if each of its maximal orbits is $\mathbb{K}$-stable.

If a maximal orbit $\mathcal{O}$ is $\mathbb{K}$-stable, $M_{\mathcal{O}}$ is a matrix of rank 1 called the *orbit matrix.* Then, for a decomposition of $M_{\mathcal{O}}$ in the product VW of two vectors, V will be called the *tail-orbit vector* of $\mathcal{O}$ and W will be called the *head-orbit vector* of $\mathcal{O}$.

Lemma 3.13. *A Glushkov $\mathbb{K}$-graph obtained from a $\mathbb{K}$-expression E in SNF is $\mathbb{K}$-stable.*

Proof. Let G be the Glushkov $\mathbb{K}$-graph of a $\mathbb{K}$-expression E in SNF, $A_{\mathbb{K}}(E) = (\Sigma, Q, s_I, F, \delta)$ its Glushkov WFA and $\mathcal{O} = (X_{\mathcal{O}}, U_{\mathcal{O}})$ be a maximal orbit of G. Following Lemma 3.4 and Theorem 3.5, G is strongly stable which implies that every orbit of G is stable. Let $s_i \in Out(\mathcal{O})$, $1 \leq i \leq |Out(\mathcal{O})|$ and $e_j \in In(\mathcal{O})$, $1 \leq j \leq Out(\mathcal{O})$. Following the extended Glushkov construction and as for all $s_i \in Out(\mathcal{O})$, $s_i \neq s_I$, we have $\delta(s_i, a, e_j) = \mathit{Coeff}_{Follow(E,s_i)}(e_j)$. As $\mathcal{O}$ corresponds to a closure subexpression F^* or F^+ (Lemma 3.11) and as (s_i, a, e_j) is an edge of $X_{\mathcal{O}} \times \Sigma \times X_{\mathcal{O}}$, we have $\delta(s_i, a, e_j) = \mathit{Coeff}_{Follow(F^*,s_i)}(e_j) = \mathit{Coeff}_{Follow(F,s_i) \uplus \mathit{Coeff}_{Last(F)}(s_i).First(F)}(e_j)$. As E is in SNF, so are F^* and F^+, and then $\delta(s_i, a, e_j) = \mathit{Coeff}_{Coeff_{Last(F)(s_i)}.First(F)}(e_j) = \mathit{Coeff}_{Last(F)}(s_i).\mathit{Coeff}_{First(F)}(e_j)$. The lemma is proved choosing $V \in \mathbb{K}^{|Out(\mathcal{O})| \times 1}$ such that $V(i,1) = \mathit{Coeff}_{Last(F)}(s_i)$ and $W \in \mathbb{K}^{1 \times |In(\mathcal{O})|}$ with $W(1,j) = \mathit{Coeff}_{First(F)}(e_j)$. $\square$

Definition 3.14 ($\mathbb{K}$-transversality). *A maximal orbit $\mathcal{O}$ of $G = (X, U)$ is $\mathbb{K}$-transverse if*

- *$\widetilde{\mathcal{O}}$ is transverse,*
- *the matrix $M_e \in \mathbb{K}^{|\mathcal{O}^-| \times |In(\mathcal{O})|}$ such that $M_e(p, e) = U(p, e)$ for each (p, e) of $\mathcal{O}^- \times In(\mathcal{O})$, can be written as a product ZT of two vectors such that $Z \in \mathbb{K}^{|\mathcal{O}^-| \times 1}$ and $T \in \mathbb{K}^{1 \times |In(\mathcal{O})|}$,*
- *the matrix $M_s \in \mathbb{K}^{|Out(\mathcal{O})| \times |\mathcal{O}^+|}$ such that $M_s(s, q) = U(s, q)$ for each (s, q) of $Out(\mathcal{O}) \times \mathcal{O}^+$, can be written as a product $T'Z'$ of two vectors such that $T' \in \mathbb{K}^{|Out(\mathcal{O})| \times 1}$ and $Z' \in \mathbb{K}^{1 \times |\mathcal{O}^+|}$.*

The graph G is $\mathbb{K}$-transverse if each of its maximal orbits is $\mathbb{K}$-transverse.

If a maximal orbit $\mathcal{O}$ is $\mathbb{K}$-transverse, M_e (respectively M_s) is a matrix of rank 1 called the *input matrix* of $\mathcal{O}$ (respectively *output matrix* of $\mathcal{O}$). For a decomposition of M_e (respectively M_s) in the product ZT (respectively $T'Z'$) of two vectors, T will be called the *input vector* (respectively T' will be called the *output vector*) of $\mathcal{O}$.

Lemma 3.15. *The Glushkov $\mathbb{K}$-graph $G = (X, U)$ of a $\mathbb{K}$-expression E in SNF is $\mathbb{K}$-transverse.*

Proof. Let $\mathcal{O}$ be a maximal orbit of G. Following Lemma 3.4 and Theorem 3.5, G is strongly transverse implies that $\mathcal{O}$ is transverse. By Lemma 3.11, there exists a maximal closure subexpression H such that $H = F^*$ or $H = F^+$. As E is in normal form, so is H. By the definition of the function *Follow*, we have in this case: for all $p \in Out(\mathcal{O})$, for all $q \in O^+$, $U(p, q) = \mathit{Coeff}_{Follow(F,p)}(q)$. We now have to distinguish three cases.

(1) If $|\mathcal{O}^+| = 1$, then the result holds immediatly. Indeed the output matrix of $\mathcal{O}$ is a vector.
(2) If $\mathcal{O}^+ = \{q_1, \cdots, q_n\}$ and $n > 1$, $\forall 1 \leq j \leq n, q_j \neq \Phi$, necessarily, we have $\mathcal{O}^+ = \bigcup_l P(First(H_l))$ with H_l some subexpressions of E. Then we have $U(p, q_j) = \mathit{Coeff}_{\mathit{Coeff}_{Last(F)}(p).First(H_l)}(q_j)$ if $q_j \in P(First(H_l))$. Then as q_j is a first position of only one subexpression, $U(p, q_j) = k_p \otimes \mathit{Coeff}_{First(H_l)}(q_j)$ where $k_p = \mathit{Coeff}_{Last(F)}(p)$ which concludes this case.
(3) Now if $\exists 1 \leq j \leq n \mid q_j = \Phi$ then $U(p, q_j) = \mathit{Coeff}_{Last(F)}(p) \otimes k$ where k is the $Null$ value of some subexpression following F not depending on p.

A same reasoning can be used for the left part of the transversality. □

Definition 3.16 ($\mathbb{K}$-balanced). *The orbit $\mathcal{O}$ of a graph G is $\mathbb{K}$-balanced if $\mathcal{O}$ is $\mathbb{K}$-stable and $\mathbb{K}$-transverse and if there exists an input vector T of $\mathcal{O}$ and an output vector T' of $\mathcal{O}$ such that the orbit matrix $M_{\mathcal{O}} = T'T$. The graph G is $\mathbb{K}$-balanced if each of its maximal orbits is $\mathbb{K}$-balanced.*

Proposition 3.17. *The Glushkov $\mathbb{K}$-graph obtained from a $\mathbb{K}$-expression E in SNF is $\mathbb{K}$-balanced.*

Proof. Lemma 3.13 enlightens on the fact that V, the tail-orbit vector of $\mathcal{O}$, is such that $V(i, 1) = \mathit{Coeff}_{Last(F)}(i)$ for all $i \in P(Last(F))$, which is, from Lemma 3.15, the output vector of $\mathcal{O}$. The details of the proofs for these lemmas show in the same way that there exists an head-orbit vector and an input vector for $\mathcal{O}$ which are equal. □

We can now define the recursive version of WFA $\mathbb{K}$-balanced property.

Definition 3.18. A $\mathbb{K}$-graph is strongly $\mathbb{K}$-balanced if (1) it has no orbit or (2) it is $\mathbb{K}$-balanced and if after deleting all edges $Out(\mathcal{O}) \times In(\mathcal{O})$ of each maximal orbit $\mathcal{O}$, it is strongly $\mathbb{K}$-balanced.

Proposition 3.19. *The Glushkov $\mathbb{K}$-graph obtained from a $\mathbb{K}$-expression E in SNF is strongly $\mathbb{K}$-balanced.*

Proof. Let G be the Glushkov $\mathbb{K}$-graph of a $\mathbb{K}$-expression E and $\mathcal{O}$ be a maximal orbit of G. The Glushkov $\mathbb{K}$-graph G is strongly stable and strongly transverse. As E is in SNF, edges of $Out(\mathcal{O}) \times In(\mathcal{O})$ that are deleted are backward edges of a unique closure subexpression F^* or F^+. Consequently, the recursive process of edges removal deduced from the definition of strong $\mathbb{K}$-stability produces only maximal orbits which are $\mathbb{K}$-balanced. The orbit $\mathcal{O}$ is therefore strongly $\mathbb{K}$-balanced. □

Theorem 3.20. *Let $G = (X, U)$. G is a Glushkov $\mathbb{K}$-graph of a $\mathbb{K}$-expression E in SNF if and only if*

- *G is strongly $\mathbb{K}$-balanced.*
- *The graph without orbit of G is $\mathbb{K}$-reducible.*

Proof. Let $G = (X, U)$ be a Glushkov $\mathbb{K}$-graph. From Proposition 3.19, G is strongly $\mathbb{K}$-balanced. The graph without orbit of G is $\mathbb{K}$-reducible (Proposition 3.10) For the converse part of the theorem, if G has no orbit and G is $\mathbb{K}$-reducible, by Proposition 3.10 the result holds immediatly. Let $\mathcal{O}$ be a maximal orbit of G. As it is strongly $\mathbb{K}$-balanced, we can write $M_{\mathcal{O}} = VW$ the orbit matrix of $\mathcal{O}$, there exists an output vector T' equal to the tail-orbit vector V and an input vector T equal to the head-orbit vector W. If the graph without orbit of $\mathcal{O}$ corresponds to a $\mathbb{K}$-expression F then $\mathcal{O}$ corresponds to the $\mathbb{K}$-expression F^+ where $\mathit{Coeff}_{First(F^+)}(i) = W(1, i), \forall i \in P(First(F^+))$, $\mathit{Coeff}_{Last(F^+)}(j) = V(j, 1), \forall j \in P(Last(F^+))$. We have also $\mathit{Coeff}_{Follow(F^+,j)}(i) = \mathit{Coeff}_{Follow(F,j) \uplus \mathit{Coeff}_{Last(F)} . First(F)}(i)$, $\forall j \in P(Last(F))$ and $\forall i \in P(First(F))$. Hence the Glushkov functions are well defined.

We now have to show that the graph without orbit of $\mathcal{O}$ can be reduced to a single vertex. By the successive applications of the $\mathbb{K}$-rules, the vertices of the graph without orbit of $\mathcal{O}$ can be reduced to a single state (giving a $\mathbb{K}$-rational expression for $\mathcal{O}$). Indeed, as $\mathcal{O}$ is transverse, no $\mathbb{K}$-rule concerning one vertex of $\mathcal{O}$ and one vertex out of $\mathcal{O}$ can be applied. □

3.5 Algorithm for Orbit Reduction

In this section, we present a recursive algorithm that computes a $\mathbb{K}$-expression from a Glushkov $\mathbb{K}$-graph . We then give an example which illustrates this method.

Algorithms

The BackEdgesRemoval function (Algorithm 2) on $\mathcal{O}$ deletes edges from $Out(\mathcal{O})$ to $In(\mathcal{O})$, returns true if vectors T, T', Z, Z' (as defined in definition 3.14) can be computed, false otherwise.

Algorithm 1 Recursive orbit reduction.

```
OrbitReduction($G$)
  ▷ Input: A $\mathbb{K}$-graph $G = (X, U)$
  ▷ Output: A newly computed graph without orbit
 1 Begin
 2   for each maximal orbit $\mathcal{O} = (X_{\mathcal{O}}, U_{\mathcal{O}})$ of $G$ do
 3     if BackEdgesRemoval($\mathcal{O}, T, T', Z, Z'$) then
 4       if OrbitReduction($\mathcal{O}$) then
 5         if Expression($E_{\mathcal{O}}, \mathcal{O}, T, T'$) then
 6           ReplaceStates($G, \mathcal{O}, E_{\mathcal{O}}, Z, Z'$)
 7         else return $False$
 8       else return $False$
 9     else return $False$
10   return $True$
11 End
```

The Expression function returns true, computes the $\mathbb{K}$-expression E of $G' = (\mathcal{O} \cup \{s_I, \Phi\}, U')$ where $U' \leftarrow U \cup \{(s_I, T(1,j), e_j) \mid e_j \in \mathit{In}(\mathcal{O})\} \cup \{(s_i, T'(i,1), \Phi) \mid s_i \in \mathit{Out}(\mathcal{O})\}$ and ouputs $E_{\mathcal{O}} \leftarrow E^+$ if $\mathcal{O}$ is $\mathbb{K}$-reducible. It returns false otherwise.

The ReplaceStates function replaces $\mathcal{O}$ by one state x labeled $E_{\mathcal{O}}$ and connected to $\mathcal{O}^-$ and $\mathcal{O}^+$ with the sets of coefficients of Z and Z'. Formally $G = (X \setminus X_{\mathcal{O}} \cup \{x\}, U)$ with $U \leftarrow U \setminus \{(u,k,v) \mid u, v \in \mathcal{O}\} \cup \{(p_j, Z(j,1), x) \mid p_j \in \mathcal{O}^-\} \cup \{(x, Z'(1,i), q_i) \mid q_i \in \mathcal{O}^+\}$.

Illustrated Example

We illustrate Glushkov WFAs characteristics developed in this paper with a reduction example in the $(\mathbb{N} \cup \{+\infty\}, min, +)$ semiring. This example deals with the reduction of an orbit and its connection to the outside. We first reduce the orbit to one state and replace the orbit by this state in the original graph. This new state is then linked to the predecessors (respectively successors) of the orbit with vector Z (respectively Z') as label of edges.

Let G be the $\mathbb{K}$-subgraph of Figure 3.12 and let $\mathcal{O}$ be the only maximal orbit of G such that $X_{\mathcal{O}} = \{a_1, b_2, c_3, a_4, b_5, b_6, c_7\}$.

Algorithm 2 Algorithm for back-edges removal of an orbit.

```
BackEdgesRemoval($\mathcal{O}, T, T', Z, Z'$)
  ▷ Input:  a $\mathbb{K}$-graph $\mathcal{O} = (X_{\mathcal{O}}, U_{\mathcal{O}})$, $M_e \in \mathbb{K}^{|\mathcal{O}^-| \times |In(\mathcal{O})|}$
  ▷ Input:  $M_s \in \mathbb{K}^{|Out(\mathcal{O})| \times |\mathcal{O}^+|}$, $M_{\mathcal{O}} \in \mathbb{K}^{|Out(\mathcal{O})| \times |In(\mathcal{O})|}$
  ▷ Output: $T \in \mathbb{K}^{1 \times |In(\mathcal{O})|}$, $T' \in \mathbb{K}^{|Out(\mathcal{O})| \times 1}$, $Z \in \mathbb{K}^{|\mathcal{O}^-| \times 1}$, $Z' \in \mathbb{K}^{1 \times |\mathcal{O}^+|}$
 1  Begin
 2    for each line $l$ of $M_e$ do
 3      $\mathrm{gcd}_l(l) \leftarrow$ LEFT GCD of all values of the line $l$
    ▷ $\mathrm{gcd}_l$ is the vector of $\mathrm{gcd}_l(l)$ values
 4    Find a vector $\overline{\mathrm{gcd}_l}$ such that $M_e = \mathrm{gcd}_l \otimes \overline{\mathrm{gcd}_l}$
 5    if $\overline{\mathrm{gcd}_l}$ does not exist then
 6      return $False$
 7    for each column $c$ of $M_s$ do
 8      $\mathrm{gcd}_r(c) \leftarrow$ RIGHT GCD of all values of the column $c$
    ▷ $\mathrm{gcd}_r$ is the vector of $\mathrm{gcd}_r(c)$ values
 9    Find a vector $\overline{\mathrm{gcd}_r}$ such that $M_s = \overline{\mathrm{gcd}_r} \otimes \mathrm{gcd}_r$
10    if $\overline{\mathrm{gcd}_r}$ does not exist then
11      return $False$
12    Find $k$ such that $M_{\mathcal{O}} = \overline{\mathrm{gcd}_r} \otimes k \otimes \overline{\mathrm{gcd}_l}$
13    if $k$ does not exist then
14      return $False$
15    $A \leftarrow$ RIGHT GCD of all values of the $\mathrm{gcd}_l$ vector
16    $B \leftarrow$ LEFT GCD of all values of the $\mathrm{gcd}_r$ vector
17    $k_1 \leftarrow$ LEFT GCD$(B, k)$
18    Find $k_2$ such that $k = k_1 \otimes k_2$
19    if RIGHT GCD$(k_2, A) \neq k_2$ then
20      return $False$
21    $T \leftarrow k_2 \otimes \overline{\mathrm{gcd}_l}$
22    $T' \leftarrow \overline{\mathrm{gcd}_r} \otimes k_1$
23    Find $Z$ such that $\mathrm{gcd}_l = Z \otimes k_2$
24    Find $Z'$ such that $\mathrm{gcd}_r = k_1 \otimes Z'$
25    delete any edge from $Out(\mathcal{O})$ to $In(\mathcal{O})$
26    return $True$
27  End
```

We have $M_s = \begin{pmatrix} 1 & 2 & 3 \\ 3 & 4 & 5 \end{pmatrix}$, $M_e = \begin{pmatrix} 4 & 2 & 2 \\ 5 & 3 & 3 \end{pmatrix}$. We can check that $\mathcal{O}$ is

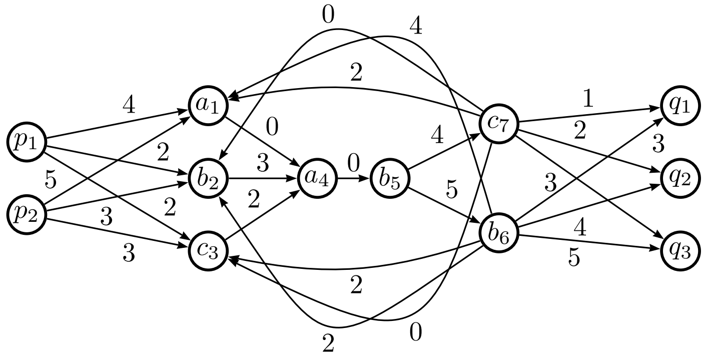
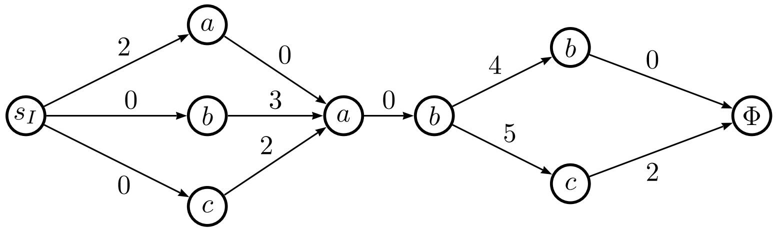


Fig. 3.12 An example for orbit reduction.

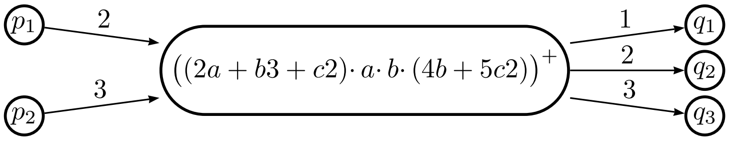
$\mathbb{K}$-transverse. $M_s = \begin{pmatrix} 0 \\ 2 \end{pmatrix} \begin{pmatrix} 1 & 2 & 3 \end{pmatrix} = T'Z'$ and $M_e = \begin{pmatrix} 2 \\ 3 \end{pmatrix} \begin{pmatrix} 2 & 0 & 0 \end{pmatrix} = ZT$.

We then verify that the orbit $\mathcal{O}$ is stable. $M_{\mathcal{O}} = \begin{pmatrix} 2 & 0 & 0 \\ 4 & 2 & 2 \end{pmatrix} = \begin{pmatrix} 0 \\ 2 \end{pmatrix} \begin{pmatrix} 2 & 0 & 0 \end{pmatrix} = VW$. We easily check that the orbit is $\mathbb{K}$-balanced. There is an input vector T which is equal to W and an output vector T' which is equal to V.

Then, we delete back edges and add s_I and Φ vertices for the orbit $\mathcal{O}$. The s_I vertex is connected to $In(\mathcal{O})$. Labels of edges are values of the T vector. Every vertex of $Out(\mathcal{O})$ is connected to Φ. Labels of edges are values of the T' vector. The following graph is then reduced to one state by iterated applications of $\mathbb{K}$-rules.



The expression F associated to this graph is replaced by F^+ and states of $\mathcal{O}^-$ (respectively $\mathcal{O}^+$) are connected to the newly computed state choosing Z as vector of coefficients (respectively Z').



p_1
p_2
2
3
$\left((2a + b3 + c2)\cdot a\cdot b\cdot (4b + 5c2)\right)^+$
1
2
3
q_1
q_2
q_3

3.6 Conclusion

While trying to characterize Glushkov 𝕂-graph, we have pointed out an error in the paper by Caron and Ziadi [6] that we have corrected. This patching allowed us to extend characterization to 𝕂-graph restricting 𝕂 to factorial semirings or fields. For fields, conditions of applications of 𝕂-rules are sufficient to have an algorithm.

For the case of strict semirings, this limitation allowed us to work with GCD and then to give algorithms of computation of 𝕂-expressions from Glushkov 𝕂-graphs .

This characterization is divided into two main parts. The first one is the reduction of an acyclic Glushkov 𝕂-graph into one single vertex labeled with the whole 𝕂-expression. We can be sure that this algorithm ends without doing a depth first search according to confluence of 𝕂-rules. The second one is lying on orbit properties. These criterions allow us to give an algorithm computing a single vertex from each orbit.

In case the expression is not in SNF or the semiring is not zero-divisor free, some edges are computed in several times (coefficients are $\oplus$-added) which implies that some edges may be deleted. Then this characterization does not hold. A question then arises: the factorial condition is a sufficient condition to have an algorithm. Is it also a necessary condition ?

References

1. Berstel, J. and Reutenauer, C. (1988). *Rational series and their languages*, EATCS Monographs on Theoretical Computer Science (Springer-Verlag, Berlin).
2. Brüggemann-Klein, A. (1993). Regular expressions into finite automata, *Theoret. Comput. Sci.* **120**, 2, pp. 197–213.
3. Buchsbaum, A., Giancarlo, R. and Westbrook, J. (2000). On the determinization of weighted finite automata, *SIAM J. Comput.* **30**, 5, pp. 1502–1531.

4. Caron, P. and Flouret, M. (2003). Glushkov construction for series: the non commutative case, *Internat. J. Comput. Math.* **80**, 4, pp. 457–472.
5. Caron, P. and Flouret, M. (2009). Algorithms for Glushkov $\mathbb{K}$-graphs, *CoRR* **abs/0907.4296v1**.
6. Caron, P. and Ziadi, D. (2000). Characterization of Glushkov automata, *Theoret. Comput. Sci.* **233**, 1–2, pp. 75–90.
7. Champarnaud, J.-M., Ouardi, F. and Ziadi, D. (2009). An efficient computation of the equation k-automaton of a regular k-expression, *Fund. Inform.* **90**, 1-2, pp. 1–16.
8. Eilenberg, S. (1974). *Automata, Languages, and Machines*, Vol. A (Academic Press).
9. Glushkov, V. M. (1960). On a synthesis algorithm for abstract automata, *Ukr. Matem. Zhurnal* **12**, 2, pp. 147–156, in Russian.
10. Hebisch, U. and Weinert, H. (1996). Semirings and semifields, in M. Hazewinkel (ed.), *Handbook of Algebra*, Vol. 1, chap. 1F (North-Holland, Amsterdam), pp. 425–462.
11. Hopcroft, J. E. and Ullman, J. D. (1979). *Introduction to Automata Theory, Languages and Computation* (Addison-Wesley, Reading, MA).
12. Kleene, S. C. (1956). Representation of events in nerve nets and finite automata, *Automata studies.*
13. Lombardy, S. and Sakarovitch, J. (2005). Derivatives of rational expressions with multiplicity. *Theor. Comput. Sci.* **332**, 1-3, pp. 141–177.
14. McNaughton, R. F. and Yamada, H. (1960). Regular expressions and state graphs for automata, *IEEE Transactions on Electronic Computers* **9**, pp. 39–57.
15. Sakarovitch, J. (2003). *Éléments de théorie des automates* (Vuibert, Paris).
16. Schützenberger, M. P. (1961). On the definition of a family of automata, *Inform. and Control* **4**, pp. 245–270.

Chapter 4

Natural Language Dictionaries Implemented as Finite Automata

Jan Daciuk

Gdańsk University of Technology, Gdańsk, Poland,
E-mail: `jandac@eti.pg.gda.pl`

Jakub Piskorski

Language Engineerig Group,
Polish Academy of Sciences, Warsaw, Poland,
E-mail: `Jakub.Piskorski@ipipan.waw.pl`

Strahil Ristov

Department of Electronics,
Rudjer Boskovic Institute, Zagreb, Croatia,
E-mail: `ristov@irb.hr`

4.1 Dictionaries as Finite Automata

Dictionaries are widely present in various computer applications, and especially so in Natural Language Processing (NLP). This is similar to a real life situation where a bookshelf or two of various dictionaries is the essential prerequisite for any serious activity involving words. Dictionaries, both in real and computer life, comprise a huge number of entries, be it natural language words, technical terms, facts, numbers or codes. With such a content, there exists an obvious need for the efficient storage and organization principles. Those principles have been established for printed dictionaries over the course of centuries as publishers have optimized the dictionary organization – among the greatest achievements being keeping the entries in a sorted order.

The digital equivalent of a well organized bookshelf volume is an efficient data structure for storing dictionaries. In most applications a huge number of entries have to be stored and accessed. Efficient implementation would be the one that needs the least amount of space and enables as fast access to the information as possible.

The rest of this chapter is organized as follows. First, in the remaining part of Section 4.1 several variants of finite-state devices relevant in the context of implementing NLP dictionaries are described. This includes deterministic and non-deterministic finite-state automata, minimal finite-state automata, tries, recursive automata and transducers. Next, Section 4.2 gives some concrete examples of dictionaries used in various areas of NLP (e.g., morphological analysis and synthesis, spelling correction, information extraction, etc.), and whose implementation is based on finite-state machines. Subsequently, Section 4.3 presents a range of state-of-the-art algorithms for constructing dictionaries from: (a) set of strings, (b) set of strings with cyclic structures, (c) regular expressions, and (d) other sources. In particular, specific variants of these algorithms are introduced too. Section 4.4 deals with models for representing finite-state devices and their time and space complexity, and briefly introduces various techniques for compressing finite-state machines. Some of the latter are particularly useful for compressing NLP dictionaries. Finally, we end with some conclusions and recommendations for further reading in Section 4.4.3.

4.1.1 *Simple and Complex, Static and Dynamic*

Electronic dictionaries can be divided in two general types: simple word lists, and complex dictionaries where each entry is associated with some additional information. Simple word lists can be used to store natural language words, mostly in applications for spelling correction. The complex dictionaries are closer to the common understanding of a term dictionary that usually implies translating from a natural language to another natural language, or explaining the dictionary entries in some way. Although complex dictionaries may include abstract information like routing tables, they are mostly used in natural language applications to store various types of linguistic (semantics, morphology, phonetics) or geographic data (gazetteers, phone books). There exists a certain ambiguity in the literature regarding this, but we will use the term *lexicon* for a simple word list and the term *dictionary* for a proper dictionary – the one with some kind of data associated with a given input key.[1] Depending on the application, the data associated with an input entry can be an explanation, a translation, or some of the various kinds of attributes. A dictionary data structure with keys and attributes is sometimes called the *associative array.*

Formally, a digital dictionary is a data structure that, for a given set of keys K, on a given input entry e, supports operations: membership, add, delete and lookup. Membership operation returns "yes" if e is a member of K and "no" if otherwise. Add operation does nothing if e is already in K and enlarges K with e if it isn't. Delete operation removes e from K if e is a member of K, and does nothing if it isn't. In a proper dictionary, associated attributes are added or deleted simultaneously with the keys. Lookup operation finds and returns the data associated with e, if e is in K, and returns nothing if it isn't. Lookup is sometimes referred to as *mapping* since it maps, or translates, input entry to whatever is the associated data. For instance, "a cat" would be mapped to "un chat" in English to French dictionary, and a number like 127.0.0.1 would translate to "localhost" in a DNS lookup dictionary. The associated data is sometimes called a value, or an attribute. A key can be mapped to more than one attribute.

A lexicon is a structure that supports only membership, add and delete operations since there is no data associated with the keys. Dictionaries that support add and delete operations are called *dynamic* – they allow for online updates on K. If a set of keys is fixed, then the dictionary data structure can be *static*. Static dictionaries allow only for the membership and lookup operations, and only membership is needed for static lexicons.

The advantage of static dictionaries is that they can be compressed more efficiently than the dynamic ones. A lot of research effort has been awarded to the field of a static dictionary compression and there is a considerable difference in the size between the dynamic and static implementations. This is useful in situations where dictionaries are distributed as read-only structures, or in applications involving more or less fixed sets of keys. For example, a comprehensive list of words for a spelling checker is usually a fixed set precompiled in advance, with the additional user-defined words stored separately. If we want to add or delete a key in a static dictionary, we need to recompile it from scratch. However, the state of the art algorithms for producing compressed static dictionaries are very fast and it is conceivable that in some applications even recompiling a static dictionary would pass unnoticed by the user. The substantial difference

[1] The ambiguity stems from the fact that collections called lexicons often contain additional descriptors.

between dynamic and static data structures is that in dynamic ones the procedures for adding a key to the existing structure are the same as those in the initial construction of the dictionary.

4.1.1.1 *Implementing a Dictionary*

Possible implementations of a dictionary data structure include lists, arrays, hash tables, balanced trees and finite state machines.

- Linked lists. The simplest way to implement a dictionary is to store key-attribute pairs as a linked list of records. The list is searched sequentially and, when an input key is matched, we can retrieve the associated attribute for a lookup, return "yes" for a membership query, or delete the record. To add a record, we can simply append it to the end of the list. If we keep the list in a sorted order, then we can use binary search to speed up the search for a matching key, and if there is no match negative result is reported faster. However, adding a new record is slower because we first must find its proper place in the sorted list. The speed of access is the main shortcoming of the list implementation.
- Arrays. A faster solution is to keep entries in an array. If the array is sorted we can use the binary search. In some applications the attribute part is much larger than the key part. In such cases, and especially if the list of key-attribute pairs is too big for the fast memory, we can split records into separate sets for keys and attributes. Then, with each key we have to keep the address of the location where its attribute is stored, and the attribute records can be stored in a slower memory because they are accessed only once per query. When the keys don't vary too much in size, it is convenient to keep them in an array of fixed size records. If the sizes do vary, then we can keep the keys contiguously in an array of variable size records and use a vector of pointers to access the keys. The disadvantage of the array implementation is its size.
- Hash tables. Finding an entry in a sorted array of keys takes logarithmic time if we use binary search. The keys can be accessed in constant time if we use a hash table. The downside again is the size. Hash tables are a classic solution for verifying membership in static lexicons where there is no need to keep the actual key list. In dictionaries where we need to store the keys explicitly

we again need an array for the keys, together with the space for a hash table. Also, an efficient (preferably one-to-one) hash function is comparatively expensive to construct and store.

- Balanced trees. A number of balanced tree structures (red-black trees, B-trees, splay trees) can also be used to store keys. Lookup and update operations on balanced trees require $(n \log n)$ time in both worst and average cases, which is slower than arrays and hash tables. Trees are often used in applications where associated data occupies a lot of space and is stored on secondary memory. The downside is the extra space needed for node pointers.
- Finite-state machines. Finite-state machines (FSM) include *finite-state automata* (FSA) and *finite-state transducers* (FST). Over the years, finite-state machines have emerged as the most efficient way to implement a dictionary. This is mostly due to the advent of algorithms for fast construction and efficient compression. The main advantage of finite machines lies in their time and space efficiency (and a good tradeoff between the two). The processing is linear with the input string length and independent of the size of the whole dictionary. This is better than with lists, arrays and trees, and almost as good as the constant time of hash tables. On the other hand, the implementations of FSMs allow for the use of various compression methods and the requirements for space can be the least of all competitive methods. Another useful property of FSMs is that they are closed under concatenation, union, intersection, difference and Kleene star operations which is beneficial in some linguistic applications.

4.1.2 *Variants of Finite-State Machines Relevant to Dictionary Data Structure Implementation*

Finite-state machines have been extensively studied and a number of different types have been analyzed and explored, each for its own merit. We shall describe some variants that are pertinent to the subject of dictionaries. The basic finite-state machine is the deterministic automaton, so we shall start from there with some groundwork definitions that will be used later in the text. Next, we shall describe a trie and the minimal automaton - which are well known and useful variants of deterministic automata, and a recursive automaton - a concept that enables compact representation. We shall briefly mention non-deterministic automata, and finish the list with transducers - a special type of automata with output.

4.1.2.1 *Deterministic Finite-State Automaton*

We shall begin with the formal definition of *deterministic finite-state automaton* (DFA). A DFA is a quintuple $A = (Q, \Sigma, \delta, q_0, F)$, where Q is a finite set of states, Σ is a finite set of symbols called the *alphabet*, $q_0 \in Q$ is a start state (also called *initial state*), $F \subseteq Q$ is a set of *final* or *accepting* states, $\delta : Q \times \Sigma \Rightarrow Q$ is a transition function. We write $\delta(q, \sigma) = \perp$ is $\delta(q, \sigma) \notin Q$. The extended transition function $\delta^* : Q \times \Sigma^* \Rightarrow Q$ is defined as follows:

$$\delta^*(q, \varepsilon) = q \tag{4.1}$$

$$\delta^*(q, ax) = \begin{cases} \delta^*(\delta(q,a), x) & \text{if } \delta(q,a) \neq \perp \\ \perp & \text{if } \delta(q,a) = \perp \end{cases} \tag{4.2}$$

A *language* $\mathcal{L}(A)$ of a finite automaton A is defined as:

$$\mathcal{L}(A) = \{w \in \Sigma^* : \delta^*(q_0, w) \in F\} \tag{4.3}$$

It is said that an automaton A *recognizes* or *generates* $\mathcal{L}(A)$. A *right language* $\overrightarrow{\mathcal{L}}(q)$ of a state q is defined as:

$$\overrightarrow{\mathcal{L}}(q) = \{w \in \Sigma^* : \delta^*(q, w) \in F\} \tag{4.4}$$

Note that $\mathcal{L}(A) = \overrightarrow{\mathcal{L}}(q_0)$. If, for a symbol σ and a language $\mathcal{L}$, $\sigma\mathcal{L}$ denotes the concatenation of σ and every word in $\mathcal{L}$, then the right language can also be defined recursively with:

$$\overrightarrow{\mathcal{L}}(q) = \bigcup_{\sigma \in \Sigma : \delta(q,\sigma) \in Q} \sigma \overrightarrow{\mathcal{L}}(\delta(q,\sigma)) \cup \begin{cases} \emptyset & \text{if } q \notin F \\ \{\epsilon\} & \text{if } q \in F \end{cases} \tag{4.5}$$

The *left language* of a state is defined as:

$$\overleftarrow{\mathcal{L}}(q) = \{w \in \Sigma^* : \delta^*(q_0, w) = q\} \tag{4.6}$$

For a state q, *fanout*(q) and *fanin*(q) are sets of outgoing and incoming transitions, respectively. With $\Gamma(q)$ we denote a set of labels of outgoing transitions from q. A state q is *reachable* if $\overleftarrow{\mathcal{L}}(q) \neq \emptyset$. If a state is not reachable, it is *unreachable*. A state is *co-reachable* if $\overrightarrow{\mathcal{L}}(q) \neq \emptyset$. A state that is not both reachable and co-reachable is *useless*.

The *size* of an automaton is defined as the number of states:

$$|A| = |Q| \tag{4.7}$$

The number of states is a conventional measure for the automata size. However, in practice, the actual memory consumption of an implementation depends also on the number of transitions and the choice of a data structure.

Let us now explain the defined terms with the help of the example in Figure 4.1. The structure in Figure 4.1 is a deterministic finite automaton. Circles represent states, Q is collection of all of the states and $|Q|$ is 19. State q_0 is the initial state, and states q_3, q_5, q_{10}, q_{13}, q_{17} and q_{18} are the set of accepting states F. Accepting states are drawn with double circles. States q_3, q_{10} and q_{17} are called internal accepting states. Transitions are represented with arrows connecting two states, and the associated symbols are called transition labels. The collection of symbols is the alphabet, i.e. $\Sigma = \{$e, g, h, i, e, o, r, s, t, w$\}$, and the size of the alphabet $\mid \Sigma \mid = 10$. The direction of transitions is indicated with arrows and the collection of transitions represents function δ. It is possible to interpret DFA as a directed graph, then states are nodes and transitions are directed arcs.

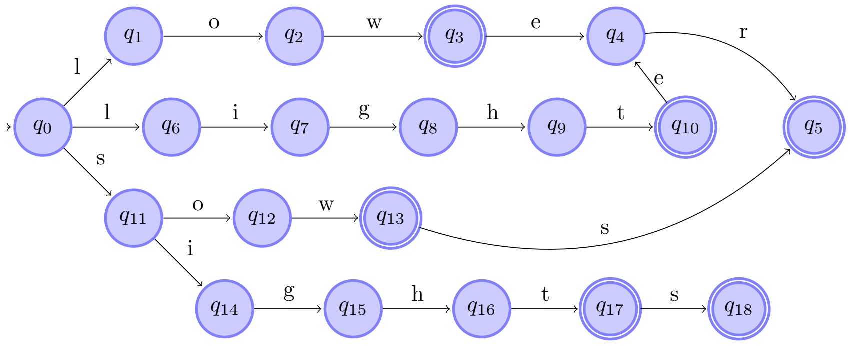


Fig. 4.1 Deterministic finite automaton recognizing words: *low, lower, light, lighter, sow, sows, sight, sights.*

The language of this automaton are eight English words. Membership operation on a DFA is simple. To see if a given input word belongs to a language represented by an automaton such as the one in Figure 4.1, we simply have to start at q_o an test if the symbols in the input word match a concatenation of labels on a path to any of the accepting states.

The automaton in Figure 4.1 has two important properties, it is *deterministic* and *acyclic*. It is deterministic in the sense that there is at most one transition from each state with the same label. In general, finite state automata can have more than one transition with the same symbol from the same state. If this is the case, the automaton is nondeterministic. Nondeterministic automata will be defined later in the text.

If the automaton is acyclic this means that there is no succession of transitions from any state that leads back to the same state. That is, there are no cycles present. The finite state automata are equivalent to regular languages that can in turn be represented by regular expressions. Regular languages are closed under operations of concatenation, union and Kleene star. In regular language (or expression) the star (*) indicates an arbitrary number of repetitions for a given symbol or a string of symbols. In a finite automaton this is represented with a cycle, i.e. a transition (or a sequence of transitions) returning to its originating state and with the label (or a concatenation of labels) consisting of the repeated symbols. Since the number of repetitions is arbitrary, the number of possible strings in such language is infinite. In most cases a dictionary is a finite set of strings and can, therefore, be represented only with an automaton that has no cycles in it. Acyclic automata represent regular languages without star operations. A finite list of strings is a regular language without star operations.

Function δ is a partial function. That means there doesn't need to exist a transition from every state in Q for each symbol in Σ, so the automaton in Figure 4.1 is *incomplete*. In general instance, if δ is a complete function, there should be a transition from each state for every symbol of the alphabet, in this case ten transitions from each state. The transitions that don't belong to a path leading from q_0 to a state in F end in a *reject* state. For the sake of clarity the reject state and transitions leading to it are usually omitted when drawing the automaton. Such automaton is called incomplete.

The paths from q_0 to a state in F form the language accepted by the automaton. Deterministic finite automaton that accepts all strings in a language L is said to recognize L and is often called a *recognizer*. There exist more than one DFA that can recognize the same language. When different DFAs recognize the same language, then we say that they are equivalent.

Deterministic acyclic finite automaton is the fundamental model for representing dictionaries. However, there are various subtypes relevant to dictionary implementation. We shall describe three important variants: a trie, the minimal automaton and the recursive automaton.

4.1.2.2 *Trie*

When a DFA for a given set of strings is constructed as a tree where paths from the root to leaves correspond to input words, and all the prefixes in the set are shared so that branching in a tree occurs only at the beginning of a new suffix, then we call this structure a trie. Identical prefixes of different words are represented with the same node. A trie equivalent to the automaton in Figure 4.1 is presented in Figure 4.2.

Historically, in the early works of de la Briandais, Friedkin and Knuth [21,22,33], nodes of tries would contain the whole of a shared prefix. As the use of the structure evolved, most of the contemporary implementations are done with a single symbol per transition and consequently a trie becomes a tree shaped DFA. This is an elegant and practical data structure that has been widely used to store and search a set of strings. It is easy to construct and update, lends itself to a simple analysis and allows various compression methods. A trie is sometimes called a *digital search tree*.

Dictionary operations on a trie are very fast. Lookup routine branches at each symbol of the input word and membership query takes only as many comparisons as there are symbols in the word. The search procedure takes one symbol of the input word at a time and, starting from the root node, tries to find a transition from the current node labeled with the current input symbol. If there is no matching transition at some point on the path to an accepting state, the input string is rejected. When adding a new string S_i we have to find a prefix of S_i that is already included in the trie. Then there are three possible cases: first, the whole S_i is already in the trie and it ends in an accepting state - this means that trie already accepts S_i and we do nothing; second, S_i is in trie but ends in non-accepting state - then we change the ending state to an accepting one; and, third, if only a prefix of S_i is in the trie, then at the first mismatch we create a new branch for the remaining part of S_i. To delete a string we only have to change the accepting state it ends in to a non-accepting one[2]. All of these operations have time complexity linear with the size of the input string.

[2]The trie can be pruned afterwards, i.e. if there are any useless states they can be removed. This is the case when the deleted word ends in a leaf, then all the states from the leaf upwards to the first branching in a trie are unused.

The exact space and time complexities depend on whether the nodes are implemented as arrays or as lists. If arrays, then the existence of the appropriate transition is verified in a constant time and time complexity of a search is proportional only to the length of the input word. If the transitions are stored in a list then, in the worst case, there can be a transition for every symbol in the alphabet, so, at the worst, the search time is proportional to the length of the input word multiplied with $\mid \Sigma \mid$. If the transitions are stored in an ordered list or a balanced tree, then the multiplying factor is $log \mid \Sigma \mid$.

Construction time of a trie is linear with the size of the input set. When adding a new word we have to match its prefix, symbol by symbol, to an existing path in the structure. On the first mismatch, we have to create new nodes for every remaining character in the word. This means that in total we have to perform exactly the same number of operations as there are symbols in the input word list.

The useful property of a trie is that there is a unique accepting state for each string in the set. This is particularly convenient when storing dictionaries. If keys are stored in a trie then the corresponding attributes can be associated with the accepting states without ambiguities, each key to a separate attribute.

The problem with tries is their size. While space complexity is linear (within a constant of $\mid \Sigma \mid$) with the size of input data, the actual amount of information stored for each node is significant and strongly increases the total space consumption. If transitions in nodes are stored as arrays, then, in the base implementation, there should be space reserved for every symbol in the alphabet. This ads a factor of $\mid \Sigma \mid$ to the total space consumption. It is much more efficient to use lists to store only the existing transitions in each node. Then, in the theoretical worst case, there could still be $\mid \Sigma \mid$ transitions in a node but in practice most of the nodes have much fewer transitions. However, regardless of the implementation, when input sets of strings are large a simple trie can grow to such proportions that its size becomes a restrictive factor in applications.

Witnessing to the fact that, except for the size problem, a trie is a very useful data structure is the great effort that has been directed over the last 20 to 30 years to the research of trie compression. Various methods have been devised for different purposes with appropriate trade-offs between size and search speed. One standard solution is to find the minimal automaton equivalent to the trie.

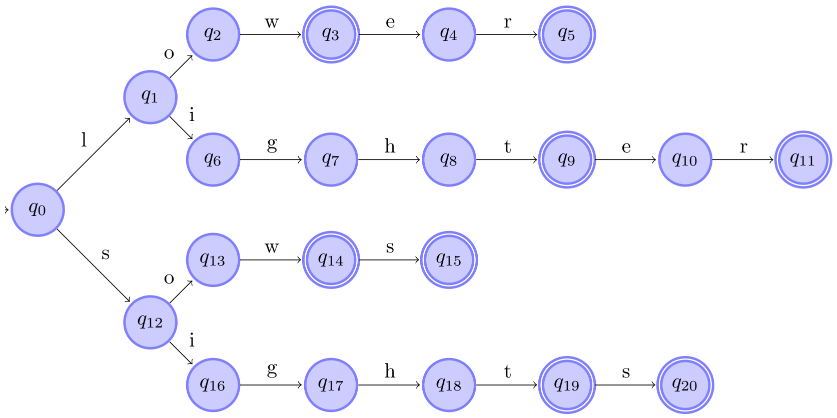


Fig. 4.2 A trie equivalent to DFA in Figure 4.1.

4.1.2.3 *Minimal Deterministic Finite Automaton*

Among all automata that recognize the same language there is one (up to isomorphisms) that has the minimal number of states. It is called the *minimal automaton*. We refer to the minimal DFA as MDFA.

$$\forall_{A:\mathcal{L}(A)=\mathcal{L}(A_{min})} |A_{min}| \leq |A| \tag{4.8}$$

Two states q and p are equivalent, written $q \equiv p$, if they have equal right languages.

$$(q \equiv p) \Leftrightarrow (\overrightarrow{\mathcal{L}}\ (q) = \overrightarrow{\mathcal{L}}\ (p)) \tag{4.9}$$

The equivalence relation divides all states of an automaton into equivalence classes. In a minimal automaton, all classes have exactly one member:

$$\forall_{q,p \in Q_{min}} (q \equiv p) \Leftrightarrow (q = p) \tag{4.10}$$

A minimal automaton has no useless states.

A minimal automaton equivalent to the trie in Figure 4.2 is presented in Figure 4.3. States q_3 and q_9 in trie are equivalent and so are the states q_{14} and q_{19}. Their respective right languages are string *er* and letter *s*. The pairs of equivalent states are merged to new states q_6 and q_{14} in a minimal automaton in Figure 4.3. Also, the final states without the outgoing transitions q_5, q_{11}, q_{15} and q_{20} are merged to one state q_8. Their right language is the empty set. The merging of equivalent states in automaton amounts to the compression of suffixes - identical suffixes are represented with a single instance. This greatly reduces the space consumption when there are many suffix repetitions throughout a word list, as is often the case for natural languages. In natural language applications an acyclic MDFA is sometimes called *directed acyclic word graph* - DAWG. (This is another ambiguous term since originally DAWG was a name for the minimal automaton that recognizes the set of all substrings in a string [5], and only later became associated with the representation of a set of strings [2].)

The best thing about MDFAs is a combination of low space requirements and the speed of construction. The minimization of a DFA has been an important subject for several decades and a classic example of progress in computer science. First algorithms were brute force and quadratic, then came a long time standard $O(n \log n)$ solution of Hopcroft [28] and finally in 1990s several authors have independently devised different variants of algorithms that under certain assumptions have linear time complexity [8, 17, 50]. The boost of interest in efficient algorithms for the minimization of DFAs was the result of their increased use in natural language processing and linguistics that came with the growth of the available textual data in digital form. The huge amount of data on the Internet and various digital libraries underscored the problem of automata size. A trie is a standard intermediate structure in automata minimization algorithms and, with large input sets, a size of trie can quickly grow above the available memory. This is solved with invention of incremental algorithms that don't start with trie as the input but take strings one at the time and output the automaton that is always near minimal [17, 63]. The advances in linear and incremental construction have made MDFA the data structure of choice for storing lexicons. The minimization algorithms are presented in detail in Section 4.3.

One undesirable property of MDFA is that, unlike in trie, the final states are ambiguous. More than one path can lead to a same accepting state. Consequently, it is not possible to put the attributes in final states when constructing a proper dictionary.

This can be solved in two ways, one is to store the whole key-attribute pair as a single string in the automaton, and the second one is to use a special form of hashing. Even if the accepting states are ambiguous, the path to them is unique for each string. It is possible to associate information to that path and in this way obtain perfect (and minimal) hashing function from the automaton and use it to access attributes. This will be elaborated later in Subsection 4.2.1.

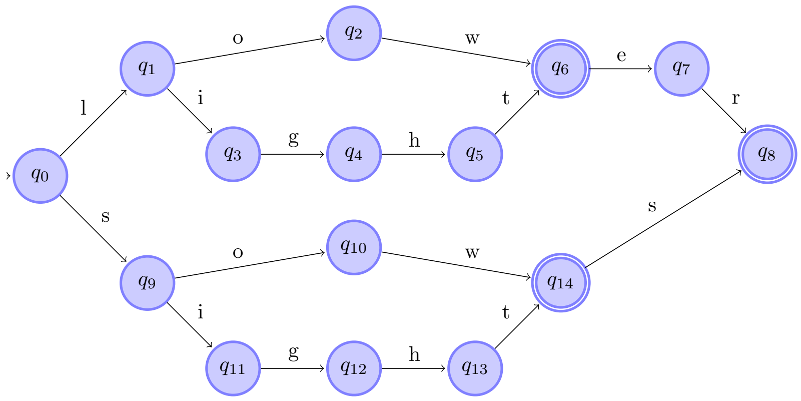


Fig. 4.3 Minimal DFA equivalent to automata in Figures 4.1 and 4.2.

4.1.2.4 *Recursive Automaton*

As a step forward from merging the equivalent states in DFA, it is possible to merge equivalent subautomata in MDFA. In this way we can construct even more compact type of a finite automaton that we call the *recursive automaton* (RA).

A recursive automaton is a sextuple $A = (Q, \Sigma, \delta, r, q_0, F)$, where Q is a finite set of states, Σ is a finite alphabet of symbols , $q_0 \in Q$ is a start state, $F \subseteq Q$ is a set of accepting states, and $\delta : Q \times \Sigma \Rightarrow Q$ and $r : Q \Rightarrow Q$ are partial transition functions. The difference from DFA is the new function $r : Q \Rightarrow Q$ that defines recursive transitions in automaton. Recursive transitions are calls to a subautomaton in A. They are recursive in the sense that calls reference a part of the structure itself. After a call is executed, and a subautomaton traversed, the lookup procedure has to return to the position of the call.

To know where to return the recursive automaton must have some sort of a stack. A stack would be a suspicious property to associate with finite automata since it is normally a part of non-finite push down automata. However, the stack in RA is not unlimited. It is bounded within the factor of the size of the minimal automaton. The recursive calls can be nested - a subautomaton that is a target of a call can include calls to other subautomata. It is easy to show that in the theoretical worst case, when the automaton is produced from one string with an alphabet of a single symbol, the upper limit on the number of nested calls is $log_2 \mid A \mid$. Therefore RA has a finite amount of states and transitions and fits into the class of automata with a finite number of states.

The equivalence of subautomata is formally defined in [23] but, informally, two subautomata SA_1 and SA_2 are equivalent if for every state in SA_1 there is a state in SA_2 with exactly the same left and right languages. Even more informally we can say that equivalent parts of the automaton are exactly those that can be replaced with a call within the automaton. We say this because the equivalence of some of the subparts may be an outcome of the specific implementation. Obviously, the advantage of recursive automata is a reduction of storage space and the actual amount of space that is saved depends strongly on the way that states and transitions are represented. We shall go into it somewhat more in the section on compression 4.4.2.

A recursive automaton equivalent to MDFA in Figure 4.3 is presented in Figure 4.4. The state RC is a recursive call that replaces states from q_9 through q_{14} in MDFA. A recursive call contains an address of the target state and the size of the subautomaton (the number of states to process at the target position). In this case RC has values [1, 6]. The search procedure is the same as in DFA, except that when it reaches a call node it has to continue traversal of the automaton at the target position of a call and, after reading the indicated number of nodes, return to the position of a call.

Again, as in MDFA the final states are ambiguous, and we can not put the attributes in them, but the path to reach them is unique for each word and this can be used for hashing.

4.1.2.5 *Nondeterministic FA and ε — NDFA*

Besides deterministic, finite automata can also be nondeterministic. This means that there can be more than one transition with the same label from one state leading to different target states. The nondeterministic automata are less efficient when implementing dictionaries because of the need for backtracking. However, they can be more space efficient than the deterministic variant, and they can be useful in automata construction algorithms. We shall therefore define them here.

A *nondeterministic finite-state automaton* (NDFA) is a quintuple $A = (Q, \Sigma, \delta, q_0, F)$, where again Q is a finite set of states, Σ a finite alphabet,

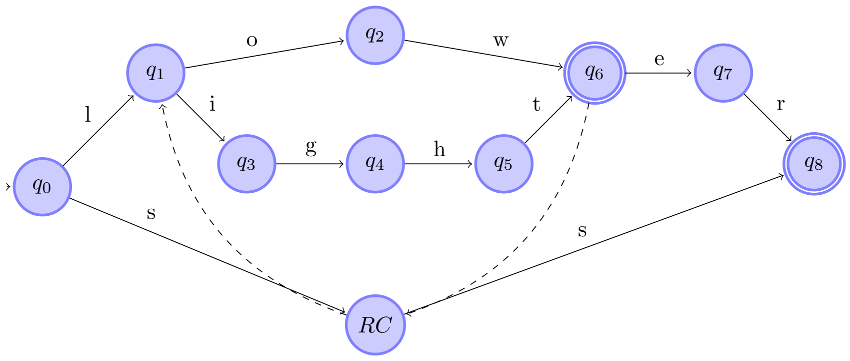


Fig. 4.4 Recursive automaton equivalent to automata in Figures 4.1, 4.2 and 4.3.

$q_0 \in Q$ is the initial state, $F \subseteq Q$ is a set of final states, but the transition function is now $\delta : Q \times \Sigma \Rightarrow 2^Q$. 2^Q denotes a set of subsets of Q. We use this to indicate that the target of transition function δ can, on the same input symbol, be not only one state but a set of states that are a subset of Q.

If we allow the existence of the empty transition that is standardly denoted with ε then the transition function becomes $\delta : Q \times (\Sigma \cup \{\varepsilon\}) \Rightarrow 2^Q$. This means that the automaton can change a state without an input symbol. Such an automaton is called ε - nondeterministic finite-state automaton (ε - NDFA). It is worth noting that for every DFA there exist equivalent NDFA (or ε - NDFA) and vice-versa.

4.1.2.6 *Transducer*

$$\begin{tikzcd} \arrow[r] & q_0 \arrow[r, "r{:}c"] & q_1 \arrow[r, "u{:}o"] & q_2 \arrow[r, "n{:}urir"] & \boxed{q_3} \end{tikzcd}$$

Fig. 4.5 Sequential transducer. Translates *run* to *courir*.

$$\begin{tikzcd} \arrow[r] & q_0 \arrow[r, "r{:}c"] & q_1 \arrow[r, "u{:}o"] & q_2 \arrow[r, "n{:}ur"] & \boxed{q_3} \arrow[r, "ir"] & {} \end{tikzcd}$$

Fig. 4.6 Subsequential transducer. Translates *run* to *courir*.

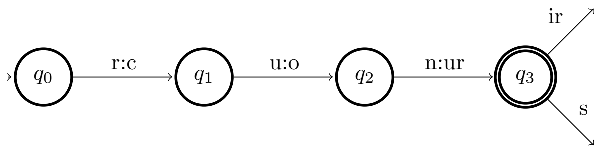


Fig. 4.7 p-subsequential transducer. Translates *run* to *courir* and *cours*.

Another class of useful finite-state machines are transducers. They are finite machines that, besides input alphabet and language, have output alphabet and output language so that they translate (or transduce) input strings to output strings. The definition is as follows. A *sequential* transducer that translates input string to output string is a septuple $A = (Q, q_0, F, \Sigma, \Delta, \delta, \sigma)$. As before Q, q_0 and F are set of states, initial state and set of final states, respectively. Σ is the finite alphabet of input strings and Δ is the finite alphabet of output strings. $\delta : Q \times \Sigma \Rightarrow Q$ is the input state transition function and $\sigma : Q \times \Sigma \Rightarrow \Delta^*$ the output function. We say that a transducer is sequential if it has deterministic input, that is, there is at most one transition with the same input symbol from each state. This is analogous to determinism in ordinary finite automata. An example of a simple sequential transducer is presented in Figure 4.5.

A *subsequential* transducer $A = (Q, q_0, F, \Sigma, \Delta, \delta, \sigma, \phi)$ has an additional element, a final output function $\phi : F \Rightarrow \Delta^*$ that maps the set of final states to words in output alphabet Δ^*. Subsequential transducer can have a single extra output string at each final state (see Figure 4.6).

A transducer is *p-subsequential* if ϕ maps F to $(\Delta^*)^p$. This means that it can have up to p additional output strings at the final states. An example of 2-subsequential transducer is shown in Figure 4.7. This transducer translates English verb *run* to the infinitive and one of the imperative forms of the equivalent verb in French.

Transducers represent mappings from input to output string and are, as such, the natural model for proper dictionaries where the set of keys can be the input language and attributes are the output. p-subsequential transducers have the ability to associate more than one attribute to a single input so they are especially well suited for natural language applications.

4.1.2.7 *Implementing Dictionaries with FSMs*

The choice of which type of machine to use for an implementation of a dictionary depends on the dictionary in question, i.e. on the characteristics of data. The principal concerns in data storage and retrieval are the speed of search, speed of construction, and the amount of used space. All variants of finite-state machines are very fast structures and the search speed is sufficient regardless of which type is used. The remaining concerns are the construction speed and the space. If dictionaries are static then the speed of construction is also irrelevant because they are precompiled in advance. Then the choice of data structure is governed only by the usage of space.

If we need to store a static lexicon then MDFA and RA are the best choices. RA uses less space but for some type of data the difference is not worth the additional effort. A recursive automaton built from a set of English words will not be much smaller than the corresponding MDFA. On the other hand, in the case of words belonging to an inflected language, such as are many Romanic and Slavic languages, the reduction in the automaton size becomes considerable. If the lexicon needs to be frequently updated, and is not too big, a trie can be a viable choice. If the size is a problem then one can use MDFA and an incremental minimization algorithm.

A proper dictionary with key and attribute pairs can be stored in two general manners. One is to concatenate pairs of keys and corresponding attributes in single strings in the form of *key:attribute* (with a separator between to distinguish them) and then to treat the strings as simple lexicons and use the appropriate data structures.

The second way is to store keys and attributes as separate entities. For that purpose we can use a transducer with keys as the input and attributes as the output language. If there is more than one attribute associated with the same key we can use p-subsequential transducer. The other way is to store key and attribute sets in separate automata. Then we have to preserve the *key* $\Leftrightarrow$ *attribute* mapping in form of links, or indices, that have to be stored in some additional structures. This last approach leads to the most space efficient representation of a static dictionary as will be described in Section 4.4.2.

4.2 Automata as Mappings

Dictionaries (implemented as automata) can be seen as mappings between words and some information. In the simplest case, a dictionary is a list of words $W \in \Sigma^*$. As such, it implements a mapping between words and $\{\text{false}, \text{true}\}$, or a relation that is a subset of $\Sigma^* \times \{\text{false}, \text{true}\}$. When a word is stored in a dictionary, it is mapped to true, otherwise it is mapped to false. Section 4.2.1 describes mappings between words and numbers. In Natural Language Processing, dictionaries are often used for morphological analysis and synthesis. Such use is described in Section 4.2.2. Another NLP application –spelling correction introduced in Section 4.2.3– implements a mapping between words that are not in a dictionary and a set of subsets of words contained in the dictionary. Finally, gazetteers are a special purpose dictionaries described in Section 4.2.4.

4.2.1 *Perfect Hashing*

With a simple modification deterministic machines can perform a two-way perfect hashing function that maps each word of input language to a different integer, and vice-versa. Perfect hashing maps each key to a different number. There are no conflicts and hence no need for conflict resolution strategies as with ordinary hash functions. If there are M input words, the integers will be in range from 1 to M so the function is minimal. Furthermore, it is ordered, i.e. the order of input words is reflected in associated numbers: if $word_i$ is stored in the automaton before $word_j$, then $\text{mapping}(word_i) < \text{mapping}(word_j)$. Therefore, a modified deterministic automaton can perform *two-way ordered minimal perfect hash function*.

This modified automaton has been proposed in [49], where it is called a *hashing transducer*, and in [39], where it is referred to as a *numbered automaton*. To add hashing functionality to an automaton, an integer is associated with each transition or with each state. If it is in a transition, this number indicates how many different paths, that are leading to some accepting state, can be extended following the current transition. If the number is stored in states, it denotes how many words belong to the right language of that state. An example of an automaton with numbers stored in transitions is presented in Figure 4.8, and the same automaton, but with numbers in states, is shown in Figure 4.9. The automaton is minimal and its language are four words (*abc, abcde, abdef, acdef*) stored in alphabetical order. We shall describe how the automaton maps these four words to numbers 1 to 4 (and numbers to words) on the example with numbers in transitions. The same logic is easily applied to the case with numbers in states.

During construction of a numbered automaton we have to add a counter to every transition and increment it every time a new word uses that transition. This produces the automaton in Figure 4.8. It is easy to see that, for example, there are four words using transition t_1 and two words using t_3. There is only one word ($abcde$) using transitions t_4 and t_5 since word abc ends in q_3.

Let us assume we want to find a number i that is a mapping for an input $word_i$. We have to set up a counter C, initialize C to 1, and then traverse the path in automaton that accepts $word_i$. While traversing, C is incremented in two cases:

- On each branching increment C by the value of how many words are stored in branches that are skipped;
- When passing through accepting state increment C by one.

This effectively counts how many words are stored in the automaton before $word_i$, and, at the point when a word is accepted, C has the value of word's ordinal number in the input set. Let us see how this works on our example in Figure 4.8. Word abc is the first word we encounter when traversing the automaton. There are no skipped branches nor accepting states before state q_3 so the value of C is 1 when abc is accepted. To reach the state q_5 that accepts the second word $abcde$ we have to traverse the accepting state q_3 at which point C is incremented by one. There are no more branches nor accepting states before q_5 so at that point C has a value of 2. To traverse the path that leads to the accepting state for the third word $abdef$ we have to follow transition t_8 at branching in q_2. There we have to increment C by 2, which is the value stored in skipped transition t_3. There are no more branches nor accepting states on the way to q_5, so the value of C is 3 when the third word is accepted. Similarly, to reach the accepting state q_5 with $acdef$, the fourth word in the set, we have to skip transition t_2 in state q_1. At that point we increment C by 3, the value stored in t_2, and when fourth word is accepted C has a value of 4.

The inverse operation is finding a word through its ordinal number. We shall denote the value V of number stored in transition i with V_i. Then, finding i^{th} word in an automaton, with a given integer i, is conducted as follows:

- Counter C is set to i.
- Automaton is traversed starting from the initial state, and transitions in states are examined one by one. Value V_i in transition t_i is compared to C:

– If $V_i < C$, t_i is skipped and C is decremented by V_i;
– If $V_i > C$, t_i is followed to the next state. If the next state is accepting, C is decremented by 1;
– If $V_i = C$, t_i is followed to the next state. If the next state is accepting, C is decremented by 1. If at that point $C = 0$, the concatenation of labels on the path we followed to reach the current accepting state is i^{th} word in the set.

To illustrate this procedure let us again turn to the example in Figure 4.8. Say we want to find the third word in the automaton. C is set to 3 and, from q_0, we follow through transitions t_1 and t_2 since their values V are larger or equal to C. At state q_2, the first transition t_3 has value $V = 2$ which is less than C. We decrement C by 2 and skip t_3. Now, with $C = 1$ we examine the next transition t_8 and find that $V = C$. We follow the sequence of transitions t_8, t_9 and t_{10}, that all have $V = 1$, to the accepting state q_5. At q_5 the counter is decremented by 1 to the value of 0, and, concatenating the labels on the path we took to reach q_5, we obtain *abdef*, the third word in the set. One more example: To find the second word we set $C = 2$ and follow transitions t_1, t_2 and t_3 to state q_3 where C is decremented by 1. Then, in transitions t_4 and t_5 values of V are again equal to C, up to state Q_5 where C is set to 0. Concatenating the labels on transitions t_1 through t_5, we get the second word *abcde*.

A variant with V values in states instead of transitions can be utilized in a similar manner. Note, however, a slightly different assignment of values in two implementations. The technique of numbering automata to implement minimal perfect hashing can be applied to any shape of deterministic FA, including minimal, trie or recursive automaton. In Section 4.2.4 a more concrete example of using numbered automata will be given (in particular, see Figure 4.15).

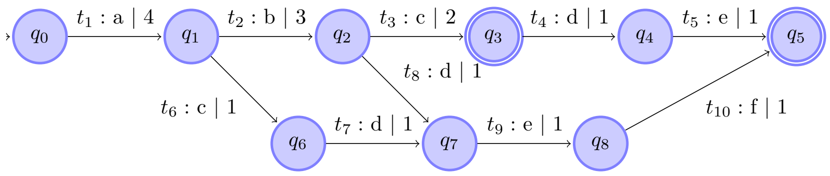


Fig. 4.8 Numbered automaton with numbers in transitions.

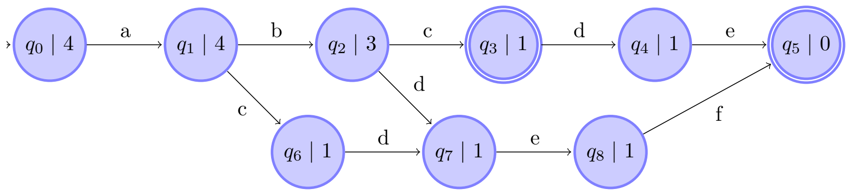


Fig. 4.9 Numbered automaton with numbers in states.

When the mapping from words to numbers is dynamic, the numbered automata described so far here cannot be used as the mapping changes with every word added or deleted. Fortunately, there are a few solutions to this problem. The first one is to use a vector that translates hash values to static numbers.

A more elegant solution is to use pseudo-minimal automata [41]. If input strings always contain an end-of-string symbol, such automata always have a proper transition for each word belonging to the language of the automaton. This transition is traversed only when recognizing a single word. Construction of pseudo-minimal automata is very much similar to the construction of minimal automata described in Section 4.3.1. States can be merged when they are equivalent and cardinality of their right languages is at most one. Details of the algorithms can be found in [15] and [16].

4.2.2 *Morphological Analysis and Synthesis*

Morphological analysis and synthesis can be performed with either transducers or finite-state automata. Figure 4.10 shows a transducer that is capable of performing a simplified form of morphological analysis of present forms of the French word *aimer*. The analysis is done by following the input labels, and gathering output labels. For example, the form *aime* has two possible analyses: *aimer+1s* and *aimer+3s* (the canonical form is *aimer*, and the inflected forms are the first or the third person singular).

Looking at the picture, several observations can be made:

(1) When the input string and the output string are of different lengths, ε has to be used to indicate the absence of either an input or output symbol.
(2) Backtracking is used to retrieve additional, alternative analyses.
(3) Morphological synthesis can be performed by following the output labels and gathering the input labels.
(4) Backtracking may be needed also when only a single result exists. For

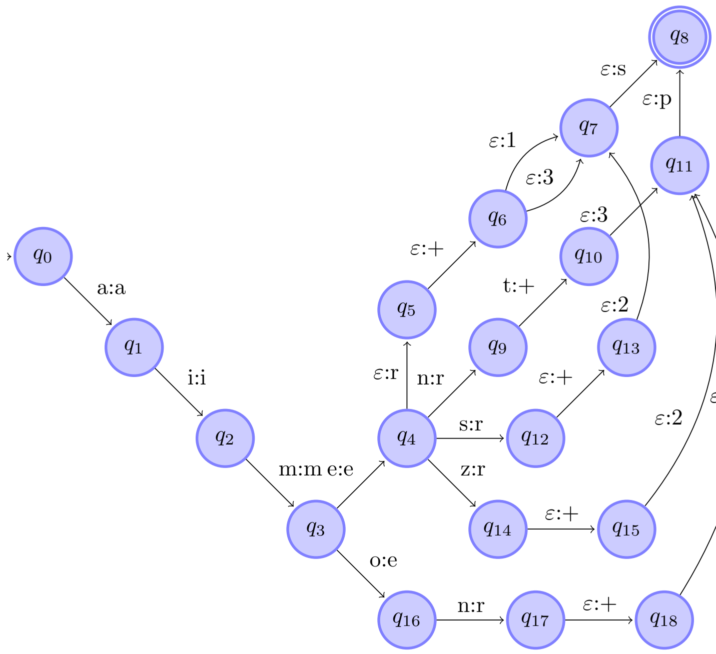


Fig. 4.10 Transducer that performs (simplified) morphological analysis of present forms of the French verb *aimer*.

example, when performing the synthesis of *aimer+1p* and following appropriate outgoing transitions from left to right, we would have to visit most states of the transducer. Transducers can be determinized in Mohri's sense, i.e. the output labels can be pushed farther away from the start state [44]. For example a part of an inverted transducer from Figure 4.10 may look as in Figure 4.11. When the input string is *aimer+1*, we still do not know whether the output will be *aime* or *aimons*, i.e. whether to output *e* or *o*. With *aimer+2* or *aimer+3*, the next output symbol must be *e*.

When all output labels are pushed after all input labels, a transducer is no longer needed. A simple finite-state automaton is sufficient. Each string collected from labels along any path in the automaton from the start state to any of the final states consists of the input string, followed by a separator, and followed by a string representing the output. However, when the morphological analysis returns also canonical forms, a naive implementation would lead to huge automata as they would contain long chains of states

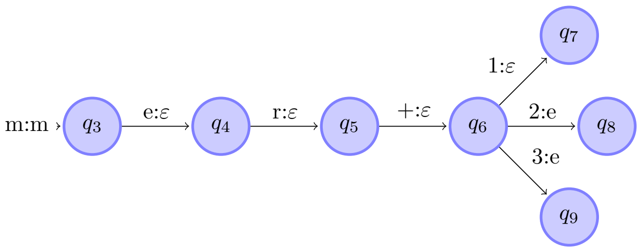


Fig. 4.11 A part of a determinized transducer that performs (simplified) morphological synthesis of present forms of the French verb *aimer*.

with a single incoming transition and a single outgoing transition for each dictionary entry. That would happen because in each entry, only a prefix of the inflected word, and a suffix of the canonical form along with associated representation of morphological information (we would call it *annotation* here, but it can also be called *tags* or *categories*) could be shared between different entries.

Instead of representing the canonical forms in full, a much better method is to code the difference between the inflected form and the canonical in an entry. One way of doing that is to replace the canonical form with a letter that codes how many characters from the end of the inflected form are to be deleted, followed by the ending of the canonical form. More formally, let $i = sf$ be an inflected form, and $c = sb$ – a canonical form, with $s = i \wedge c$ being the longest common prefix. We replace c with $\mathrm{chr}(\mathrm{ord}('A') + |f|)b$, where $\mathrm{chr}(x)$ returns a character that has the code x, and $\mathrm{ord}(y)$ returns the code of the character y. For example, *aimer+3p* –a result of morphological analysis of *aimons*– would be coded as *Der+3p*. Figure 4.12 shows a complete automaton equivalent to that in Figure 4.10. Note that representing the canonical form outside the automaton, and using a numbered automaton to provide an index in that external storage would require more memory.

It is possible to produce a dictionary containing all words that may appear in texts only for a limited domain. Unrestricted texts may contain additional words, and it may be necessary to analyze them morphologically. There are at least two ways to do that using finite-state machines. The first one –guessing automata– is simpler to implement. It can be used when we only have dictionary data, i.e. words with their annotations, and not a full morphological description with concatenation and orthographic rules.

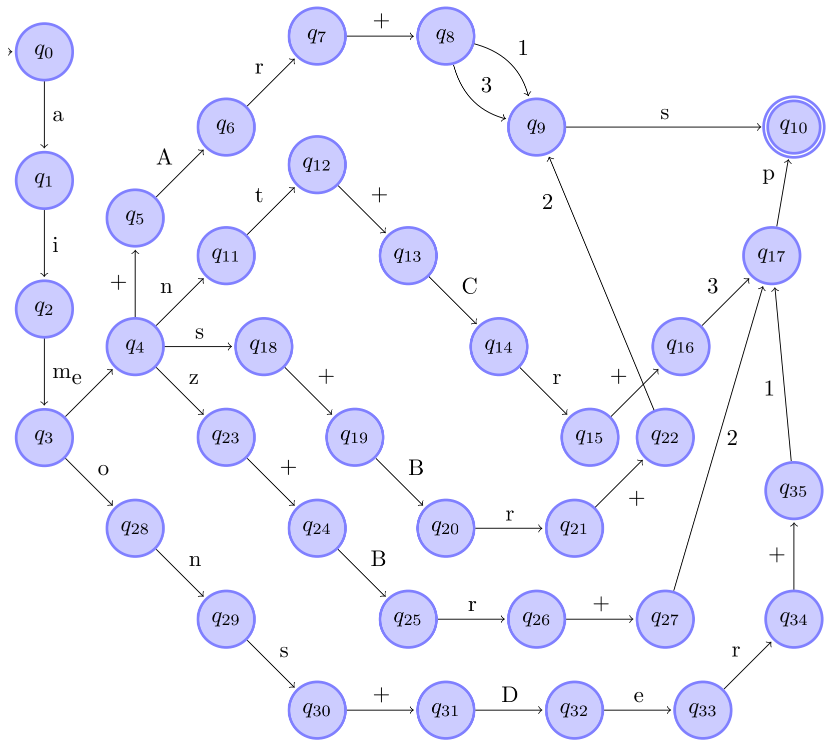


Fig. 4.12 A DFA that can perform the same morphological analysis as the transducer in Figure 4.10. Note that generation is not possible with the same automaton.

People can usually guess morphological categories and canonical forms from endings of words. For example, if we were to classify present forms of French regular verbs of the first group, then we could guess that *travaillons* is a present, plural, second person form of *travailler*. As this can be guessed from the ending, a guessing automaton should associate endings with categories. It should analyze words from the end, therefore the words in the guessing automaton should be reversed. For example, the data from our tiny morphological dictionary of French would contain the following strings:

```
emia+Ar+1s
emia+Ar+3s
semia+Br+2s
snomia+Der+1p
```

```
tnemia+Cr+3p
zemia+Br+2p
```

The automaton built from such strings is depicted in Figure 4.13. When an unknown word is analyzed, it is reversed, and searched for in the automaton. Since the word is not in the dictionary, the search will end at a state with no outgoing transition labeled with an annotation separator. Then, paths are followed that are labeled with symbols that are not the annotation separator. Those paths lead to one or more states that have outgoing transitions labeled with the annotation separator. Those transitions start paths leading to some final states. By concatenating labels on those paths we obtain results of morphological analysis.

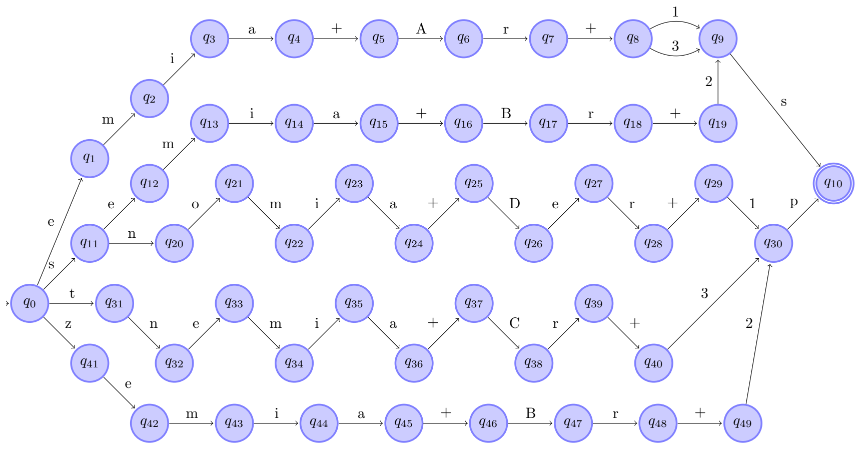


Fig. 4.13 An automaton built from strings containing reversed inflected forms and annotations. The data comes from a morphological dictionary depicted as finite-state machines in Figures 4.10 and 4.12.

Note that the results are known earlier. As it can be seen in Figure 4.13, it is sufficient to recognize one of the endings *e, es, ns, t* and *z*. Longer endings do not contribute any more information. States $q_2 \dots q_4$, $q_{13} \dots q_{15}$, $q_{21} \dots q_{24}$, $q_{32} \dots q_{36}$, and $q_{42} \dots q_{45}$ are redundant. Removing them gives a guessing automaton shown in Figure 4.14.

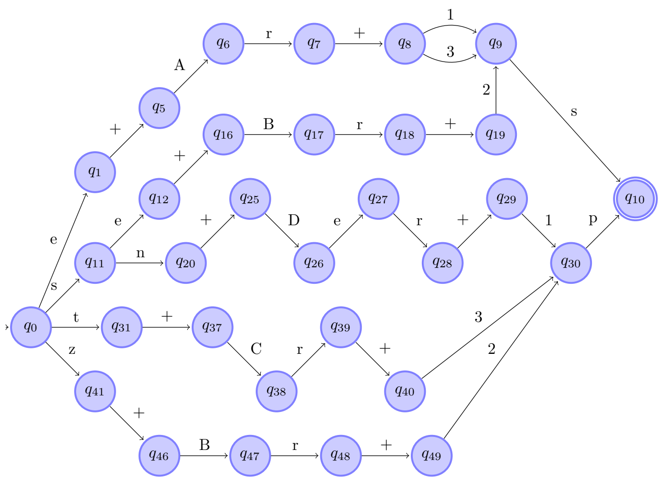


Fig. 4.14 A guessing automaton obtained from the automaton in Figure 4.13 by pruning redundant states and transitions.

4.2.3 *Spelling Correction and Restoration of Diacritics*

Correct spelling has always been difficult for many people. The problem was made more acute with the advent of television and video: people read less books and newspapers, so they are less exposed to correct spelling. There are various causes of spelling errors. They include medical conditions, lack of education, but also recently, a decline in reading. There are various types of spelling errors:

- Orthographic errors – resulting from lack of knowledge of orthography,
- Typographical errors – caused by lack of precision in typing,
- Other errors – e.g. repetition of words, incorrect morphology, etc.

Fortunately, a widespread use of computers brings hope to those who want to write correctly. Incorrect words may be detected because they are not present in the dictionary, or they may be detected because they are not correct in the place where they were written, i.e. their context is wrong. In either case, search for the correct word is done using a dictionary. Therefore, we focus our attention on isolated-word correction. For a thorough investigation on types of spelling errors, and correction techniques, see [38].

Damerau [20] distinguishes four types of simple transformations that are applied one or more times to produce an erroneous word from the correct one:

- Insertion of a letter,
- Deletion of a letter,
- Change of one letter into another,
- Transposition of letters.

The transformations are reversible. Their reversals can be applied to an incorrect form to produce a correct one. If a word is transformed to a string that is also present in a dictionary, i.e. to a word that was not meant by the writer, the error can only be detected using the context of the word. However, if the transformation leads to a string that cannot be found in the dictionary (presumably to a non-word), then it is assumed that the writer meant a similar word present in the dictionary. The similarity is measured by an *edit distance*. In its general form, the edit distance between strings X and Y is calculated as:

$$ed(X_{1\ldots i+1}, Y_{1\ldots j+1}) = \begin{cases} ed(X_{1\ldots i, Y_{1\ldots j}}) & \text{if } x_{i+1} = y_{j+1} \\ min \begin{pmatrix} TC + ed(X_{1\ldots i-1}, Y_{1\ldots j-1}), \\ DC + ed(X_{1\ldots i+1}, Y_{1\ldots j}), \\ IC + ed(X_{1\ldots i}, Y_{1\ldots j+1}) \end{pmatrix} & \begin{array}{l} \text{if } x_i = y_{j+1} \\ \wedge\ x_{i+1} = y_i \end{array} \\ min \begin{pmatrix} RC + ed(X_{1\ldots i}, Y_{1\ldots j}), \\ DC + ed(X_{1\ldots i+1}, Y_{1\ldots j}), \\ IC + ed(X_{1\ldots i}, Y_{1\ldots j+1}) \end{pmatrix} & \text{otherwise} \end{cases} \tag{4.11}$$

If the last letters of X and Y are the same, then the value of the edit distance is the same as without the last letters, as seen in the first line of Equation (4.11). If the last two letters of Y are the last two letters of X transposed, then three cases are possible:

(1) The basic operation was transposition with the cost TC. The edit distance is the sum of TC and the edit distance of X and Y both shortened by two characters.
(2) The basic operation was deletion with the cost DC. The edit distance is the sum of DC and the edit distance of two strings Y shortened by one character and X.
(3) The basic operation was insertion with the cost IC. The edit distance is the sum of IC and the edit distance of two strings X shortened by one character and Y.

Values of TC, DC, and IC can be constant, or they can be functions of the characters taking part in the operations, and also of the context, i.e. the characters preceding or following those taking part in the operations, and the positions of the characters. If the last two letters of X are not transposed last letters of Y, nor the last letter of X is the last letter of Y, then we have a similar situation as the one above, with two out of three cases (deletion and insertion) the same as described above, and with replacement (change of one letter into another) with cost RC replacing transposition.

When TC, DC, IC, and RC are constants all equal one, Equation (4.11) takes the form of Equation (4.12) as in [47] and [46]:

$$ed(X_{1...i+1}, Y_{1...j+1}) = \begin{cases} ed(X_{1...i,Y_{1...j}}) & \text{if } x_{i+1} = y_{j+1} \\ 1 + min\begin{pmatrix} ed(X_{1...i-1}, Y_{1...j-1}), \\ ed(X_{1...i+1}, Y_{1...j}), \\ ed(X_{1...i}, Y_{1...j+1}) \end{pmatrix} & \begin{matrix}\text{if } x_i = y_{j+1} \\ \wedge\ x_{i+1} = y_i\end{matrix} \\ 1 + min\begin{pmatrix} ed(X_{1...i}, Y_{1...j}), \\ ed(X_{1...i+1}, Y_{1...j}), \\ ed(X_{1...i}, Y_{1...j+1}) \end{pmatrix} & \text{otherwise} \end{cases} \tag{4.12}$$

Border conditions apply:

$$\begin{aligned} ed(X_{1...0}, Y_{1...j}) &= j & \text{if } 0 \leq j \leq |X| \\ ed(X_{1...i}, Y_{1...0}) &= i & \text{if } 0 \leq i \leq |Y| \\ ed(X_{1...-1}, Y_{1...j}) &= ed(X_{1...i}, Y_{1...-1}) = \max(|X|, |Y|) & \end{aligned} \tag{4.13}$$

Note that $X_{1...0} = Y_{1...0} = \varepsilon$. The first two lines of Equation (4.13) express the fact that transforming an empty string ε into a sequence of n symbols, or transforming a sequence of n symbols into an empty string ε requires n insertions or deletions respectively. In the last line of Equation (4.13), -1 occurs as transposition requires checking the penultimate symbol in a string. In the general case, the values in the border conditions must be multiplied by insertion cost.

Using the edit distance, an incorrect form should be compared to all words in the dictionary to find the best match, i.e. words that lie not farther than a given threshold value t of the edit distance, usually one. A list of plausible corrections of an incorrect string w is given as:

$$\mathcal{C} = \{w_i \in L : ed(w, w_i) \leq t\} \tag{4.14}$$

However, in most cases, it is not necessary to compare whole words. For example, if we compare *locomotive* to *engine*, we don't need to check all letters of both words to know that their edit distance is greater than one. A few initial letters are enough. Therefore, instead of checking the edit distance of whole strings, we check their initial segments, and stop as soon as the distance becomes too big. A *cut-off edit distance* between an initial segment of string X of length m and initial segment of Y of length n with the threshold edit distance t is defined as:

$$cuted(X_{1\ldots m}, Y_{1\ldots n}) = \min_{l \le i \le u} ed(X_{1\ldots i}, Y_{1\ldots n}) \tag{4.15}$$

where $l = \min(1, n-t)$, and $u = \max(m, n+t)$. Note that as we use dictionaries in form of finite-state machines, an initial segment of a word is shared in the dictionary with other words. By abandoning a path leading to a word with an edit distance above the threshold, we abandon checking many words.

Implementing equations given in this chapter in a straightforward manner would lead to repeated computation of the same values – a situation often found in computer science. A popular solution to this problem is the use of dynamic programming. Results of partial computations are saved in table D and recalled later when needed. An edit distance $ed(X_{1\ldots j}, Y_{1\ldots i})$ is saved as $D[i,j]$.

Restoration of diacritics can be viewed as a special case of spelling correction, where only one basic editing operation –replacement– is allowed. In addition a letter can only be replaced with a letter based on the same Latin letter, but with a diacritic.

4.2.4 *Gazetteers and Information Extraction*

A gazetteer is a dictionary of geographical names, where each of them is usually associated with some related information. For instance, country names might be associated with: known abbreviations (e.g. *UK* stands for *United Kingdom*), social statistics, full name (e.g. *Peoples Republic of China* is a full name of *China*), the makeup of a country, etc. In the NLP field, lists of person names, organizations and other type of named entities are also referred to as gazetteers. For instance, organization names in such gazetteers are associated with their abbreviations, full names, organization type, location of their headquarters, etc. Probably the best way to define gazetteers is to see them as mappings from named entities to the properties of the real-world objects these named entities refer to. Gazetteers are widely deployed in information retrieval and information extraction applications, in particular, for matching names in texts and for enriching these texts with semantic annotations.

There are several ways of implementing gazetteers by means of finite-state automata. Before delving into the details, let us note that raw gazetteer resources are usually represented by a text file, where each line of such a file represents a single entry in the following format: `keyword (attribute:value)+`, i.e., each name is associated with a list of arbitrary attribute-value pairs. Let us consider as an example two gazetteer entries for the word *Bayern*.

```
Bayern | type:region | full-name:Freistaat Bayern
       | location:Germany | subtype:state
Bayern | type:org | full-name:FC Bayern
       | location:Germany | subtype:football_club
```

The first entry refers two a state, whereas the second refers to a football club. In gazetteers, attribute values may be string valued, numerical or mixed. Frequently, string-valued attributes are not inflected or derived forms of the keyword as demonstrated in the above example (e.g., the attribute `location`).

Turning such data, as in the example above, into a single MDFA by treating each single entry as a single path in an automaton [37] would not yield a good compression. In the following two methods ways of using DFAs for implementing gazetteers are briefly sketched.

4.2.4.1 *Pure DFA Approach*

The first technique is based on reorganising and transforming gazetteer entries in such a way that suffix sharing is maximized. To be more precise, each gazetter entry is split into a disjunction of subentries, where each such subentry represents some partial information. In order to describe this process, let us differentiate between open-class and closed-class attributes depending on their range of values, e.g., `full-name` and `location` in the above example are open-class attributes, whereas `type` and `subtype` are closed-class attributes. For each open-class attribute-value pair present in an entry, a single subentry is created, whereas closed-class attribute-value pairs are merged into a single subentry and rearranged in order to fulfill the *first most specific, last most general* criterion [10]. In our example, for the two entries for *Bayern* we get the following six subentries (three subentries for each entry):

```
Bayern #1 NAME(subtype) VAL(state) NAME(type) VAL(region)
Bayern #1 NAME(full-name) Freistaat Bayern
Bayern #1 NAME(location) Germany

Bayern #2 NAME(subtype) VAL(football_club) NAME(type) VAL(org)
Bayern #2 NAME(full-name) FC Bayern
Bayern #2 NAME(location) Germany
```

Here, `NAME` and `VAL` map attribute names and values of the closed-class attributes into single univocal characters representing them. The tags `#1` and `#2` denote the interpretation ID of the keyword *Bayern* (state vs. football club). Subsequently, we can observe that some attribute values could be derived from the keyword by performning some edition operations. Therefore we replace these values by application of formation patterns (analogously as in the case of morphological dictionaries discussed earlier). For instance, *Freistaat Bayern* can be derived from *Bayern* by the insertion of the word *Freistaat* in front of the word *Bayern*. Applying formation patterns to the entries in our example yields:

```
Bayern #1 NAME(subtype) VAL(state) NAME(type) VAL(region)
Bayern #1 NAME(full-name) Freistaat $(ID)
Bayern #1 NAME(location) Germany

Bayern #2 NAME(subtype) VAL(football_club) NAME(type) VAL(org)
Bayern #2 NAME(full-name) FC $(ID)
Bayern #2 NAME(location) Germany
```

In particular, `$(ID)` is a unique symbol representing a formation pattern, which simply implements the identity function (keyword is copied). As can be observed, such representation allows for better suffix sharing. The formation patterns used for encoding attribute values in the context of gazetteer entries resemble the ones for encoding morphological information, but they partially rely on other information. For instance, frequently, attribute values are just the capitalized form or the lowercase version of the corresponding keywords. Next, patterns for forming acronyms or abbreviations from the full form are applied, e.g., *ACM* can be derived from *Association for Computing Machinery*, by simply concatenating all capitals in the full name. Nevertheless, some part of the attribute values can not be replaced by patterns.

Once tha gazetteer entries are 'transformed' into subentries in the fashion described above the rest boils down to constructing an MDFA from the set of such subentries, through applying the algorithms presented in Section 4.3. The states having outgoing transitions labeled with the unique symbols in the range of `NAME` are implicit final states. The right languages of these states represent attribute-value pairs attached to the gazetteer entries. A comprehensive description of the outlined technique is given in [48].

4.2.4.2 *Indexing Automaton Approach*

The second method for modelling gazetteers is an adaptation of the standard approach to implementing general dictionaries and thesauri presented in [26,36]. The main idea is to encode the keywords and all attribute values in a single numbered automaton (see Section 4.2.1), which we call an indexing automaton. In order to distinguish between keywords and different attribute values the indexing automaton has $n + 1$ initial states, where n is the number of attributes. The strings accepted by the automaton starting from the first initial state (q_0) correspond directly to the set of the keywords, whereas the right language of the i-th initial state, namely q_i (for $i \geq 1$), corresponds to the range of values appropriate for i-th attribute. Furthermore, the subautomaton starting in each initial state implements different perfect hashing function. Hence, the aforementioned automaton constitutes a word-to-index and index-to-word engine for keywords and attribute values. Once the index of a given keyword is known, the indices of all associated attribute values in a row of an auxiliary table can be accessed. Consequently, these indices can be used to extract the proper values from the indexing automaton. In the case of multiple readings an intermediate array for mapping the keyword indices to the absolute position of the block of rows containing all readings of a given keyword is indispensable. The overall architecture is sketched in Figure 4.15 which is accompanied by the pseudocode of the algorithm for accessing attribute values for a given keyword.

It is not necessarily convenient to index all attribute values and store the proper values in the numbered automaton, e.g., numerical data such as longitude or latitude could be stored directly in the attribute-value matrix since automata are not the best choice in such situation. Alternatively, some attribute values could also be stored elsewhere (as depicted in Figure 4.15). This procedure is reasonable if the range of the values is bounded and integer representation is more compact than anything else (e.g. long alphanumeric identifiers). Fortunately, the vast majority (but definitely not all) of attribute values in a gazetteer deployed in NLP happens to be natural language words or multi-word expressions. Therefore, one can intuitively expect the major part of the entries and attribute values to share suffixes, which leads to a better compression of the indexing automaton. The prevalent bottleneck of the presented approach is a potentially high redundancy of the information stored in the attribute-value matrix. However, this problem can be partially alleviated via automatic detection of column dependency, which might expose sources of information redundancy to gain better compression of the data [61]. Reccurring patterns consisting of raw fragments of the attribute-value index matrix could be indexed and represented only once.

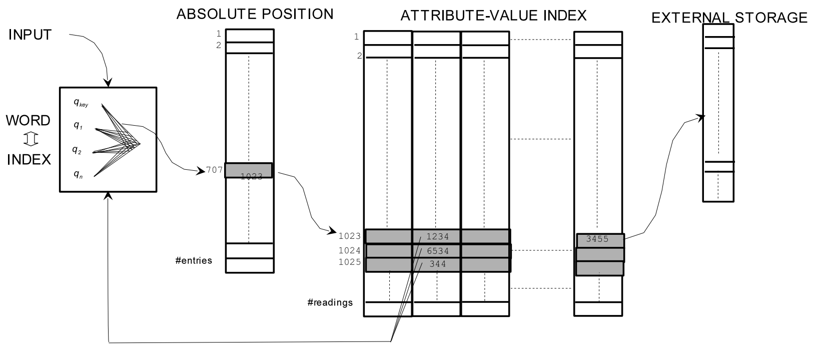


Fig. 4.15 Compact storage model for a gazetteer look-up component based on numbered automata. The pseudocode of the algorithm for accessing the value of i-th attribute for j-th interpretation of the keyword *word* is given in Algorithm 1. First, *word* is mapped to its numerical ID via a call to WORDTOINDEX function which also takes as an argument the start state from which index is computed (line 2). Next, the way in which values of the i-th attribute are stored is computed (line 3). There are three possibilities to do it: (a) directly in the ATTRIBUTE-VALUE matrix, (b) indirectly in the numbered automaton, (c) externally in EXTERNAL-STORAGE array. Finally, the attribute value is computed (lines 4-12).

4.3 Construction Methods

Deterministic finite automata representing dictionaries can be constructed from various data with a variety of methods. The construction method depends on the input data. In the simplest case, which is one of the most common ones, a dictionary stores a collection of strings. The corresponding

Algorithm 1 The pseudocode of the algorithm for accessing the value of i-th attribute for the j-th interpretation of the keyword *word*.

```
 1: function GetValue(word,j,i)
 2:     $id$ = WordToIndex($word$,$q_{key}$)
 3:     $howIsValueStored$ = HowIsValueStored($i$)
 4:     $val = ATTRIBUTE\text{–}VALUE[ABSOLUTE\text{–}POSITION[id] + j][i]$
 5:     if $howIsValueStored = directly$ then
 6:         return $val$
 7:     else
 8:         if $howIsValueStored = usingNumberedAutomaton$ then
 9:             return IndexToWord($val$,$q_i$)
10:         end if
11:     end if
12:     return $EXTERNAL\text{–}STORAGE[val]$          ▷ value is stored externally
13: end function
```

automaton is acyclic. For such data, incremental algorithms described in Section 4.3.1 offer great speed combined with small memory requirements. If the dictionary must have some cyclic core (e.g. for productive compounding), and a collection of strings, then it is possible to modify the algorithms from Section 4.3.1 to add strings to the language of an already cyclic automaton. The modifications are given in Section 4.3.2. When we know how to construct automata that recognize parts of the target language, we can combine them using algorithms described in Section 4.3.3. Other methods are mentioned in Section 4.3.4.

4.3.1 *Construction from Strings*

Traditional construction of dictionaries in form of finite automata from strings involves constructing a *trie* (a tree where the edges are labeled with single characters), and then performing minimization using one of well known general minimization algorithms. Although such method is correct, the size of the intermediate product –the trie– can be enormous. While contemporary computers have increasingly bigger operational memories, the input data can also grow. Also, it does not make much sense to use huge amounts of memory when it is possible to use a much smaller amount of it without any speed penalty.

Incremental methods keep the size of the automaton close to the minimal one during all stages of the construction process. Each time a string is added to the language of the automaton, the automaton is either minimized, or brought close to minimal. There are two main algorithmic approaches to incremental construction of minimal, deterministic, acyclic, finite automata from strings. The first one requires data to be lexicographically sorted. The other does not have that limitation, but it is slower.

In both algorithms, an important issue is minimization. If for a state q, an equivalent state r can be found, then q can be replaced by r, i.e. state q is deleted, and all transitions that led to q are redirected to r. The question is how to find an equivalent state. Let us recall the recursive definition of the right language of a state (Equation (4.5) on page 138):

$$\overrightarrow{\mathcal{L}}(q) = \bigcup_{\sigma\in\Sigma:\delta(q,\sigma)\in Q} \sigma \overrightarrow{\mathcal{L}}(\delta(q,\sigma)) \cup \begin{cases} \emptyset & \text{if } q \notin F \\ \{\epsilon\} & \text{if } q \in F \end{cases} \tag{4.16}$$

Two states q and p are equivalent (written $q \equiv p$) when their right languages are equal:

$$q \equiv p \Leftrightarrow \begin{pmatrix} \Gamma(q) = \Gamma(p) \wedge \\ \forall_{\sigma\in\Gamma(q)}(\overrightarrow{\mathcal{L}}(\delta(q,\sigma)) = \overrightarrow{\mathcal{L}}(\delta(p,\sigma))) \wedge \\ (q \in F) \Leftrightarrow (p \in F) \end{pmatrix} \tag{4.17}$$

Automata in form of trees are acyclic. In acyclic automata, it is possible to visit states in such an order that all states reachable from a given state have already been visited before the state is visited. One possible order of this type is *postorder*. If the targets of all transitions going out from state q have already been visited and replaced with equivalent states when possible, then if two states in the already processed part have the same right language, they must be the same state. Under those conditions:

$$q \equiv p \Leftrightarrow ((\text{fanout}(q) = \text{fanout}(p)) \wedge ((q \in F) \Leftrightarrow (p \in F))) \tag{4.18}$$

We have two processes: one process that adds strings to the language of an automaton so that it creates a trie, and another one that visits states and replaces them with equivalent ones if they can be found in the part of the automaton that has already been minimized. The processes need to synchronized. There are two possibilities. The first one is to minimize

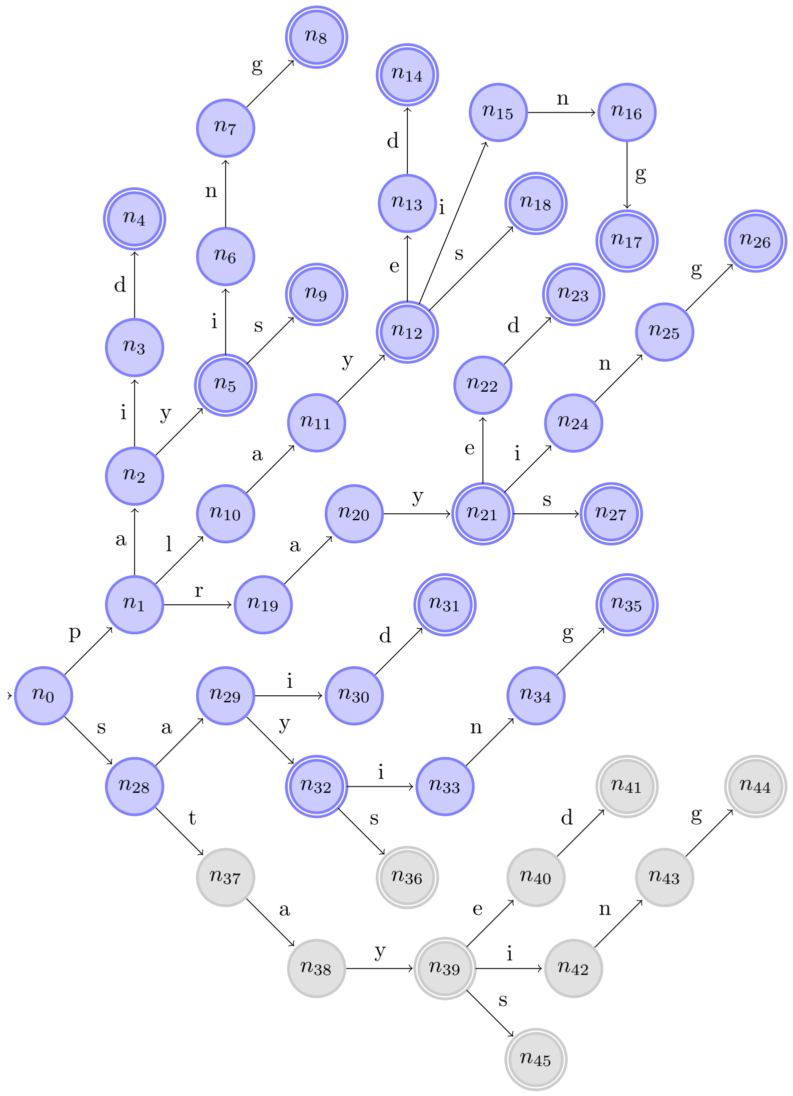


Fig. 4.16 A trie recognizing inflected forms of words *pay*, *play*, *pray*, *say* and *stay*.

those states of the automaton that will not change their language when new words are added – this leads to the algorithm for sorted data. The second one is to minimize the automaton completely each time a new strings is added – this leads to the algorithm for unsorted data.

The first question to be asked is which states will not change their right language when new strings are added to the language of the automaton. In Figure 4.16, all inflected forms of words *pay, play, pray, say*, and *stay* are recognized by an automaton in form of a trie. States and transitions printed in normal intensity are those that depict an automaton recognizing the forms: *paid, pay, pays, paying, pray, prayed, praying, prayed, said, say, saying*. Forms *says, stay, stayed, staying*, and *stays* that come later in lexicographical order are printed in gray. A look at the figure reveals that states n_0, n_{28}, n_{29}, and n_{32} will get additional outgoing transitions as a result of adding the additional words. The states all lie on a path of states $(n_0, n_{28}, n_{29}, n_{32}, n_{33}, n_{34}, n_{35})$ that are visited while recognizing the last string that has been added. Only those states can change their right language as a result of adding *any* new words that lexicographically follow the last word added to the automaton. Let w be the last word added to the automaton, $|w|$ its length, and v a new word that follows w in lexicographical order $(w < v)$. Then $\exists_{0 \leq n \leq |w|} \forall_{1 \leq i \leq n} w_i = v_i$ and either $n = |w|$ or $w_{n+1} < v_{n+1}$.

Each time a string is added, local minimization follows. It affects only those states that have not already been processed before. In Figure 4.16, the last string added is $w =$*saying*. The string to be added immediately after that is $v =$*says*. The states $(n_0, n_{28}, n_{29}, n_{32}, n_{33}, n_{34}, n_{35})$ form a path that is followed while recognizing w. States $(n_0, n_{28}, n_{29}, n_{32})$ are also in the path for v as the longest common prefix of w and v written $w \wedge v$ is *say*, and $\delta^*(n_0, say) = n_{32}$. It should be noted that $w \wedge v = \mathcal{L}(A) \wedge v$. States n_{33}, n_{34} and n_{35} are in the path for w but not in the path for v, so they are subject to local minimization. States n_{33}, n_{34} and n_{35} recognize a suffix of w. Note that until the longest common prefix is recognized, the suffix that determines which states undergo local minimization is not yet known. Also note that the suffix can be empty. The string added immediately before w was $u =$*say*. As $u \wedge w = u$, there are no states to undergo minimization after v has been added.

The skeleton of the algorithm is given as Algorithm 2. The variable u represents a string that was added to the language of the automaton in the previous cycle. In line 6, the longest common prefix is traversed. The code in that line is used for its brevity. An actual implementation would rather test for outgoing transitions with labels equal to subsequent symbols of the string. The call to LocMin in line 13 is necessary as the path recognizing the last string added to the automaton did not yet undergo minimization.

Function AddSuffix is trivial. It creates a chain of states and transi-

Algorithm 2 Skeleton of the algorithm for sorted data.

```
1: function SortedConstruction
2:     Create empty $A = (\{q_0\}, \Sigma, \emptyset, q_0, \emptyset)$
3:     $u \leftarrow \epsilon$; $R \leftarrow \emptyset$
4:     while input not empty do
5:         $w \leftarrow$ next string from input
6:         $v \leftarrow u \wedge w$; $p \leftarrow \delta^*(q_0, v)$  ▷ Traverse the longest common prefix
7:         if $|u| > |v|$ then
8:             $\delta(p, u_{|v|+1}) \leftarrow$ LocMin$(A, R, \delta(p, u_{|v|+1}), u_{|v|+2\ldots|u|})$
9:         end if
10:        AddSuffix$(A, p, w_{|v|+1\ldots|w|})$
11:        $u \leftarrow w$
12:    end while
13:    LocMin$(A, R, q_0, u)$
14:    return $A$
15: end function
```

Algorithm 3 Create a chain of states and transitions starting from state q so that $\delta(q, w) \in F$. Parameter A is the automaton.

```
1: function AddSuffix(A, q, w)
2:     while $w \neq \epsilon$ do
3:         $p \leftarrow$ new state
4:         $\delta(q, w_1) \leftarrow p$
5:         $q \leftarrow p$; $w \leftarrow w_{2\ldots|w|}$
6:     end while
7:     $F \leftarrow F \cup \{q\}$
8: end function
```

tions so that $\delta^*(q, w) \in F$. Note that in the initial call to AddSuffix in line 10 of Algorithm 2, the suffix is never empty.

Function LocMin takes four parameters. The first two are are the automaton A and a register R (more on the register later in this paragraph). The third one is a starting state q of a chain of states to be minimized. The fourth one is a string w so that subsequent symbols of w are labels of outgoing transitions of q and subsequent states reachable by those transitions. As we use equation (4.18) to compare states for equality, LocMin must be called recursively to start equality testing from the last state of the chain, i.e. the state $\delta(q, w)$. The equivalent state is searched for in the part

Algorithm 4 Perform local minimization on a chain of states starting from q and reachable with any prefix of w. Return q or an equivalent state. A is the automaton, R is the register. A and R are modified by the function.

```
1:  function LocMin(A, R, q, w)
2:      if $w \neq \epsilon$ then
3:          $\delta(q, w_1) \leftarrow \textsc{LocMin}(A, R, \delta(q, w_1), w_{2\ldots|w|})$
4:      end if
5:      if $\exists_{r \in R} r \equiv q$ then
6:          delete $q$
7:          return $r$
8:      else
9:          $R \leftarrow R \cup \{q\}$
10:         return $q$
11:     end if
12: end function
```

of the automaton that has already been minimized. Comparing the states one-by-one would be very ineffective. The equation (4.18) can be used not only to directly compare right languages of two states. It can also be used in implementing a hash function on right languages of states. A hash function is computed using outgoing transitions (labels and targets) as well as finality of the state itself. A hash table storing states in the minimized part of the automaton is called the *register*. It appears as parameter R in the function LocMin. If an equivalent state r for state q can be found, state q is deleted, and state r is returned. The returned value is used in redirecting the transition to state r either in line 3 of LocMin or in line 8 of the main algorithm. Note that only one transition needs to be redirected as there is only one transition leading to state q. More incoming transitions can only appear as the result of minimization. As the state r is already in the register, there is no need to put it there again. If a state equivalent to state q cannot be found, then q is put into the register R, and q is returned, which means no redirection.

The main loop in Algorithm 2 runs n times, where n is the number of strings read from the input. Traversal of the longest common prefix takes $\mathcal{O}(|w_{max}|)$, where w_{max} is the longest string. Function AddSuffix recursively calls itself $|w|$ times per input string, where $|w|$ is the length of a suffix, which itself is never longer than w_{max}. Function LocMin is called recursively $|w|$ times per input string, where w is the suffix determining the path to be minimized. As before $|w| \leq |w_{max}|$. In each call, a state is searched for in the register, and if not found, it is put there. The time complexity of the operations on the register depends on its implementation. As it is possible to implement a hash table so that a lookup and insertion take constant time on average, we assume constant time. Under this assumption, the complexity of the whole algorithm is $\mathcal{O}(n)$, with n meaning the number of strings, as $|w_{max}|$ is also constant.

In the algorithm for sorted data, the automaton is brought very close to minimality after addition of each string. When a new string is read, and when the longest common prefix is traversed in line 6 of Algorithm 2, the automaton is minimal except for a chain of states recognizing the last part of the string added to the automaton. That chain is processed in line 8 so that after that call, the automaton is minimal. A call to AddSuffix in line 10 creates another chain of states such that the states are not unique.

Let us go through an example to see how the algorithm works. Suppose that our input consists of the words *pay, paying, pays, play, playing,* and *plays.* We create an empty automaton, and initialize the register R, and the previous word u. The main loop of Algorithm 2 reads *pay* from the input. As the previous word is ε, the longest common prefix is also ε, and we stay in the initial state $q_0 = n_0$. From the same reason, LocMin is not called. AddSuffix creates a chain of states and transitions to recognize *pay.* The situation is depicted in Figure 4.17.

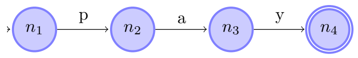


Fig. 4.17 Sorted construction – string *pay* added with AddSuffix.

In the next cyclce of the main loop of Algorithm 2, *paying is read.* The longest common prefix of *paying* and *pay* is *pay*, so p becomes n_4, and as *pay* is a prefix of *paying*, LocMin is again not called. AddSuffix adds more states and transitions to the automaton, creating a new automaton shown in Figure 4.18.

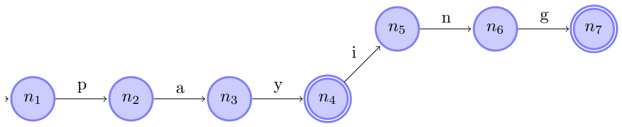


Fig. 4.18 Sorted construction – string *paying* added.

The next word read is *plays*, *plays* $\wedge$ *playing* = *play*, so p becomes n_4, and LocMin is called with the automaton A, the empty register R, the state n_5, and the suffix *ng*. The function calls itself recursively in line 3 of Algorithm 4 with the first two parameters staying the same, and with tyhe last two being first n_6 and g, and then n_7 and ε. As $w = \varepsilon$, the last call puts n_7 into R, and returns n_7. As n_6 and n_5 have different right languages, they are also put into the register and returned. Now SortedConstruction calls AddSuffix with A, n_4 and s. The latter function creates a final state n_8 and a transition from n_4 to n_8 labelled with s. The result can be seen in Figure 4.19, $R = \{n_5, n_6, n_7\}$.

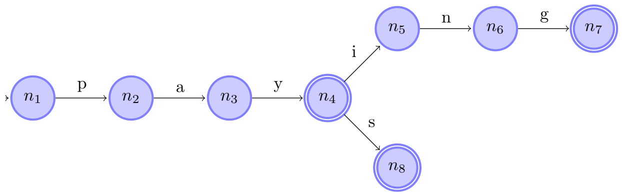


Fig. 4.19 Sorted construction – string *pays* added.

SortedConstruction reads *play*, the longest common prefix is v=p, and the state $p = n_2$. As the previous word u is longer than v, LocMin is called with A, $R = \{n_5, n_6, n_7\}$, n_3, and *ys*. The function calls itself twice with n_4 and s, and then with n_8 and ε as the last two parameters, finds the state $n_7 \in R$ to be equivalent to n_8, so n_8 is deleted. The function returns n_7, so the transition from n_4 with the label s is redirected to n_7. There is no state equivalent to n_4 in the register, so n_4 is added to R; the same applies to n_3, which is returned by the top-level invocation of LocMin. Then AddSuffix, called with A, n_2, and *lay* as parameters, creates a chain of states and transitions that recognize that suffix, as depicted in Figure 4.20.

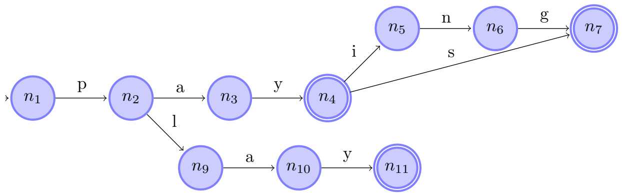


Fig. 4.20 Sorted construction – string *play* added.

The next word to be added is *playing*. The longest common prefix v is the same as the previous word u=*play*. It leads to the state $p = n_{11}$. LocMin is not called, and a call to AddSuffix creates a chain of states and transitions that recognize the suffix *ing* as shown in Figure 4.21.

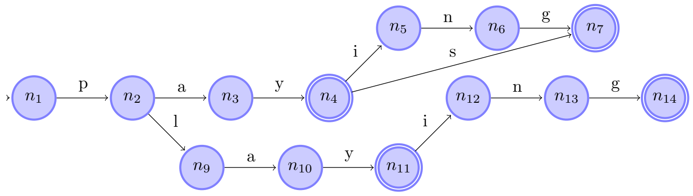


Fig. 4.21 Sorted construction – string *playing* added.

Finally, we add *plays*. The longest common prefix –*play*– leads to n_{11}. LocMin is called with A, with $R = \{n_1, n_2, n_3, n_4, n_5, n_6, n_7\}$, with state n_{12}, an with *ng*. Recursive calls lead to n_{14} with $w = \varepsilon$. State n_{14} has an equivalent state $n_7 \in R$. State n_{14} is deleted, and its incoming transition is redirected towards n_7. Similarly, state n_{13} is deleted, and its incoming transition redirected towards n_6, and state n_{12} is deleted, and state n_5 is returned instead by LocMin. The incoming transition of state n_{12} is redirected towards n_5 in line 8 of Algorithm 2. Afterwards, AddSuffix creates state $_{15}$, and a transition from n_{11} to n_{15} labelled with *s*. The situation is showed in Figure 4.22.

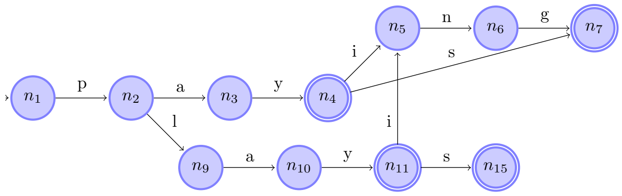


Fig. 4.22 Sorted construction – string *plays* added.

As there are no more words to be read, the main loop of Algorithm 2 is terminated, and LocMin is called again in line 13. Its arguments are the automaton A, the register R, which contains all state of the automaton except those that lie on the path that recognizes the last word added, the initial state n_1, and the last word added *plays*. The function calls itself recursively, until it reaches n_{15} with $w = \varepsilon$. As n_{15} is equivalent to n_7, n_{15} is deleted, and its incoming transition redirected towards n_7. Similarly, n_{11} is deleted, and its incoming transition is redirected towards n_4, and n_{10} is deleted with its incoming transition redirected towards n_3. State n_9 has a unique right language, so it has no equivalent state in R, and it is added to the register. The same applies to n_2 and n_1. We get a minimal automaton with all states in the register. The final, minimal automaton is shown in Figure 4.23.

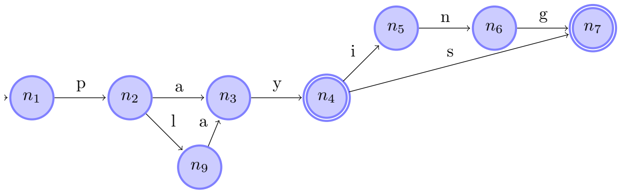


Fig. 4.23 Sorted construction – the minimal automaton.

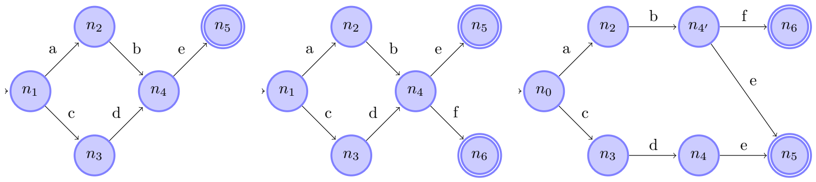


Fig. 4.24 The automaton on the left recognizes the language *abe, cde*. When we naively add the string *abf* by creating a state n_6 and a transition from state n_4 to state n_6 with label *f*, we inadvertly also add *cdf* as it can be seen in the middle. The automaton on the right recognizes *abe, abf*, and *cde*.

The second approach to incremental construction is to minimize the whole automaton completely each time a new string is added. Contrary to the algorithm for sorted data, where each states first is created as part of a trie, and then undergoes local minimization once, some states may be processed many times during construction. One of the results of minimization is redirection of transitions so that a single state can have more than one incoming transition. Adding an outgoing transition to such state (we call it a *confluence state*) may result in adding more than one string to the language of the automaton. Figure 4.24 shows an example. The reason why the automaton in the middle contains not only *abf*, but also *cdf*, is that state n_4 is reachable from the start state not only by following transitions labeled with a and then b, but also with c and then d. State n_4 represents two vertices of a trie. To add *abf* and only *abf* to the language of the automaton, only one of the vertices should change its right language by creating an outgoing transition. Therefore, the vertex needs to be separated from other vertices before a suffix is added to the language of the automaton. The separation is done by *cloning* the state. Cloning means creating an exact copy of the state. The clone has the same finality and the same suite of outgoing transitions. In Figure 4.24, the transition from n_2 to n_4 has to be redirected to $n_{4'}$, which is a clone of n_4.

Algorithm 5 Skeleton of the algorithm for unsorted data.

```
1: function UnsortedConstruction
2:     Create empty $A = (\{q_0\}, \Sigma, \emptyset, q_0, \emptyset)$
3:     $R \leftarrow \emptyset$
4:     while input not empty do
5:         $w \leftarrow$ next string from input
6:         UnsortedAdd$(A, R, w)$
7:     end while
8:     return $A$
9: end function
```

Algorithm 5 shows the skeleton of the algorithm for unsorted data. The algorithm starts with an empty automaton and an empty register. Each call to UnsortedAdd with a new string w leaves with the automaton being minimal and recognizing w in addition to its previous language, and with the register containing all states of the automaton.

Algorithm 6 Add a string to the language of the automaton using the unsorted construction. A is the automaton, R is the register, w is the string to be added. A and R are modified by the procedure.

```
1: procedure UnsortedAdd(A, R, w)
2:     $(P, i, j) \leftarrow$ TraverseLCP$(A, R, q_0, w)$
3:     $q \leftarrow P_i$
4:     AddSuffix$(A, q, w_{i \ldots |w|})$
5:     $q \leftarrow$ LocMin$(A, R, \delta(q, w_j), w_{j+1 \ldots |w|})$
6:     MinimizeBackwards$(A, R, P, j)$
7: end procedure
```

Algorithm 6 shows the function UnsortedAdd. First, function TraverseLCP is called to traverse a path in the automaton recognizing the longest initial part of w that is shared with some other string already in the automaton. The function returns the path and the number of items in it, as well as an index of the first state in the path that was removed from the register. It also creates clones of states when necessary, so that each state in the path P has only one incoming transition. The register R is a parameter of TRAVERSELCP because the function removes from the register a state that will change its suite of outgoing transitions either as a result of cloning, or as a result of a call to ADDSUFFIX in line 4. We will return to ADDSUFFIX to explain the rest of the function after we describe the function TRAVERSELCP.

Algorithm 7 Traverse the longest common prefix, cloning confluence states on the way. A is the automaton, R is the register, p is the start state of the traversal, w is the string that selects the path. Return a triple consisting of a path of states visited during the traversal, of an index of the last item on the path, and of an index of the state that was removed from the register.

```
 1: function TRAVERSELCP(A, R, p, w)
 2:     $q \leftarrow p$; $i \leftarrow 1$; $P_i \leftarrow q$
 3:     while $(i \leq |w|) \wedge (\delta(q, w_i) \in Q) \wedge (|\mathrm{fanin}(\delta(q, w_i))| = 1)$ do
 4:         $q \leftarrow \delta(q, w_i)$; $i \leftarrow i + 1$
 5:         $P_i \leftarrow q$
 6:     end while
 7:     $R \leftarrow R \setminus \{q\}$                    ▷ $q$ will change its right language
 8:     $j \leftarrow i$                                   ▷ $j$ is the index of $q$
 9:     while $(i \leq |w|) \wedge (\delta(q, w_i) \in Q)$ do
10:         $r \leftarrow \mathrm{clone}(\delta(q, w_i))$; $\delta(q, w_i) \leftarrow r$; $i \leftarrow i + 1$
11:         $q \leftarrow r$; $P_i \leftarrow q$
12:     end while
13:     return $(P, i, j)$
14: end function
```

Function TRAVERSELCP shown as Algorithm 7 has four arguments: A – the automaton, R – the register, p – the initial state of the automaton, w – the string to be added. The automaton and the register can be modified by the function. Subsequent symbols of the string are recognized as labels of subsequent transitions on a path starting from state p. The path –a sequence of states– is stored in variable P during the traversal. In the first while loop in lines 3-6, the first part of the path is traversed. The loop ends when either a subsequent symbol in w cannot be matched against a label of an outgoing transition of a subsequent state in the path (i.e. the whole path has been traversed), or a target state of a transition is a confluence state. In both cases, the state reached is removed from the register because its suite of outgoing transitions will be modified. The index of that state is stored in variable j. When it is the latter case, the loop in lines 9-12 is executed. The loop is identical to the previous one, except that the states visited are cloned, and transitions from the previous states in the path are redirected towards the clones. Note that if a state is cloned, then its suite of outgoing transitions is copied to the clone, all targets of outgoing transitions have at least two incoming transitions (from the original and from the clone), and the next state on the path to be traversed becomes a confluence state. All clones are new states, and (contrary to originals) they are not present in the register.

Let us return to function UnsortedAdd shown as Algorithm 6. Function AddSuffix –the same that is used in sorted construction algorithm– creates a chain of states and transitions that recognize the part of string w that was not found and followed in the automaton by the function TraverseLCP. The chain is attached to state q that is the i-th state in the path P. Symbols from the string part $w_{1\dots i}$ are labels on transitions in the path P. In line 5 of UnsortedAdd, LocMin is called. Note that the chain of states to be visited by LocMin starts at the index j, not i. If no confluence states were encountered in TraverseLCP, then $j = i$. Otherwise, some states in the path –P_i to P_j– were cloned. They are not in the register and they should be minimized. After a call to LocMin, only the state P_j is not in the register. Recall that due to the order used for visiting the states, equivalence can be checked by comparing the finality and the suite of outgoing transitions of two states. When a state is replaced with an equivalent one, and the transition from the previous state in the path is redirected towards the replacement state, the suite of outgoing transitions of the previous state in the path changes, the state becomes an object of possible replacement, and it should be removed from the register.

That is done in function MinimizeBackwards shown as Algorithm 8. While a current state is equivalent to another state in the register, its is replaced, the transition from the previous state is redirected towards the replacement, and the previous state is removed from the register. The process ends when no replacement is possible.

Let us go through an example. The automaton A in Figure 4.25 recognizes words *paid*, *pay*, *paying*, *pays*, *play*, *played*, *playing*, *plays*, *pray*, *prayed*, *praying*, *prays*, *said*, *say*, *saying*, *says*, *stay*, *staying*, and *stays*. We assume that the function UnsortedConstruction has created A and the

Algorithm 8 As state P_i changed, check it for equivalence with states in the already minimized part. If the state is replaced, propagate changes backwards towards the initial state until no farther replacement takes place.

```
procedure MinimizeBackwards(A, R, P, i)
    $q \leftarrow P_i$
    while $i > 1 \wedge \exists_{r \in R} r \equiv q$ do
        $i \leftarrow i - 1$
        if $i > 1$ then $R \leftarrow R \setminus \{P_i\}$
        end if
        $\delta(P_i, w_i) \leftarrow r$
        delete $q$
        $q \leftarrow r$
    end while
    $R \leftarrow R \cup \{q\}$
end procedure
```

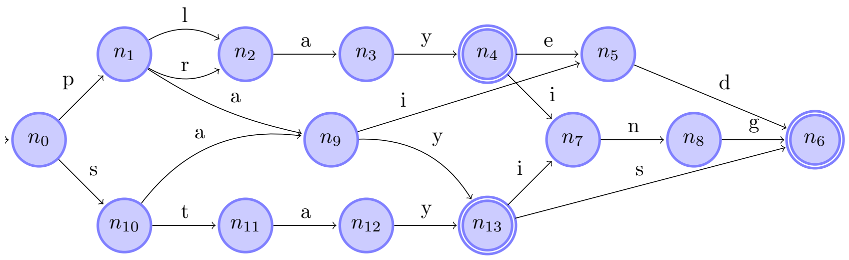


Fig. 4.25 A DFA recognizing words *paid*, *pay*, *paying*, *pays*, *play*, *played*, *playing*, *plays*, *pray*, *prayed*, *praying*, *prays*, *said*, *say*, *saying*, *says*, *stay*, *staying*, and *stays*.

register R in previous runs of the main loop. String w=*stayed* is read from the input, and the procedure UnsortedAdd is called with A, R (containing all states of A), and *stayed* as parameters. Procedure UnsortedAdd calls TraverseLCP. The first while loop of that function traverses states states n_0, n_{10}, n_{11}, and n_{12} storing them in variable P. Since state n_{13} has two incoming transitions, the loop cannot be executed again on that state. The state q being n_{12} is removed from the register, and its position 4 of the label y on the transition from n_{12} to n_{13} in the string w is stored in variable j. The second loop clones state n_{13} as n'_{13}, and puts it into P. The resulting automaton is shown in Figure 4.26. Function TraverseLCP returns $((n_0, n_{10}, n_{11}, n_{12}, n'_{13}), 5, 4)$.

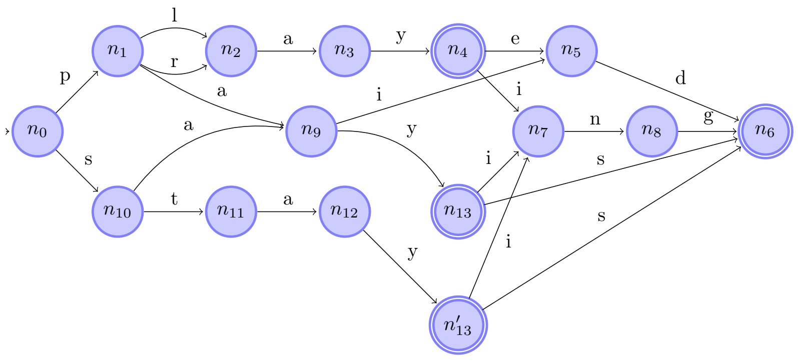


Fig. 4.26 A DFA from Figure 4.25 with state n_{13} cloned as n'_{13}.

Procedure UNSORTEDADD calls ADDSUFFIX with A, n'_{13}, and *ed* as parameters. The latter creates a non-final state n_{14}, and a final state n_{15}, as well as two transitions: one from n'_{13} to n_{14} labelled with e, and a second one from n_{14} to n_{15} labelled with d. The situation is depicted in Figure 4.27.

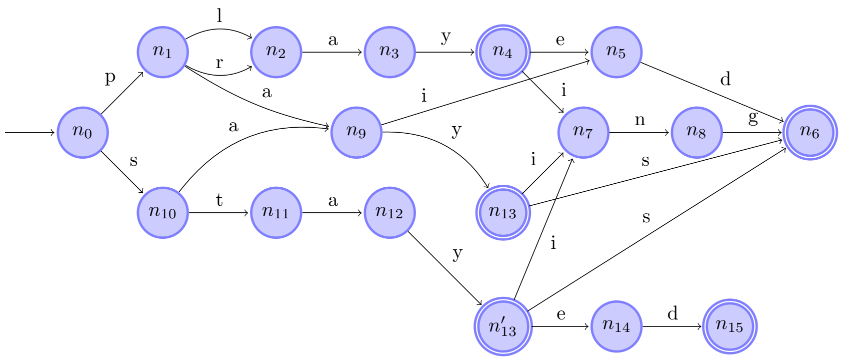


Fig. 4.27 A DFA from Figure 4.26 with suffix *ed* added.

The next step is the local minimization implemented in the procedure LOCMIN, called with A, with R containing all states of A except for n'_{13}, n_{14}, and n_{15}, with the state n'_{13}, and with a string d as parameters. The function calls itself in line 3 of Algorithm 4 until state n_{15} (and an empty suffix) is reached. State n_{15} is found equivalent to n_6, so n_{15} is deleted, the function returns n_6, and the transition from n_{14} to n_{15} labeled with d is redirected to state n_6 in line 3 of the call one level up in the call hierarchy. Similarly, n_{14} is replaced with n_5, and n'_{13} (now with a transition labelled

e going to n_5, and a transition labelled i going to n_7) is replaced with n_4. The function returns n_4. The situation is shown in Figure 4.28.

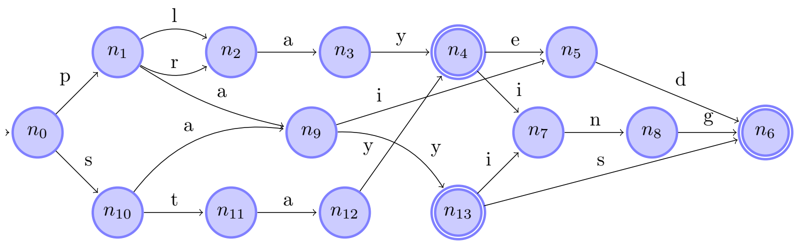


Fig. 4.28 A DFA from Figure 4.27 after LocMin was called.

Procedure MinimizeBackwards takes over from LocMin to handle those states that were present in the automaton when a new string was added, but which change their right language, and which therefore can be replaced with other states. The parameters of MinimizeBackwards are the automaton A, the register R containing all states except for n_{12}, the path $P = (n_0, n_{10}, n_{11}, n_{12})$, and $i = 4$. State $P_4 = n_{12}$ is handled first. An equivalent state n_3 is found in the register, so the previous state in the path $P_3 = n_{11}$ is removed from the register, a transition from n_{11} to n_{12} labelled with a is redirected to n_3, and state n_{12} is deleted. Similarly, $P_3 = n_{11}$ is replaced with n_2, $P_2 = n_{10}$ is removed from the register, a transition is redirected, and n_{11} is deleted. As there is no state equivalent to n_{10}, the loop is skipped, and n_{10} is put back into the register. A is now minimal, as shown in Figure 4.29.

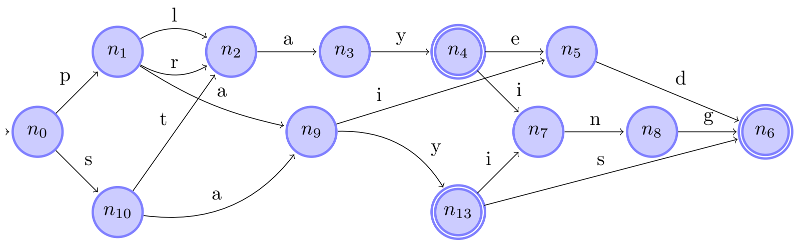


Fig. 4.29 A DFA from Figure 4.27 after MinimizeBackwards was completed.

4.3.2 *Construction from Strings with Some Cyclic Structures*

Some dictionaries may have a cyclic core recognizing words of infinite lengths while still recognizing finite strings. In such cases, the cyclic core may be built first, and then the finite strings are added. Construction of the cyclic core is covered later in this section. Here, we describe how to modify the incremental algorithms to work on cyclic automata. Algorithm 9 lists the skeleton of the modified sorted data algorithm.

Algorithm 9 Add lexicographically sorted finite strings to the language of a cyclic automaton A. Return a new automaton that recognizes the additional strings.

```
1: function AddSortedToCyclic(A)
2:     $R \leftarrow Q$; $r \leftarrow \text{clone}(q_0)$; $w' \leftarrow \epsilon$
3:     while input not empty do
4:         $w \leftarrow$ next string from input
5:         CyclicSortedAdd($A, R, r, w, w'$)
6:     end while
7:     LocMin($A, R, r, w$)
8:     if $r \neq q_0$ then
9:         DeleteBranch($A, R, q_o$); $q_0 \leftarrow r$
10:    end if
11:    return A
12: end function
```

In contrast to the acyclic version, in a cyclic automaton, confluence states can always be found when recognizing the initial part of the string. Confluence states can create a boundary between the unmodified, original part of the automaton, and the part created to recognize newly added strings. The new part is handled in a similar way to the original algorithm for acyclic automata. In the old part, confluence states are cloned to ensure that the conditions for the original algorithm, i.e. no confluence states when adding new states to a trie. The boundary is created by cloning the initial state in line 2. Since cloning involves copying the suite of outgoing transitions, every target of those transitions becomes a confluence state. Like in the original algorithm, local minimization is called in line 7 to minimize states in the path recognizing the last added string. The path contains no confluence states. Since we use the cloned start state as the root of our new trie, the original start state and some states reachable from it may become unreachable if the last call to LocMin did not unify the clone with the original start state. The unreachable states are removed with a call to DeleteBranch.

Algorithm 10 Add a single string to the language of a cyclic, deterministic automaton A. R is the register, r is the start state, w is the string to be added, w' is the previous string added. A and R are modified by the procedure.

```
1: procedure CyclicSortedAdd($A, R, r, w, w'$)
2:     $q \leftarrow r$; $i \leftarrow 1$
3:     while $i < |w| \wedge \delta(q, w_i) \in Q \wedge |\text{fanin}(\delta(q, w_i))| = 1$ do
4:         $q \leftarrow q$; $i \leftarrow i + 1$
5:     end while
6:     while $i < |w| \wedge \delta(q, w_i) \in Q$ do
7:         $p \leftarrow \text{clone}(\delta(q, w_i))$; $\delta(q, w_i) \leftarrow p$
8:         $q \leftarrow p$; $i \leftarrow i + 1$
9:     end while
10:    if $|w'| > i$ then
11:        LocMin($A, R, q, w'_{i\ldots|w'|}$)
12:    end if
13:    AddSuffix($A, q, w_{i\ldots|w|}$)
14: end procedure
```

Procedure CyclicSortedAdd is depicted as Algorithm 10. The first while loop traverses the part of the longest common prefix that has already been cleared of confluence states. The second one clones confluence states. That second loop is the only difference with respect to the corresponding part of the original algorithm.

Procedure DeleteBranch is shown as Algorithm 11. It is called in line 9 of Algorithm 9. Its purpose is to delete unreachable states starting from state q. It is assumed that the state q is unreachable. Under this assumption, all states that are not reachable from any state not reachable from q are also unreachable. If a transition going out from state q reaches a state p with only one incoming transition, then p is reachable only from q, and DeleteBranch is called to act on p to delete the state and other states reachable only from p. After that, the transition is deleted. If the target state p has more than one incoming transition, then only the

Algorithm 11 Delete unreachable states. A is the automaton, R is the register, q is the first unreachable state to be deleted. A and R are modified by the algorithm.

```
1: procedure DeleteBranch(A, R, q)
2:     for all $\sigma \in \Sigma : \delta(q, \sigma) \in Q$ do
3:         if $|\mathrm{fanin}(\delta(q, \sigma))| = 1$ then DeleteBranch($A, R, \delta(q, \sigma)$)
4:         end if
5:         delete transition $\delta(q, \sigma)$
6:     end for
7:     $R \leftarrow R \setminus \{q\}$
8:     delete $q$
9: end procedure
```

transition from q is deleted. Note that if state p has more than one incoming transition, but all of them coming from q, then the transitions will be deleted one by one until there is only one left, and then DeleteBranch is called on p. After all outgoing transitions of q are dealt with, the state is removed from the register and deleted. Note that all the states that are arguments to DeleteBranch are present in the register since they are all "old" states.

The main loop in Algorithm 9 is executed n times, where n is the number of strings to be added to the language of the automaton A. The call to LocMin is executed in $\mathcal{O}(|w_{max}|)$ time, where w_{max} is the longest string to be added, provided that operations on the register take constant time. A call to DeleteBranch takes $\mathcal{O}(n|w_{max}|)$ time under the same assumption as every state to be deleted is a clone of a state visited following a path recognizing any of the n strings added. Function CyclicSortedAdd is called n times. In each call, the two while loops traverse the longest common prefix, and they are executed in $\mathcal{O}(|w_{max}|)$ time. Also calls to LocMin and to AddSuffix run in $\mathcal{O}(|w_{max}|)$ time. Thus the whole algorithm runs in $\mathcal{O}(n|w_{max}|)$ time provided that register operations are executed in constant time.

The modifications to the algorithm for unsorted data shown as Algorithm 5 on page 176 are minute. They are given as Algorithm 12.

We clone the start state if it has any incoming transitions. As this may create unreachable states, a call to DeleteBranch deletes them. Since we do not start from scratch, we do not need to create it, but we need to initialize the register with all the states Q of the automaton.

Algorithm 12 A skeleton of the algorithm for adding a set of finite strings to a cyclic automaton. A is the automaton. Function AddUnsortedToCyclic returns a modified automaton.

```
function AddUnsortedToCyclic($A$)
    if $|\text{fanin}(q_0)| > 0$ then
        $r \leftarrow q_0$; $q_0 \leftarrow \text{clone}(q_0)$
        $R \leftarrow Q$
    end if
    while input not empty do
        $w \leftarrow$ next string from input
        UnsortedAdd($A, R, w$)
    end while
    if $r \neq q_0$ then
        DeleteBranch($A, R, r$)
    end if
end function
```

4.3.3 *Construction from Smaller Automata*

Dictionaries can be very large. The incremental algorithms can construct acyclic automata very efficiently, but some dictionaries can be cyclic, and some may already come as automata. Even for dictionaries containing only finite length strings, automata were constructed by building automata from smaller parts before the advent of incremental algorithms.

Given two automata $A_1 = (Q_1, \Sigma, \delta_1, q_{01}, F_1)$ and $A_2 = (Q_2, \Sigma, \delta_2, q_{02}, F_2)$, we can obtain a minimal DTA A such that $L(A) = L(A_1) \cup L(A_2)$ in at least two ways:

(1) Build an NFA $A' = (Q_1 \cup Q_2 \cup \{q_0\}, \Sigma, \delta, q_0, F_1 \cup F2)$ with δ such that $\forall_{p,q \in Q_1, \sigma \in \Sigma} \delta_1(q, \sigma) = p \Rightarrow \delta(q, \sigma) = \{p\}$, $\forall_{q,p \in Q_2, \sigma \in \Sigma} \delta_2(q, \sigma) = p \Rightarrow \delta(q, \sigma) = \{p\}$, and $\delta(q_0, \varepsilon) = \{q_{01}, q_{02}\}$. Then determinize and minimize A'.

(2) Build a DFA $A' = (Q_1 \times Q_2, \Sigma, \delta, (q_{01}, q_{02}), F_1 \times F_2)$, with δ such that $\forall_{q_1, p_1 \in Q_1} \forall_{q_2, p_2 \in Q_2} \forall_{\sigma \in \Sigma} \delta_1(q, \sigma) = p_1 \wedge \delta(q_2, \sigma) = p_2 \Rightarrow \delta((q_1, q_2), \sigma) = (p_1, p_2)$. Minimize A'. Note that using the formula directly would lead to many unreachable states. It is better to start from q_0, and build subsequent reachable states by calculating δ.

4.3.4 *Construction from Other Sources*

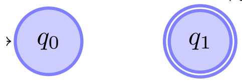
Data for dictionaries may come from various sources, so the construction methods must take that into account; they can also profit from the situation.

Automata can be constructed from regular expressions. A regular expression RE is:

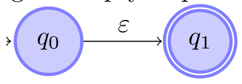
- An empty set $\emptyset \in \text{RE}$
- An empty sequence of symbols $\varepsilon \in \text{RE}$
- A symbols from the alphabet $\Sigma \ni \sigma \in \text{RE}$
- A concatenation $rs \in \text{RE}$ of two regular expressions $r, s \in \text{RE}$
- An alternative $r|s \in \text{RE}$ of two regular expressions $r, s \in \text{RE}$
- A transitive closure $r^* \in \text{RE}$ of a regular expression $r \in \text{RE}$

Two of the most known construction methods are Thompson construction [57] and Glushkov/Yamada-McNaughton construction [24], [42]. In Thompson construction, each regular expression is associated with a nondeterministic automaton that has one start state and one final state that is different from the start state. The automata recognizing the basic building blocks or regular expressions are built as follows:

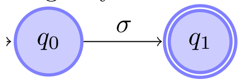
- An automaton $A = (\{q_0, q_1\}, \Sigma, \emptyset, q_0, \{q_1\})$recognizing an empty set:



- An automaton $A = (\{q_0, q_1\}, \Sigma, \delta, q_0, \{q_1\})$, with $\delta(q_0, \varepsilon) = q_1$ recognizing an empty sequence of symbols ε:



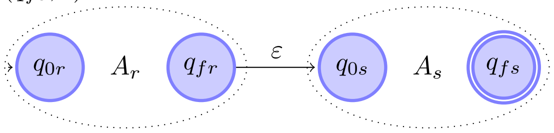
- An automaton $A = (\{q_0, q_1\}, \Sigma, \delta, q_0, \{q_1\})$, with $\delta(q_0, \sigma) = q_1$ recognizing a symbol $\sigma \in \Sigma$:



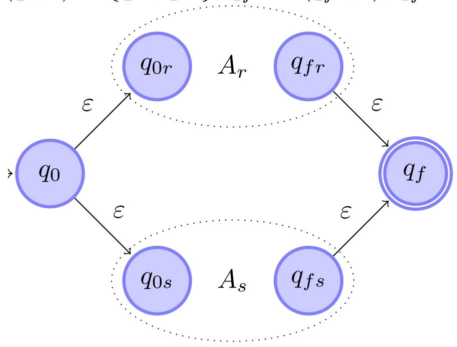
Given regular expressions r and s, and automata $A_r = (Q_r, \Sigma, \delta_1, q_{0r}, \{q_{fr}\})$ and $A_s = (Q_s, \Sigma, \delta_s, q_{0s}, \{q_{fs}\})$ recognizing them, we can construct automata recognizing more complex structures:

- Concatenation rs: $A = (Q_r \cup Q_s, \Sigma, \delta, q_{0r}, q_{fs})$, with all transitions of δ_r and δ_s also present in δ: $\forall_{q \in Q_r} \forall_{a \in \Sigma \cup \{\varepsilon\}} \delta_r(q, a) \subseteq \delta(q, a)$,

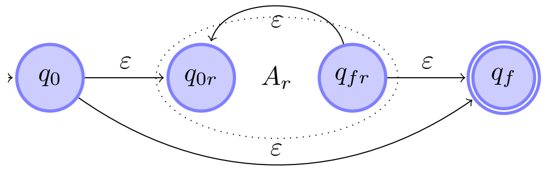
$\forall_{q \in Q_s} \forall_{a \in \Sigma \cup \{\varepsilon\}} \delta_s(q,a) \subseteq \delta(q,a)$, and one additional transition $q_{0s} \in \delta(q_{fs}, \varepsilon)$



- Alternative $r|s$: $A = (Q_r \cup Q_s \cup \{q_0, q_f\}, \Sigma, \delta, q_0, \{q_f\})$, with all transitions of δ_r and δ_s also present in δ: $\forall_{q \in Q_r} \forall_{a \in \Sigma \cup \{\varepsilon\}} \delta_r(q,a) \subseteq \delta(q,a)$, $\forall_{q \in Q_s} \forall_{a \in \Sigma \cup \{\varepsilon\}} \delta_s(q,a) \subseteq \delta(q,a)$, and with additional transitions $\delta(q_0, \varepsilon) = \{q_{0r}, q_{0s}\}$, $q_f \in \delta(q_{fr}, \varepsilon)$, $q_f \in \delta(q_{fs}, \varepsilon)$



- Transitive closure r^*: $A = (Q_r \cup \{q_0, q_f\}, \Sigma, \delta, q_0, \{q_f\})$, with all transitions of δ_r also present in δ: $\forall_{q \in Q_r} \forall_{a \in \Sigma \cup \{\varepsilon\}} \delta_r(q,a) \subseteq \delta(q,a)$, and additional transitions $\delta(q_0, \varepsilon) = \{q_{0r}, q_f\}$, $\{q_{0r}, q_f\} \in \delta(q_{fr}, \varepsilon)$



4.4 Internal Structure and Compression

4.4.1 *Representation of Automata*

Since finite-state automata can be seen as a labeled directed graphs, one can represent them by using standard data structures used to represent graphs, i.e. *adjacency matrices* and *adjacency lists*, with some slight enhancements. In principle, selecting an appropriate storage model for an automaton requires consideration of three major operations which will be carried out on that automaton: (a) acessing $\delta(q, a)$, (b) iterating over transitions from a given state q, and (c) modifications, e.g., insertion and deletion of transitions. In other words, it is essential to know whether the automaton is static (it will be deployed in read-only mode) or dynamic (modification operations will be carried out). Unfortunately, there is no data structure that would be optimal in terms of time and space complexity for all kinds of operations. Consequently, different data structures are deployed at different processing stages. The remainder of this section gives an overview of a three most prevalent data structures used for representing automata. For simplicity reasons, we focus on DFAs, but most of the data structures can be extended to NFAs and FSTs straightforwardly.

4.4.1.1 *Transition Matrix*

The simplest way to represent a DFA $A = (\Sigma, Q, \delta, q_0, F)$ is to use a $|Q| \times |\Sigma|$ matrix whose ij-th element contains the value of $\delta(q_i, a_j)$, where a_j is the j-th symbol in the alphabet Σ. If $\delta(q_i, a_j) = \perp$, then the corresponding matrix element contains a **null** value. The presented data structure is known as the *adjacency matrix*, which is called in the context of automata a *transition matrix*. An extension to NFAs is straightforward, i.e., the ij-th element of the transition matrix contains a list of all target states of outgoing transitions from state q_i labeled with a_j. Figure 4.30 gives an example of a simple DFA with the corresponding transition matrix depicted in Table 4.1. For marking states as initial, accepting or rejecting, one can deploy a simple boolean-valued vector.

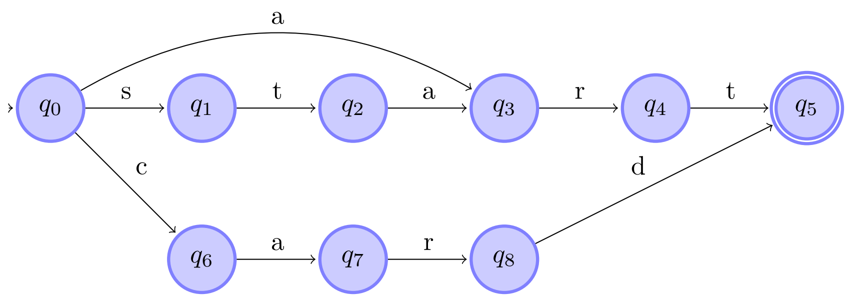


Fig. 4.30 An DFA accepting the language $\{start, card, art\}$.

The major advantage of a transition matrix representation is that accessing $\delta(q, a)$ information and performing transition insertion/deletion

Q/Σ	a	c	d	r	s	t
q_0	q_3	q_6			q_1	
q_1						q_2
q_2	q_3					
q_3				q_4		
q_4						q_5
q_5						
q_6	q_7					
q_7				q_8		
q_8			q_5			

costs $O(1)$ time. The major disadvantage is the $\Theta(|\Sigma|)$ time complexity of iterating over all transitions from a given state. A further drawback is the fact that transition matrix requires $\Theta(|Q| \cdot |\Sigma|)$ space. In the example in Table 4.1 only a minor part of the transition matrix is filled with non-**null** values, i.e. the average number of outgoing transitions from a given state is relatively low. We call such automata *sparse*, whereas automata with a high average number of outgoing transitions from a given state are called *dense*. In most NLP applications, however, one deals with very sparse automata. Therefore, the matrix representation is used relatively rarely, most typically for dense and small automata, especially when there is a need to frequently modify transitions in constant time.

4.4.1.2 *Transition Lists*

An alternative way of representing an automaton is to use *adjacency lists*, which are called in the world of finite-state devices *transition lists*. For each state q in the automaton one defines a list of all pairs (a, p) such that $\delta(q, a) = p$. This data structure is suitable for both DFAs and NFAs. Figure 4.31 shows an example of transition-list representation of the automaton in Figure 4.30.

The main advantage of using transition list model is its low space complexity, proportional to the size of the automaton, which amounts to $\Theta(|Q| + |\delta|)$. Furthermore, iterating over all transitions from a given state q costs $\Theta(|fanout(q)|)$, which is an improvement compared to transition matrix. Having the two nice aforementioned features is penalized by slower access to $\delta(q, a)$. In the worst case the whole list for a given state has to be traversed, which results in $O(|\Sigma|)$ complexity in the case of DFAs and

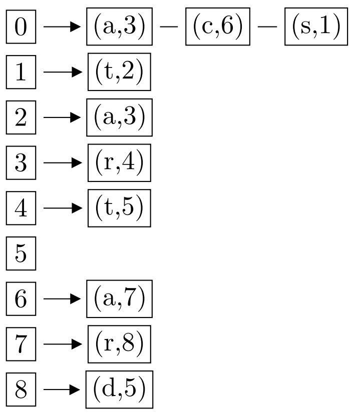


Fig. 4.31 Transition-list representation of the automaton in Figure 4.30.

$O(|Q| \cdot |\Sigma|)$ for NFAs. As mentioned earlier, in the context of NLP dictionaries one deals with sparse automata, which means that the runtime performance of the $\delta(q, a)$-look-up operation does not constitute a critical factor. Nevertheless, in the case of denser automata, one could sort the transition lists by input symbol and store them using balanced trees or other more complex data structure for implementing dynamic sorted sets [9]. Consequently, accessing $\delta(q, a)$ information and performing modification operations could be done in $O(log|fanout(q)|)$ time. We will denote this variant representation as *sorted transition lists.*

In comparison to transition matrix there is still one more advantage, namely: introducing new transitions labeled with symbols not covered by the current alphabet is straightforward, whereas in the case of a transition matrix expensive reorganization of the matrix is indispensible.

4.4.1.3 *Compressed Transition Matrix*

Sparse transition matrices can be compressed in such a manner that the space requirement is nearly linear in the number of transitions, without sacrifying the constant time for accessing $\delta(q, a)$. The idea is to shift and overlap the rows of the transition matrix so that no two non-zero entries end up in the same position, and to store them in a one-dimensional array. This can be done in a greedy manner by successively placing the consecutive rows of the transition matrix into the array DELTA in such a way that collisions are avoided, i.e. a single element in the array DELTA may refer to at most one element in some row of the transition matrix. Additionally, one introduces an array ROW[1...$|Q|$] for storing for each state a pointer to the beginning of the corresponding transition row stored in DELTA. Figure 4.32 gives an example of shifting and overlapping the rows of the transition matrix presented in Table 4.1. The rows 0,1,2,3 and 5 can share the same space in DELTA without performing any shifting operations, whereas other rows have to be displaced in order to avoid clashes.

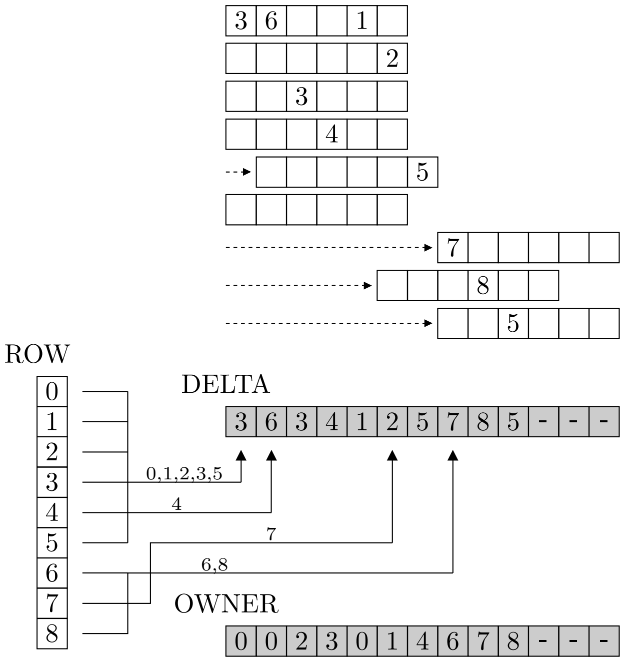


Fig. 4.32 Compression of transition table presented in Figure 4.1.

For accessing $\delta(q, a)$, the element in DELTA at position ROW[q]+INDEX[a] has to be checked, where INDEX maps each alphabet symbol to a unique integer. In order to guarantee that a non-empty value in DELTA at a given index encodes the target state of some outgoing transition from state q another one-dimensional array OWNER of the same length as DELTA is used, which assigns each element in DELTA an associated state, i.e. OWNER[i]=q means that the information stored in DELTA[i] refers to a transition for state q. If $\delta(q, a) = \perp$ and the corresponding element in DELTA is not utilized, the latter is assigned a separate value which denotes “undefined and unused” (dash in our example). The pseudocode for accessing δ is given in Algorithm 13.

Algorithm 13 The pseudocode of the algorithm for accessing $\delta(q, a)$.

```
1: function GetDelta(q,a)
2:     if $OWNER[ROW[q] + INDEX[a]] = q$ then
3:         return $DELTA[ROW[q] + INDEX[a]]$
4:     end if
5:     return NIL
6: end function
```

The task of finding an optimal set of matrix row displacements, i.e., one which results in minimal size of DELTA, is NP-complete. Nevertheless, there are many heuristics which yield nearly optimal compression rates. The simplest way is the so called 'first-fit' strategy. In the i-th iteration one tries to shift the i-th row from left to right over the previously packed first $i - 1$ rows already stored in DELTA, until a zero-collision overlap has been identified. Since the initial elements in DELTA will already be covered by the first couple of transition rows, a more efficient way is to start computing a collision-free overlap from the first non-occupied position in DELTA. Further improvements can be achieved by sorting all transition rows with respect to the number of transitions they encode, which imposes the order of packing the rows. The application of the above techniques may result in a fair compression rate. But for all that, compressed matrices are not suitable for representing dynamic automata since adding new transitions would lead to time-intensive recomputations.

4.4.1.4 *Comparison*

Table 4.2 summarizes the main features of the presented storage models for DFAs in terms of space and time complexity of relevant operations ($FO(q)$ denotes $fanout(q)$).

For NFAs all the values in the table are identical except the space complexity of the transition matrix, which amounts to $\Theta(|Q| \cdot |\Sigma| + |\delta|)$ since the elements of the transition table are lists of total length $|\delta|$. Analogously,

Table 4.2 Comparison of different data structures for representing DFAs.

Data Structure	space	accessing $\delta(q, a)$	iteration	modification
transition matrix	$\Theta(\lvert Q\rvert \cdot \lvert\Sigma\rvert)$	$O(1)$	$\Theta(\lvert\Sigma\rvert)$	$O(1)$
transition lists	$\Theta(\lvert Q\rvert + \lvert\delta\rvert)$	$O(FO(q))$	$\Theta(FO(q))$	$O(FO(q))$
sorted transition lists	$O(\lvert Q\rvert + \lvert\delta\rvert)$	$O(log(FO(q)))$	$\Theta(FO(q))$	$O(log(FO(q)))$
compressed matrix	$\Omega(\lvert Q\rvert + \lvert\delta\rvert)$ $O(\lvert Q\rvert \cdot \lvert\Sigma\rvert)$	$O(1)$	$\Theta(\lvert\Sigma\rvert)$	n.a.

the space complexity of a compressed transition matrix is $O(|Q| \cdot |\Sigma| + |\delta|)$ $(\Omega(|Q| + |\delta|))$.

4.4.2 *Compression Techniques*

There are several ways of further reducing the space complexity and speeding-up crucial operations. We briefly sketch some ideas which are of somewhat technical nature:

Mixed representation: Sometimes only a small part of the frequently-visited states have a large number of outgoing transitions. If that is a case, one could introduce transition arrays of length $|\Sigma|$ for representing outgoing transitions from such states, whereas for all other states conventional adjacency lists could be applied.

Macros: For a subset of symbols in the alphabet reoccurring in some contexts, e.g., symbols appearing as labels on transitions between the same pair of source and target state, one could introduce macros or wildcars, which in some scenarios might lead to tremendous space savings.

Path compression: All sequential paths, i.e. paths whose intermediate states have single ingoing and single outgoing transition could be compressed by replacing them by a single transition whose label is obtained by jamming all labels of the transitions in the path [4, 48].

Using every bit: Some parts of the automaton do not have to be stored explicitly, e.g., instead of storing states we can implement transitions in such a way to include pointers to memory blocks containing transitions of target states [12].

Sharing space: Some parts of the automaton can be shared, e.g., the set of transitions from one state could be a subset of a transition set of another state. Consequently, through some rearrangement of these two sets of transitions. we could use the same memory block for encoding shared transitions [12],

Hardcoding: For a certain range of automata a significant performance improvement in terms of processing efficiency can be achieved by hardcoding transition function [45].

The detailed description of each of these methods is beyond the scope of this book. Up to a degree, the decision what is the best space/speed trade-off in compression is a subjective one, and the choice of methods is usually based on the preferences of software designer.

We shall present here one method that is the best as far as the compression goes. It is based on space optimized implementation of a recursive automaton that is used straightforwardly for storing lexicons. Compressed RA, with some additional tricks, is the foundation part for a compact representation of a proper dictionary.

4.4.2.1 *A Space Optimized Representation of RA*

As of recently, a number of authors have observed the potential of subautomata substitution for compression [18, 54, 58]. Each have implemented different automata representations with different compression results. We shall describe here the one implementation that leads to the best compression. The implementation uses lists of transitions, some of which are final. Storing finality in transitions instead of states is equivalent to using Mealy automaton instead of more common Moore's. This is a non-standard representation in the field of finite automata, but leads to a smaller number of states and transitions [10]. States in an automaton are represented implicitly with lists of transitions leading from the state. The transitions are composed of four parts:

- Label;
- Pointer to the next transition in the same state (NT$\rightarrow$);
- Flag indicating whether the transition is final;
- Flag indicating whether there is a continuation of some path in the automaton past the current transition.

An example of a recursive automaton implemented in this way is presented in Figures 4.33 and 4.34. Figure 4.33 shows a set of sixteen transitions that are the actual stored data, and in Figure 4.34 is the layout of underlying Mealy machine. This is a real world implementation of the conceptual recursive automaton presented in Figure 4.4.

Out of sixteen transitions fourteen are regular transitions, and entries 10 and 15 are calls to a subautomaton. In regular transition a label is a letter of a word. Transitions are stored in such order that the consecutive transitions do not belong to the same state but to the same path. The transitions from 1 through 3 form the word *low* and from 1 through 5 *lower.*

Transition	Label	NT→	Final	Next
1	l	10 (rel)		●
2	o	4 (rel)		●
3	w	-	●	●
4	e	-		●
5	r	-	●	
6	i	-		●
7	g	-		●
8	h	-		●
9	t	-	●	●
10 (RC)	∞	4 (abs)		
11	s	-		●
12	o	3 (rel)		●
13	w	-	●	●
14	s	-	●	
15 (RC)	4	6 (abs)		
16	s	-	●	

Fig. 4.33 Space efficient implementation of a recursive automaton.

The *Final* flag at transition 3 indicates that the word *low* is accepted, and the *Next* flag denotes that there is a continuation of a path to the next entry. Transition 5 is the last one on this particular path so its *Next* flag is *off*.

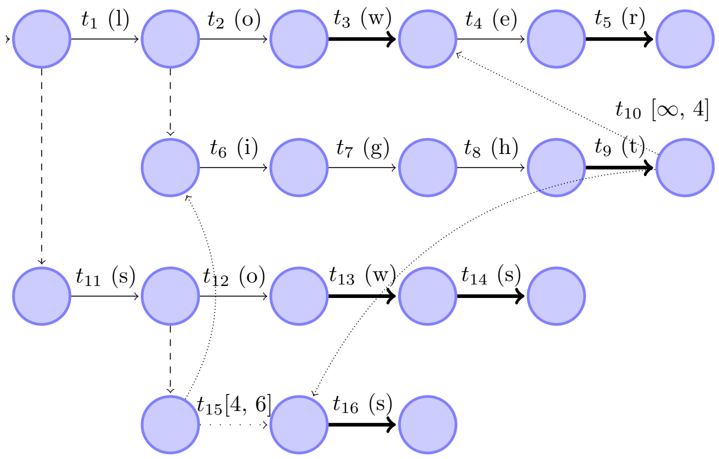


Fig. 4.34 Mealy machine implemented with transitions shown in Figure 4.33. Accepting transitions are drawn with thick arrows. The machine is equivalent to the automaton in Figure 4.4.

Transitions belonging to the same state are linked with NT$\rightarrow$ pointers. States q_0 and q_1 in Figure 4.4 are composed of two transitions each. The transitions 1 and 11 form the state q_0 and transitions 2 and 6 the state q_1. Transitions belonging to a same state are connected with dashed lines in Figure 4.34. The values of NT$\rightarrow$ pointers are stored as a relative distance in number of transitions. For example, the pointer in transition 2 has a value 4 and therefore points to transition 6. Relative addressing is convenient because in this way more subautomata remain identical, but for this reason all transitions have to be of the same size. This is also the case with entries 10 and 15 that are subautomaton calls. They are of the same size as regular transitions, but with a different assignment of values. Entry 10 is a call to subautomaton consisting of transitions 4 and 5, and entry 15 is a call to subautomaton consisting of 4 transitions starting at position 6.

Besides target address, a call must contain the length of replaced part. An infinity sign ∞ (as in entry 10) denotes that we don't have to return after executing the lookup procedure at the target. This means that the target state and the state that is the source of a call have the same right language, which amounts to merging of the equivalent states in MDFA. Some reserved code is used for the infinity notion, and a combination when both *Final* and *Next* flags are *off* indicates that the entry is a call and not a regular transition.

The target locations are stored as absolute addresses. This way every call to the same subautomaton has the same values at each position in the structure and can be included in replaced subautomata. The whole idea of the described coding is to have as much as possible uniformity throughout the structure. One of the necessary prerequisites for this is that words are added to the structure in some predefined order, usually alphabetical.

Calls can replace only those sequences of transitions that have exactly the same value in all fields. That is why the transitions 12 and 13 can not be substituted with 2 and 3. This underlines the difference between the conceptual representation of RA from Figure 4.4 and the actual one. In real life the equivalence of automaton parts depends on the way the automaton is constructed. In our implementation all transitions and calls are of equal size, so the space is saved when a call replaces at least two transitions. That is why we would gain nothing by replacing, for example, transition 16 with a call to transition 14. The coding depicted in Figure 4.33 leads to the most compact representation of automaton published so far [6] and, with some additional standard compression tricks, gives the best results in lexicon compression [53].

The algorithm for construction of compressed automaton needs to find all subautomata that are equal. That means any sequence of two or more transitions that is repeated anywhere else in the initial automaton. If we regard the automaton as a string of transitions then this becomes a well known substring repetition problem. It can be solved in time linear with the input string size by using suffix tree or array [27]. However, there is one computationally hard problem linked to this method. Different ordering of transitions may lead to more or less repeated transition sequences. We can get different ordering of transitions with alternative ordering of input words. For instance, the words in our example automaton are not in alphabetical order. If they were, different sets of transitions would be replaced with calls. In this case the total number of entries would still be sixteen (which we invite interested reader to verify), but there are cases where the final size would vary with the order of input words. Apparently, there doesn't exist an efficient algorithm for finding the optimal order of transitions [54]. Fortunately, the difference in final sizes can never be greater than few percent and storing the input words in alphabetical order is accepted as a standard solution.

4.4.2.2 *Dictionary Compression*

An efficient method for compressing a proper dictionary would be to construct strings out of keys and corresponding attributes and then store the strings in described compressed structure. However, better compression can be achieved if the input and output sets are separated and stored in respective hashing automata. Hashing is necessary so that we can keep the links between two sides. Hashing automata for keys and attributes associate unique numbers with each entry. When attributes are separated from keys they have to be alphabetically sorted in order to produce a compact automaton. This destroys the connections between keys' and attribute's numbers. Therefore, an additional index must be constructed. This index can be an array where an entry for each key is the hashing number of its associated attribute. Fortunately, it is usually possible to store such an array in a compressed format. The dictionary implementation then has three components: keys hashing automaton, attributes hashing automaton and index that links them. It has been shown, for a case of a natural language dictionary, and when index is compressed with a combination of delta and statistical coding, that cumulative sum of the sizes of components is significantly smaller than the size of a single automaton [53].

Although the access speed is slower than in less complicated implementations, it is still very fast and the slow-down is not significant in absolute terms. As a rule of thumb, on present-day personal computers the lookup speed of described dictionary is over 100 000 entries per second.

4.4.2.3 *Compact Representation of Dynamic Dictionaries*

The compression is by far more successful with static then with dynamic dictionaries. This does not mean that we can't employ efficient methods for applications with dynamic dictionaries, too. The minimal automaton is a compressed structure in itself, so an efficient incremental and linear algorithm for the construction of MDFA is a compact way to implement dynamic dictionary. The earlier dictionary implementations used a number of different methods for dynamic trie compression. Some of them may still be of interest for their simplicity. However, if the space is critical, it is usually a good strategy to combine precompiled and compact static core dictionary and add what is needed online in some auxiliary structure. Then, at a convenient time, or perhaps on an alternating machine, recompile the new core.

4.4.3 *Conclusions and Further Reading*

We have demonstrated how to construct dictionaries of finite-state machines, and then how to use them. Our focus was mainly on acyclic automata, and we payed particular attention to compact representation of dictionaries. In application of dictionaries, we presented methods for morphological analysis and synthesis, spelling correction, and restoration of diacritics. Naturally, the chapter is only an overview, and more can be learned by reading other sources.

Deterministic finite-state automata, tries and DAWGs have been extensively studied and are well covered in a number of textbooks. See, for example, [25, 55].

Morphological analysis and synthesis can be done both with finite-state automata, and with transducers. For a description of such analysis or synthesis see e.g. [3]. Description of a coding of canonical form of lexemes when using FSAs can be found e.g. in [37] and [40]. Guessing automata were proposed by Jan Daciuk in [11]. More advanced treatment of unknown words can be obtained using concatenation and spelling rules like those described in [3].

An overview of spelling correction is given in [38]. A thorough description of isolated-word spelling correction with finite-state automata can be found in [46]. A technique that uses two dictionaries (one is inverted) for fast spelling correction is described in [43]. Other efficient implementations can be found in [60] and [59].

The algorithm for sorted data was developed independently by Jan Daciuk [19], Stoyan Mihov [17], and Ciura and Deorowicz [8]. The algorithm for unsorted data was developed independently by Aoe, Morimoto and Hase [1], Jan Daciuk, Bruce Watson and Richard Watson [19], and by Dominique Revuz [51] as well as Kyriakos Sgarbas, Nikos Fakotakis, and Georgios Kokkinakis [56]. A semi-incremental algorithm has been developed by Bruce Watson [62]. It requires data sorted on decreasing string length; such sorting can be done in linear time. Incremental deletion of strings is also possible, see e.g. [1].

The extension of the unsorted data algorithm to the case with an initial cyclic automaton has been proposed by Rafael Carrasco and Mikel Forcada [7]. The extension of the sorted data algorithm, as well as Watson's semi-incremental algorithm was proposed by Jan Daciuk [13], [14].

Automata for morphological analysis and synthesis can also be built directly from morphology systems using concatenation and spelling rules as proposed by Lauri Karttunen [31]. Both concatenation rules and spelling rules (phonological rules) can be handled by finite-state devices. For an overview of finite-state morphology see [32]. Two-level phonological rules are described in [34, 35], more rules (including sequential ones) are tackled in [30]. An excellent textbook covering all issues in natural language processing, including finite-state dictionaries, is [29].

The first implementation of a recursive automaton was presented in [52], the name, however, comes from [23]. Recursive automaton can be constructed by finding the repeated subautomata in MDFA in linear time [18, 58], but the best compression is achieved when the starting structure is a space intensive trie and with a slower algorithm [6, 53]. It is an open issue whether it is possible to build a space efficient recursive automaton in a linear time and using incremental procedure to avoid building the intermediate trie.

References

1. Aoe, J., Morimoto, K. and Hase, M. (1992). An algorithm for compressing common suffixes used in trie structures, *Trans. IEICE* .
2. Appel, A. W. and Jacobson, G. J. (1988). The world's fastest scrabble program, *Commun. ACM* **31**, 5, pp. 572–578, doi:http://doi.acm.org/10.1145/42411.42420.
3. Beesley, K. R. and Karttunen, L. (2002). *Finite-State Morphology: Xerox Tools and Techniques* (CSLI Publications).
4. Beijer, N., Watson, B. and Kourie, D. (2003). Stretching and Jamming of Automata, in *Proceedings of SAICSIT 2003* (Republic of South Africa), pp. 198–207.
5. Blumer, A., Blumer, J., Ehrenfeucht, A., Haussler, D. and McConnell, R. M. (1983). Linear size finite automata for the set of all subwords of a word - an outline of results, *Bulletin of the EATCS* **21**, pp. 12–20.
6. Budiscak, I., Piskorski, J. and Ristov, S. (2009). Compressing Gazetteers Revisited, Handout CD-ROM containing the accompanying papers for the presentations during the FSMNLP 2009 workshop. Published by the University of Pretoria, Pretoria, South Africa. ISBN 978-1-86854-743-2.
7. Carrasco, R. C. and Forcada, M. L. (2002). Incremental construction and maintenance of minimal finite-state automata, *Computational Linguistics* **28**, 2.
8. Ciura, M. and Deorowicz, S. (2001). How to squeeze a lexicon, *Software – Practice and Experience* **31**, 11, pp. 1077–1090.
9. Cormen, Leiserson and Rivest (1990). *Introduction to Algorithms* (MIT Press, Cambridge Mass.).
10. Daciuk, J. (1998). *Incremental Construction of Finite-State Automata and Transducers, and their Use in the Natural Language Processing*, Ph.D. thesis, Technical University of Gdańsk.
11. Daciuk, J. (1999). Treatment of unknown words, in *proceedings of Workshop on Implementing Automata WIA'99* (Potsdam, Germany), pp. IX–1 – IX–9.
12. Daciuk, J. (2000). Experiments with automata compression, in M. Daley, M. G. Eramian and S. Yu (eds.), *Conference on Implementation and Application of Automata CIAA'2000* (University of Western Ontario, London, Ontario, Canada), pp. 113–119.
13. Daciuk, J. (2004a). Comments on incremental construction and maintenance of minimal finite-state automata by Rafael C. Carrasco and Mikel Forcada, *Computational Linguistics* **30**, 2, pp. 227–235.
14. Daciuk, J. (2004b). Semi-incremental addition of strings to a cyclic finite automaton, in K. T. Mieczysław A. Kłopotek, Sławomir T. Wierzchoń (ed.), *Proceedings of the International IIS: IIP WM'04 Conference*, Advances in Soft Computing (Springer, Zakopane, Poland), pp. 201–207.
15. Daciuk, J., Maurel, D. and Savary, A. (2005). Dynamic perfect hashing with pseudo-minimal automata, in M. A. Kłopotek, S. Wierzchoń and K. Trojanowski (eds.), *Intelligent Information Processing and Web Mining, Proceed-*

ings of the International IIS: IIPWM'05 Conference held in Gdańsk, Poland, June 13-16, 2005, *Advances in Soft Computing*, Vol. 31 (Springer), pp. 169–178.

16. Daciuk, J., Maurel, D. and Savary, A. (2006). Incremental and semi-incremental construction of pseudo-minimal automata, in J. Farre, I. Litovsky and S. Schmitz (eds.), *Implementation and Application of Automata: 10th International Conference, CIAA 2005*, *LNCS*, Vol. 3845 (Springer), pp. 341–342.
17. Daciuk, J., Mihov, S., Watson, B. and Watson, R. (2000). Incremental construction of minimal acyclic finite state automata, *Computational Linguistics* **26**, 1, pp. 3–16.
18. Daciuk, J. and Piskorski, J. (2006). Gazetteer Compression Technique Based on Substructure Recognition, in *Intelligent Information Processing and Web Mining, Proceedings of the International Conference on Intelligent Information Systems, Ustroń, Poland, June 19-22, 2006* (Book Series Advances in Soft Computing, Vol. 35/2006, Springer, Berlin-Heidelberg), pp. 87–95.
19. Daciuk, J., Watson, R. E. and Watson, B. W. (1998). Incremental construction of acyclic finite-state automata and transducers, in K. Oflazer and L. Karttunen (eds.), *Finite State Methods in Natural Language Processing* (Bilkent University, Ankara, Turkey), pp. 48–56.
20. Damerau, F. J. (1964). A technique for computer detection and correction of spelling errors, *Communications of the ACM* **7**, 3, pp. 171–176.
21. De La Briandais, R. (1959). File searching using variable length keys, in *IRE-AIEE-ACM '59 (Western): Papers presented at the the March 3-5, 1959, western joint computer conference* (ACM, New York, NY, USA), pp. 295–298, doi:http://doi.acm.org/10.1145/1457838.1457895.
22. Fredkin, E. (1960). Trie memory, *Commun. ACM* **3**, 9, pp. 490–499, doi: http://doi.acm.org/10.1145/367390.367400.
23. Georgiev, K. (2007). Compression of Minimal Acyclic Deterministic FSAs Preserving the Linear Accepting Complexity. in *Proceedings of the Workshop on Finite-State Techniques and Approximate Search 2007, Borovets, Bulgaria*, pp. 7–13.
24. Glushkov, V. M. (1961). The abstract theory of automata, *Russian Mathematical Surveys* **16**, pp. 1–53.
25. Gonnet, G. H. and Baeza-Yates, R. (1991). *Handbook of algorithms and data structures: in Pascal and C (2nd ed.)* (Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA), ISBN 0-201-41607-7.
26. Graña, J., Barcala, F. M. and Alonso, M. A. (2002). Compilation Methods of Minimal Acyclic Automata for Large Dictionaries, *Lecture Notes in Computer Scuence - Implementation and Application of Automata* **2494**, pp. 135–148.
27. Gusfield, D. (1997). *Algorithms on Strings, Trees, and Sequences* (Cambridge University Press).
28. Hopcroft, J. E. (1971). An n log n algorithm for minimizing the states in a finite automaton, in Z. Kohavi (ed.), *The Theory of Machines and Computations* (Academic Press), pp. 189–196.

29. Jurafsky, D. and Martin, J. H. (2008). *Speech and Language Processing*, 2nd edn. (Prentice Hall).
30. Kaplan, R. M. and Kay, M. (1994). Regular models of phonological rule systems, *Computational Linguistics* **20**, 3, pp. 331–378.
31. Karttunen, L. (1994). Constructing lexical transducers, in *COLING-94* (Kyoto, Japan), pp. 406–411.
32. Karttunen, L. and Beesley, K. (2001). A short history of two-level morphology, in *Twenty Years of Two-Level Morphology. ESSLLI 2001 Special Event organised by Lauri Karttunen, Kimmo Koskenniemi and Gertjan van Noord.*
33. Knuth, D. E. (1973). *The Art of Computer Programming, Volume III: Sorting and Searching* (Addison-Wesley), ISBN 0-201-03803-X.
34. Koskenniemi, K. (1983). Two-level morphology: a general computational model for word-form recognition and production, Tech. Rep. 11, Department of General Linguistics, University of Helsinki.
35. Koskenniemi, K. (1984). A general computational model for word-form recognition and production, in *COLING-84* (Association for Computational Linguistics, Stanford University, California, USA), pp. 178–181.
36. Kowaltowski, T., Lucchesi, C. L. and Stolfi, J. (1993). Application of finite automata in debugging natural language vocabularies, in *First South American String Processing Workshop* (Belo Horizonte, Brasil).
37. Kowaltowski, T., Lucchesi, C. L. and Stolfi, J. (1998). Finite automata and efficient lexicon implementation, Tech. Rep. IC-98-02, icunicamp.
38. Kukich, K. (1992). Techniques for automatically correcting words in text, *ACM Comput. Surv.* **24**, 4, pp. 377–439.
39. Lucchesi, C. L. and Kowaltowski, T. (1993). Applications of finite automata representing large vocabularies, *Softw., Pract. Exper.* **23**, 1, pp. 15–30.
40. Lucchiesi, C. and Kowaltowski, T. (1993). Applications of finite automata representing large vocabularies, *Software Practice and Experience* **23**, 1, pp. 15–30.
41. Maurel, D. (2000). Pseudo-minimal transducer, *Theoretical Computer Science* **231**, 1, pp. 129–139.
42. McNaughton, R. and Yamada, H. (1960). Regular expressions and state graphs for automata, *IEEE Transactions on Electronic Computers* **9**, pp. 39–47.
43. Mihov, S. and Schulz, K. U. (2004). Fast approximate search in large dictionaries, *Computational Linguistics* **30**, 4, pp. 451–477.
44. Mohri, M. (1997). Finite-state transducers in language and speech processing, *Computational Linguistics* **23**, 2, pp. 269–311.
45. Ngassam, E. K., Watson, B. and Kourie, D. (2003). Hardcoding finite state automata processing, in *SAICSIT '03: Proceedings of the 2003 annual research conference of the South African institute of computer scientists and information technologists on Enablement through technology* (South African Institute for Computer Scientists and Information Technologists, Republic of South Africa), ISBN 1-58113-774-5, pp. 111–121.
46. Oflazer, K. (1996). Error-tolerant finite state recognition with applications to morphological analysis and spelling correction, *Computational Linguistics* **22**, 1, pp. 73–89.

47. Oflazer, K. and Güzey, C. (1994). Spelling correction in agglutinative languages, in *4th Conference on Applied Natural Language Processing* (Stuttgart, Germany), pp. 194–195.
48. Piskorski, J. (2005). On Compact Storage Models for Gazetteers, in A. Yli-Jyrä, L. Karttunen and J. Karhumäki (eds.), *Proceedings of the* 5^{th} *International Workshop on Finite-State Methods and Natural Language Processing (FSMNLP 2005)* (LNAI 4002, Springer, Helsinki, Finland), pp. 227–238.
49. Revuz, D. (1991). *Dictionnaires et lexiques: méthodes et algorithmes*, Ph.D. thesis, Institut Blaise Pascal, Paris, France, lITP 91.44.
50. Revuz, D. (1992). Minimisation of acyclic deterministic automata in linear time, *Theoretical Computer Science* **92**, 1, pp. 181–189.
51. Revuz, D. (2000). Dynamic acyclic minimal automaton, in *CIAA 2000, Fifth International Conference on Implementation and Application of Automata* (London, Canada), pp. 226–232.
52. Ristov, S. (1995). Space Saving with Compressed Trie Format, in D. Kalpić and V. Hljuz Dobrić (eds.), *Proceedings of the* 17^{th} *International Conference on Information Technology Interfaces ITI95, Pula, Croatia*, pp. 269–274.
53. Ristov, S. (2005). LZ Trie and Dictionary Compression, *Software - Practice & Experience* **35**, pp. 445–465.
54. Ristov, S. and Laporte, E. (1999). Ziv Lempel Compression of Huge Natural Language Data Tries Using Suffix Arrays, in M. Crochemore and M. Paterson (eds.), *Proceedings of Combinatorial Pattern Matching (CPM 1999), LNCS 1645* (Springer), pp. 196–211.
55. Roche, E. and Schabes, Y. (eds.) (1997). *Finite-State Language Processing*, Bradford Book (MIT Press, Cambridge, Massachusetts, USA).
56. Sgarbas, K., Fakotakis, N. and Kokkinakis, G. (1995). Two algorithms for incremental construction of directed acyclic word graphs, *International Journal on Artificial Intelligence Tools, World Scientific* **4**, 3, pp. 369–381.
57. Thompson, K. (1968). Regular expression search algorithm, *Communications of the ACM* **11**, pp. 419–422.
58. Tounsi, L., Bouchou, B. and Maurel, D. (2008). A Compression Method for Natural Language Automata, in J. Piskorski, B. Watson and A. Yli-Jyr (eds.), *Proceedings of the* 7^{th} *International Workshop on Finite-State Methods and Natural Language Processing, Ispra, Italy*, pp. 146–157.
59. Vilares, M., Otero, J. and na, J. G. (2005). Regional vs. global finite-state error repair, in A. Gelbukh (ed.), *Computational Linguistics and Intelligent Text Processing*, *Lecture Notes in Computer Science*, Vol. 3406 (Springer Verlag), pp. 120–131.
60. Vilares, M., Otero, J. and Vilares, J. (2006). Robust spelling correction, in J. Farré, I. Litovsky and S. Schmitz (eds.), *Implementation and Application of Automata*, *Lecture Notes in Computer Science*, Vol. 3845 (Springer Verlag), pp. 319–328.
61. Vo, B. and Vo, K.-P. (2004). Using Column Dependency to Compress Tables, in *Proceedings of the 2004 IEEE Data Compression Conference* (IEEE Computer Society Press, Los Alamitos, California), pp. 92–101.

62. Watson, B. (1998). A fast new (semi-incremental) algorithm for the construction of minimal acyclic DFAs, in *Third Workshop on Implementing Automata* (Lecture Notes in Computer Science, Springer, Rouen, France), pp. 91–98.
63. Watson, B. W. and Daciuk, J. (2003). An efficient incremental DFA minimization algorithm, *Natural Language Engineering* **9**, 1, pp. 49–64.

Chapter 5

Tree-Language Based Querying of Hierarchically Structured and Semi-Structured Data

Alexandru Berlea
SAP Research,
Bleichstraße 8, 64283 Darmstadt, Germany,
E-mail: berlea@googlemail.com

5.1 Introduction

The interest in tree language theory has significantly increased recently as result of the proliferation of XML in virtually any area of automatic data processing and due to the fact that the XML specification [81] essentially consists of a syntax for sequentially representing trees and a formalism for defining tree languages.

Results from the tree theory that have been known long before the emergence XML have helped devising the XML specification and XML related applications. The results presented here, on the other hand, are in line with research work that has been driven by the new needs of these applications. Correspondingly, our motivation is, as introduced next, drawn from the XML world and we have an eye for the practical related aspects; the results themselves are nevertheless generally applicable to tree-structured data, rather than being specific to the semi-structured, XML world.

5.1.1 *Motivation*

One advantage of XML is that it facilitates exchanging information in a standardized way, such that communicating entities are coupled as loosely as possible in order to achieve better interoperability. XML has become almost indispensable as an exchange format between communicating applications.

The second major advantage of XML is that it makes possible the separation of information content from information presentation. This is highly desirable when different presentations of the same content are needed, for instance XHTML for presence on the Internet, and Printable Document Format (PDF) for documents to be printed. Maintaining a separate document for each layout is inconvenient, due to the overhead required to keep all copies consistent whenever the informational content changes. Instead, one can maintain a single document containing the information represented in an XML language and one transformation program from the XML language to each desired layout. Thereby, keeping a layout up to date simply requires the running of the corresponding transformation program on the content whenever this changes.

Exploiting the advantages of using XML thus intrinsically requires the ability to transform XML documents, either to convert them from and to the exchange formats, or to produce a desired layout. A fundamental task thereby is *querying*, i.e. locating sub-components with some specified properties to be used for creating new content in XML transformations. Additionally, querying is used on its own in order to extract information from documents. The importance of query-languages becomes apparent if one notes that XPath [83], the XML query language proposed by the W3C Consortium, is integral part of many other important specifications, for example XML Schema Language [82], XSLT [86] or XQuery [85].

5.1.2 *Scope and Outline*

In this chapter we present how *tree languages* can be used for specifying powerful queries for tree-structured documents, and for this matter XML data, and show how these queries are efficiently implementable using (tree) automata. In particular, we address the following querying aspects:

Specifying *k*-ary queries Most of the existing XML-query languages identify only individual locations in the XML input data. We introduce a very expressive method that allows the specification of k-ary queries, that is, queries retrieving k locations that are in a specified context. The method is based on grammars, which are already applied in XML, but mainly only as schema languages. We call these queries k-ary grammar queries.

Efficient implementation of binary queries We identify binary queries (obtained for $k = 2$) as an important special case, particularly in view of their use in transformations. We introduce a tree-automata based algorithm which allows the efficient evaluation of binary grammar queries.

Query evaluation on XML streams Some XML documents may be very large, which makes it prohibitively expensive to keep them completely in the main memory while processing them. Also, there are increasingly many real-life applications in which the document to be processed is received linearly via some communication channel – as an XML stream –, rather than being completely available in advance. Special algorithms have to be developed to cope with these constraints. We show how grammar queries on XML streams can be answered by providing an algorithm which is very efficient in terms of time and highly adaptive in terms of memory consumption.

Practical application The viability of the algorithms and ideas presented in this part has been put to work in the XML querying tool Fxgrep [54], which provides access to the powerful grammar querying formalism via a more intuitive, and thus more user-friendly, specification language.

The chapter is organized as follows. In Section 5.2 we introduce terminology, definitions, notations and a classic automata construction which are used throughout the chapter. The forest grammars used for specifying queries, the languages specified by them, and their recognition are presented in Section 5.3. Section 5.4 presents how forest grammars can be used to specify k-ary queries and how these queries, in particular the binary ones, can be evaluated. Fxgrep is introduced in Section 5.5. Answering queries on XML streams is presented in Section 5.6.

5.2 Preliminaries

Hierarchically structured information, such as XML documents, can be conceptually represented as trees. XML processing is thus basically tree processing. An even more basic task is string processing. Regular expressions are an intuitive yet quite expressive way of specifying properties of strings. Furthermore, they are at the basis of more elaborated patterns to be located in trees that are presented in the following section. Therefore, we start in Section 5.2.1 with a presentation of regular expressions and the classic Berry-Sethi automata construction checking the conformance to a specified regular expression. In Section 5.2.2 we then introduce trees and a couple of related definitions and notations which will be used throughout the remainder of the chapter. In Section 5.2.3 we briefly present the basic XML concepts, notations and terminology and relate them to the usual tree terminology.

5.2.1 *Regular Expressions*

Let Σ be a finite set. We call Σ *alphabet* and its elements *symbols.* The number of elements of Σ is denoted as $|\Sigma|$. The set Σ^* of *strings* w over the alphabet Σ is defined as follows:

$$w \in \Sigma^* \text{ iff } w = \lambda \text{ or } w = aw_1 \text{ with } a \in \Sigma \text{ and } w_1 \in \Sigma^*$$

where λ is the *empty string.*

The set of *regular expressions* over the alphabet Σ denoted as $\mathcal{R}_\Sigma$ is defined as:

$$\begin{aligned} r \in \mathcal{R}_\Sigma \text{ iff } & r = \varnothing, \text{ or} \\ & r = \lambda, \text{ or} \\ & r = a \text{ and } a \in \Sigma, \text{ or} \\ & r = r_1? \text{ and } r_1 \in \mathcal{R}_\Sigma, \text{ or} \\ & r = {r_1}^* \text{ and } r_1 \in \mathcal{R}_\Sigma, \text{ or} \\ & r = r_1 r_2 \text{ and } r_1, r_2 \in \mathcal{R}_\Sigma, \text{ or} \\ & r = r_1 \,|\, r_2 \text{ and } r_1, r_2 \in \mathcal{R}_\Sigma. \end{aligned}$$

Parentheses may be omitted, in which case the composition of a regular expression is given by using the following operator precedence: ?, *, concatenation and $|$, from the strongest to the weakest. The number of occurrences of symbols from Σ in a regular expression r is denoted as $|r|_\Sigma$.

The *language of a regular expression* $r \in \mathcal{R}_\Sigma$ is a set $[\![r]\!] \subseteq \Sigma^*$ defined as follows:

$$\begin{aligned} [\![\varnothing]\!] &= \varnothing \\ [\![\lambda]\!] &= \{\lambda\} \\ [\![a]\!] &= \{a\}, \text{ for all } a \in \Sigma \\ [\![r?]\!] &= \{\lambda\} \cup [\![r]\!] \\ [\![r^*]\!] &= \{\lambda\} \cup \{w_1 \dots w_n \mid n > 0, w_i \in [\![r]\!] \text{ for all } 1 \le i \le n\} \\ [\![r_1 r_2]\!] &= \{w_1 w_2 \mid w_1 \in [\![r_1]\!], w_2 \in [\![r_2]\!]\} \\ [\![r_1 \mid r_2]\!] &= [\![r_1]\!] \cup [\![r_2]\!] \end{aligned}$$

A language is called a *regular string language* if it is the language of a regular expression.

Finite Automata

The membership of a string to a regular string language can be tested by a finite automaton. A *finite automaton* over an alphabet Σ is a tuple $A = (Q, q_0, F, \delta)$ consisting of a set of *states* Q, an *initial state* $q_0 \in Q$, a set of *final states* $F \subseteq Q$ and a *transition relation* $\delta \subseteq Q \times \Sigma \times Q$. The language $\mathcal{L}_A$ accepted by the automaton A is defined as follows:

$$\begin{array}{lll} \lambda \in \mathcal{L}_A & \text{iff} & q_0 \in F \\ a_1 \ldots a_n \in \mathcal{L}_A & \text{iff} & \text{there are } q_1, \ldots, q_{n+1} \in Q \text{ such that } q_1 = q_0 \text{ and} \\ & & (q_i, a_i, q_{i+1}) \in \delta \text{ for all } 1 \leq i \leq n. \end{array}$$

If δ is a function, rather than a relation, A is called *deterministic finite automaton* (DFA). Otherwise, A is called *non-deterministic finite automaton* (NFA).

The Berry-Sethi Construction

One method to construct an NFA accepting the language of a regular expression is the algorithm proposed by Berry and Sethi [10]. Given a regular expression r this constructs an NFA, $\mathit{Berry}(r) = (Q, q_0, F, \delta)$, accepting exactly the language $[\![r]\!]$ as follows.

If $r = \lambda$ then $\mathit{Berry}(r) = (\{q_0\}, q_0, \{q_0\}, \varnothing)$ where q_0 is some arbitrarily chosen state. If $r = \varnothing$ then $\mathit{Berry}(r) = (\{q_0\}, q_0, \varnothing, \varnothing)$ where q_0 is some arbitrarily chosen state.

Otherwise, a set P of positions p is generated s.t. $|P| = |r|_\Sigma$. Further, a bijection f from the set of occurrences of symbols in r into P is defined. A regular expression $\bar{r} \in \mathcal{R}_P$ is constructed by replacing each occurrence o with $f(o)$. Then, for each subexpression $\bar{r}_1$ of $\bar{r}$ the following information is computed in the given order.

(1) $\mathit{Empty}(\bar{r}_1)$, denoting whether $\lambda \in [\![\bar{r}_1]\!]$, given as follows:

$$\begin{array}{ll} \mathit{Empty}(p) & = \mathit{false} \\ \mathit{Empty}(r_1?) & = \mathit{true} \\ \mathit{Empty}(r_1{}^*) & = \mathit{true} \\ \mathit{Empty}(r_1 r_2) & = \mathit{Empty}(r_1) \text{ and } \mathit{Empty}(r_2) \\ \mathit{Empty}(r_1 \mid r_2) & = \mathit{Empty}(r_1) \text{ or } \mathit{Empty}(r_2) \end{array}$$

(2) $First(\bar{r}_1)$, denoting the symbols with which strings from $[\![\bar{r}_1]\!]$ may start:

$$\begin{aligned}
&First(p) &&= \{p\}\\
&First(r_1?) &&= First(r_1)\\
&First(r_1{}^*) &&= First(r_1)\\
&First(r_1 \,|\, r_2) &&= First(r_1) \cup First(r_2)\\
&First(r_1 r_2) &&= First(r_1) \cup \begin{cases} First(r_2), & \text{if } Empty(r_1)\\ \varnothing \quad\quad\quad\;\;, & \text{otherwise}\end{cases}
\end{aligned}$$

(3) $Follow(\bar{r}_1)$, denoting the symbols which can immediately follow after a string w from $[\![\bar{r}_1]\!]$ within a string from $[\![\bar{r}]\!]$ or \$ (an auxiliary symbol) if w might be a suffix of a string in $[\![\bar{r}]\!]$:

If $\bar{r}_1 = \bar{r}$ then $Follow(\bar{r}_1) = \{\$\}$
If $\bar{r}_1 = r_1?$ then $Follow(r_1) = Follow(\bar{r}_1)$
If $\bar{r}_1 = r_1{}^*$ then $Follow(r_1) = Follow(\bar{r}_1) \cup First(r_1)$
If $\bar{r}_1 = r_1 \,|\, r_2$ then $Follow(r_1) = Follow(r_2) = Follow(\bar{r}_1)$

If $\bar{r}_1 = r_1 r_2$ then

$$\begin{aligned}
&Follow(r_2) = Follow(\bar{r}_1)\\
&Follow(r_1) = First(r_2) \cup \begin{cases} Follow(\bar{r}_1), & \text{if } Empty(r_2)\\ \varnothing \quad\quad\quad\quad\;\;, & \text{otherwise}\end{cases}
\end{aligned}$$

Given the definitions above, $Empty()$ and $First()$ can be computed in a bottom-up while $Follow()$ can be computed in a top-down manner. Using $Empty()$, $First()$ and $Follow()$, the NFA is defined as follows. The set of states is $Q = \{q_0\} \cup P$ with some arbitrarily chosen start state $q_0 \notin P$. The set F of final states is obtained as:

$$F = \begin{cases} \{p \in P \mid \$ \in Follow(p)\} \cup q_0, & \text{if } Empty(\bar{r})\\ \{p \in P \mid \$ \in Follow(p)\} \quad\quad\;\;, & \text{otherwise}\end{cases}$$

Let *sym* be the inverse of function f, i.e. the mapping of each position p to the symbol occurring at $f^{-1}(p)$. The transition relation is given by:

$$\begin{aligned}
\delta = \;& \{(q_0, sym(p), p) \mid p \in First(\bar{r})\} \cup\\
& \{(p, sym(p_1), p_1) \mid p, p_1 \in P, p_1 \in Follow(p)\}
\end{aligned}$$

Example 5.1. Consider the regular expression $r = a^*(a \,|\, b)b^*$. We choose as the set of positions $P = \{1, 2, 3, 4\}$ and associate i with the i-th symbol occurrence, hence $sym = \{(1, a), (2, a), (3, b), (4, b)\}$ and $\bar{r} = 1^*(2 \,|\, 3)4^*$. The syntax tree of $\bar{r}$ is depicted in Figure 5.1. The internal nodes of the tree denote the subexpressions of $\bar{r}$. The Berry-Sethi construction proceeds as follows.

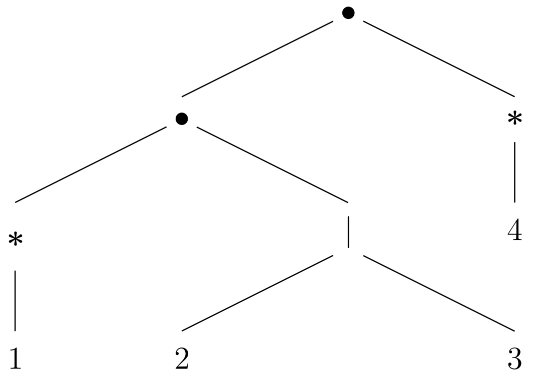


Fig. 5.1 Syntax tree of the regular expression $1^*(2\,|\,3)4^*$.

(1) $Empty(1) = Empty(2) = Empty(3) = Empty(4) = \mathit{false}$
$Empty(1^*) = Empty(4^*) = \mathit{true}$
$Empty(2\,|\,3) = \mathit{false}$
$Empty(1^*(2\,|\,3)) = \mathit{false}$
$Empty(1^*(2\,|\,3)4^*) = \mathit{false}$

(2) $First(i) = \{i\}$ for all $1 \leq i \leq 4$
$First(1^*) = \{1\},\ First(4^*) = \{4\}$
$First(2\,|\,3) = \{2,3\}$
$First(1^*(2\,|\,3)) = \{1,2,3\}$
$First(1^*(2\,|\,3)4^*) = \{1,2,3\}$

(3) $Follow(1^*(2\,|\,3)4^*) = \{\$\}$
$Follow(1^*(2\,|\,3)) = \{4,\$\},\ Follow(4^*) = \{\$\}$
$Follow(4) = \{4,\$\}$
$Follow(1^*) = \{2,3\},\ Follow(2\,|\,3) = \{4,\$\}$
$Follow(1) = \{1,2,3\},\ Follow(2) = \{4,\$\},\ Follow(3) = \{4,\$\}$

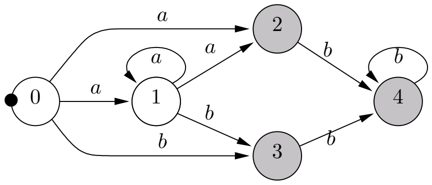


Fig. 5.2 NFA obtained by the Berry-Sethi construction for $a^*(a\,|\,b)b^*$.

By choosing $q_0 = 0$ it follows that $Q = \{0,1,2,3,4\}$, $F = \{2,3,4\}$ and $\delta = \{(0,a,1),\ (0,a,2),\ (0,b,3),\ (1,a,1),\ (1,a,2),\ (1,b,3),\ (2,b,4),\ (3,b,4),\ (4,b,4)\}$. The obtained NFA is depicted in Figure 5.2, where the initial state is marked by the $\bullet$ symbol and final states are depicted in gray.

An NFA obtained by the Berry-Sethi construction has the important property that all transitions coming into the same state are labeled by the same symbol. We denote the label of the incoming transitions into an NFA state y by $in(y)$.

5.2.2 *Trees and Forests*

Let Σ be a set that we call *alphabet*. The sets $\mathcal{T}_\Sigma$ of trees t and $\mathcal{F}_\Sigma$ of *forests* f over Σ is defined as follows:

$$t \in \mathcal{T}_\Sigma \quad \text{iff} \quad t = a\langle f\rangle \text{ with } a \in \Sigma \text{ and } f \in \mathcal{F}_\Sigma$$

$$f \in \mathcal{F}_\Sigma \quad \text{iff} \quad f = \varepsilon \text{ or } f = tf_1 \text{ with } t \in \mathcal{T}_\Sigma \text{ and } f_1 \in \mathcal{F}_\Sigma$$

where ε denotes the *empty forest*. Given $t = a\langle f\rangle$, the symbol a denotes the *label* and f the *children* of t. To denote the label of t we also write $lab(t) = a$.

A tree $a\langle\varepsilon\rangle$ is called a *leaf* and may be denoted by $a\langle\rangle$ or simply by a. Also, rather than $t\varepsilon$ or εt, we write t. The notation t can be thus interpreted both as a tree or a forest consisting of exactly one tree. Both interpretations are valid in most usage contexts. We will explicitly note the intended interpretation when the distinction is relevant and if it is not obvious from the context.

Let $t = a\langle t_1 \dots t_n\rangle$. The trees t_i are the *children* of t, while t is the *father* of all t_i trees for $1 \le i \le n$. Two trees t_i and t_j with $1 \le i, j \le n$ and $i \ne j$ are called *siblings*. If $i < j$ then t_i is a *left sibling* of t_j and t_j is a *right sibling* of t_i.

Note that our trees are *unranked*, that is the sequence f of children of a tree $a\langle f\rangle$ may have an arbitrary length. We could have used as well a ranked representation like in the traditional tree theory, as each tree or forest can be reversibly encoded into a unique ranked tree (see for example [53]). Working with the encoded representations however complicates both the operations on trees and forest and the intuitions behind them, hence we preferred the straightforward unranked representation. Also note that the trees defines above are *ordered*, i.e. the order of the children is relevant.

Nodes, Paths and Locations

Any subtree t of a forest f is uniquely identified by a *node*. A node is a string of natural numbers, denoting the *path* leading to t, formally defined as follows. The set $\Pi(f) \subseteq \mathbb{N}^*$ contains all *paths* π in f and is defined as follows:

$$\Pi(\varepsilon) = \{\lambda\}$$
$$\Pi(t_1 \ldots t_n) = \{\lambda\} \cup \{i\pi \mid 1 \leq i \leq n, \pi \in \Pi(f_i) \text{ for } t_i = a_i\langle f_i\rangle\}$$

where $\mathbb{N}^*$ is the set of strings over the alphabet of positive natural numbers and λ denotes the empty string.

The *nodes* of a forest f are elements of the set $N(f) = \Pi(f) \setminus \{\lambda\}$. For $\pi \in N(f)$, $f[\pi]$ is called the *subtree of f located at π* and is defined as follows:

$$(t_1 \ldots t_n)[i\pi] = \begin{cases} t_i \quad , & \text{if } \pi = \lambda \\ f_i[\pi], & \text{if } \pi \neq \lambda \text{ and } t_i = a\langle f_i\rangle \end{cases}$$

For a node π, we define $last_f(\pi)$ as the number of children of π:

$$last_f(\pi) = max(\{n \mid \pi n \in N(f)\} \cup \{0\})$$

with $last_f(\pi) = 0$ iff π identifies a leaf.

Note that a path always locates a tree in a *forest*, not in a tree. Given a tree t, $t[\pi]$ denotes the tree located by π in the forest which consists of t only. One can see by definition that in this case π always begins with the symbol 1. In particular, one can use the subtree $t = f[\pi_1]$ located by a path π_1 in a forest f to further locate a subtree of t. In this case we have that $f[\pi_1][1\pi_2] = f[\pi_1\pi_2]$.

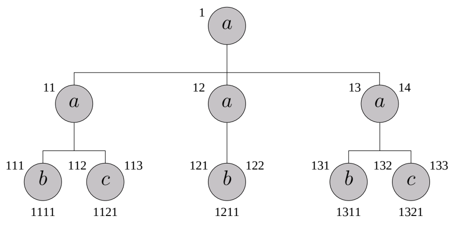


Fig. 5.3 Locations in a tree.

The *document order* is defined as the lexicographic order of the nodes of a forest f. Note that this is precisely the order in which the nodes are visited during a left-to-right depth-first search (DFS) traversal of f. Sometimes we need to precisely identify the locations reached during a DFS traversal of a forest. To this aim we define the set $L(f) \subseteq \mathbb{N}^*$ of *locations* in a forest f as:

$$\begin{aligned} L(\varepsilon) &= \{1\} \\ L(t_1 \ldots t_n) &= \{i \mid 1 \leq i \leq n+1\} \cup \\ &\quad\ \{il \mid 1 \leq i \leq n, l \in L(f_i) \text{ for } t_i = a_i\langle f_i\rangle\} \end{aligned}$$

Figure 5.3 depicts these locations in a sample tree. The location at which the root of a subtree is reached (depicted to its left) equals the node at which the subtree is located.

5.2.3 *XML Basics*

This section is only meant to briefly introduce the essential XML constructs and to relate the XML terminology to the tree terminology. For a thorough introduction to XML we refer to the books dedicated to this subject, such as to [35].

Basically, an XML document is a serial representation of an ordered, unranked, labeled tree. The XML representation of a tree $a\langle f\rangle$ is an XML *element* given as $serialize(a\langle f\rangle)$, where:

$$\begin{array}{lcl} serialize(a\langle f\rangle) & ::= & \texttt{<a>}serialize(f)\texttt{</a>} \\ serialize(t_1 \ldots t_n) & ::= & serialize(t_1) \ldots serialize(t_n) \\ serialize(\varepsilon) & ::= & \lambda \end{array}$$

For example, $a\langle b\langle c\rangle d\rangle$ can be denoted in XML as `<a><b><c></c></b><d></d></a>` or using (irrelevant) white spaces for enhanced readability:

```
<a>
  <b>
    <c></c>
  </b>
  <d></d>
</a>
```

Consider a tree $a\langle f\rangle$ and the corresponding XML element `<a>`$serialize(f)$ `</a>`. The XML terminology denominates the symbol `a` *tag* or *element name*, `<a>` *start tag*, `</a>` *end tag* and $serialize(f)$ *element content*. An element with empty content `<a></a>` is called *empty element* and might be as well denoted as `<a/>`. The element corresponding to the root of the top-level tree is the *root element*.

Additionally, XML elements can be provided with named properties via *attributes*. An attribute is a pair consisting of an *attribute name* and an *attribute value* given as arbitrary sequences of symbols. Attributes are specified along with the start tag of the element, after the tag name, as the attribute name followed by the "=" sign followed by the attribute value enclosed in single or double quotes, as for example in:

```
<circle radius='25' x="40" y='60' color='green'/>
```

Note that attributes do not add to the expressiveness of XML as they could also be represented by using dedicated element names, as for example:

```
<circle>
  <attributes>
    <name>radius</name><val>25</val>
    <name>x</name><val>40</val>
    <name>y</name><val>60</val>
    <name>color</name><val>green</val>
</circle>
```

Besides other elements, *text* and *processing instruction nodes* may occur anywhere within an element. A text node consists of a sequence of symbols which occur within the enclosing element. A processing instruction is intended to provide an *instruction* to some *target processor* of the XML representation and has the form `<?target attributes?>`. The attributes are as for elements and are to be interpreted by the target processor. Processing instructions are also allowed to occur before and after the root element. Therefore, an XML document is a forest rather than a tree as the root element might be preceded and followed by processing instruction nodes.

For example an XSL-enabled browser uses the processing instruction at the beginning of the following XML document:

```
<?xml-stylesheet type="text/xsl" href="program2html.xsl"?>
<Program>
  <Output>Hello World!<newline/>Und Tschüss!</Output>
</Program>
```

to retrieve the stylesheet `program2html.xsl` and apply it to the document in order to obtain its Web presentation.

Similarly to attributes, processing instructions do not actually add to expressiveness, as their information can be provided via elements with a dedicated name.

An element like `Output` in XML Example 4 which encompasses both element and text nodes is said to have *mixed content*, while an element with only text nodes is said to have *text content*.

XML Schema Languages

The XML specification includes a method for specifying structural constraints for XML documents. The set of documents adhering to a set of given constraints is called an XML *language.* An XML language is defined using a *document type definition* (DTD). An XML document can be declared as belonging to an XML language by providing it with a *document type declaration.* The document type declaration indicates the root element and either directly provides the DTD, in which case the DTD is called *internal*, or it provides a reference to an *external* location where the DTD is to be found as for example in:

```
<!DOCTYPE dblp SYSTEM "dblp.dtd">
```

saying that the DTD is given in the file named `dblp.dtd`.

An XML document that conforms to its declared DTD is called *valid.* Checking validity of XML documents is achieved by XML validating parsers. Checking the validity of XML is very important for applications that rely on a specific format for their XML input, especially if the source of the input is not controllable, as it is often the case in highly dynamic settings.

A DTD consists of declarations restricting the content of the elements occurring in XML documents conforming to the DTD. The content of an element might be restricted depending on the element's name. One can either specify that an element should have only text content, or mixed content, or give a *content-model* for it. A content model specified in a DTD is a regular expression over element names which has to be fulfilled by the string of element names of the enclosed elements. For example:

```
<!ELEMENT ulist (item+)>
```

specifies that an element named `ulist` must consist of one or more `item` elements. Furthermore one can specify which are the attributes that an element might have and whether they are required, optional, or that they have some fixed value.

Even though DTDs are the only means of specifying XML languages anchored in the XML specification, they are just one way of doing so. The languages used to specify XML languages are called XML *schema languages*, as they specify a *schema* to which the XML documents belonging to the XML language must conform. The structural constraints expressible with the proposed schema languages are in general more precise than those allowed by DTDs, and are basically subsumed by the capabilities of forest grammars, which will be introduced in Section 5.3.1.1. A comparison of forest grammars and the most frequently used schema languages is presented in the Section 5.3.1.3.

5.3 Regular Forest Languages

Specifying and checking conformance of XML documents to a schema, i.e. their membership to an XML language, is a very important task for XML processing. Since the introduction of DTDs as a basic schema language in the XML specification [81], more powerful schema languages have been defined, which allow a more precise specification of XML languages. Among the better known are XML Schema Language [82], DSD [39] and RelaxNG [65].

The main purpose of schema languages is to specify the structure of the documents conforming to the defined XML language. The structural properties of XML languages specifiable using the various proposed schema languages are essentially captured by *regular forest languages*. That is, XML languages are essentially regular forest languages. Correspondingly, checking conformance to a schema basically means testing membership in a regular forest language.

Since the structural conditions expressible with regular forest languages are at the basis of the querying techniques presented in this chapter, we briefly review how these can be specified, in Section 5.3.1, and recognized, in Section 5.3.2.

5.3.1 *Specifying Regular Forest Languages*

Regular forest languages constitute a very expressive and theoretically robust formalism for specifying properties of forests. One way of specifying regular forest languages is by using forest grammars. In fact, the proposed XML schema languages essentially specify more or less restricted forms of forest grammars. The relation between forest grammars and XML schema languages is discussed in Section 5.3.1.3. The reason why forest grammars are chosen among the other possibilities for specifying regular forest languages is that, in our opinion, they are more comprehensible than the other formalisms.

5.3.1.1 *Forest Grammars*

A *forest grammar* is a tuple $G = (\Sigma, X, R, r_0)$, where Σ and X are alphabets of terminal and non-terminal symbols, respectively, $R \subseteq X \times \Sigma \times \mathcal{R}_X$ is a set of productions and $r_0 \in \mathcal{R}_X$ is the *start expression*[1]. As Σ and X are visible from the set of productions R we omit them when there is no risk of confusion and write $G = (R, r_0)$. We denote a production $(x, a, r) \in R$ as $x \to a\langle r\rangle$. We write $x \to a$ rather than $x \to a\langle\lambda\rangle$.

Intuitively, and using the terminology from schema languages, a production $x \to a\langle r\rangle$ specifies that the children of an a element derived using the production must conform to the *content model* r. Also, the start expression is a content model which must be fulfilled by the sequence consisting of the root element and the possible preceding and following processing instructions.

Example 5.2. Consider for example the following excerpt from a file `sample.dtd` containing a DTD for books:

```
<!ELEMENT BOOK (TITLE, SUBTITLE?, CHAPTER+, APPENDIX?)>
<!ELEMENT CHAPTER (TITLE, (CHAPTER|PAR)+)>
<!ELEMENT APPENDIX (CHAPTER+)>
```

Further suppose that the root element is `BOOK` as declared in the following document type declaration:

```
<!DOCTYPE BOOK SYSTEM "sample.dtd">
```

The same can be specified using a forest grammar with the following productions:

$$\begin{array}{lcl} x_{book} & \to & \mathtt{BOOK}\langle x_{title}\ x_{subtitle}^{?}\ x_{chapter}^{+}\ x_{appendix}^{?}\rangle \\ x_{chapter} & \to & \mathtt{CHAPTER}\langle x_{title}\ (x_{chapter}|x_{par})^{+}\rangle \\ x_{appendix} & \to & \mathtt{APPENDIX}\langle x_{chapter}^{+}\rangle \end{array}$$

We obtain the equivalent forest grammar by choosing as start expression x_{book}, corresponding to the root element in the DTD, and further assuming the presence in the grammar of productions for the non-terminals x_{title}, $x_{subtitle}$ and x_{par}, according to the DTD definitions of the elements `TITLE`, `SUBTITLE` and `PAR`, respectively.

In the following we give the formal definition of conformance to a schema specified by a forest grammar.

A set of productions R together with a distinguished non-terminal $x \in X$ or a regular expression $r \in \mathcal{R}_X$ defines a *tree derivation* relation

[1]Recall that $\mathcal{R}_X$ is the set of regular expressions over X.

$\mathcal{D}eriv_{R,x} \subseteq \mathcal{T}_\Sigma \times \mathcal{T}_X$ or a *forest derivation* relation $\mathcal{D}eriv_{R,r} \subseteq \mathcal{F}_\Sigma \times \mathcal{F}_X$, respectively, as follows:

$$\begin{array}{ll}
(a\langle f\rangle, x\langle f'\rangle) \in \mathcal{D}eriv_{R,x} & \text{iff } x \to a\langle r\rangle \in R \text{ and } (f, f') \in \mathcal{D}eriv_{R,r} \\
(t_1 \dots t_n, t'_1 \dots t'_n) \in \mathcal{D}eriv_{R,r} & \text{iff } x_1 \dots x_n \in [\![r]\!] \text{ and } (t_i, t'_i) \in \mathcal{D}eriv_{R,x_i} \\
 & \quad \text{for } i = 1, \dots, n \\
(\varepsilon, \varepsilon) \in \mathcal{D}eriv_{R,r} & \text{iff } \lambda \in [\![r]\!]
\end{array}$$

If $(f, f') \in \mathcal{D}eriv_{R,r}$, we say that f' is a *derivation* of f w.r.t. R and r. In the following we omit R when it is clear from the context which set of productions is meant. Given a grammar $G = (R, r)$ we write $(f, f') \in \mathcal{D}eriv_G$ iff $(f, f') \in \mathcal{D}eriv_{R,r}$ and say that f' is a derivation of f w.r.t. the grammar G.

If $(f, f') \in \mathcal{D}eriv_G$ for some f', then f conforms to the schema G. Observe that a derivation f' is a relabeling of f and can be seen as a proof of the validity of f according to the schema G. If $lab(f'[\pi]) = x$ we say that f' *labels* $f[\pi]$ with x.

Note also that forest grammars have been also called unranked tree or hedge automata elsewhere [13]. From this viewpoint, non-terminals are states, productions are transitions, and derivations are accepting runs of the automaton.

Example 5.3. Let R be the set of following productions:

$$\begin{array}{l}
x_a \to a\langle (x_a|x_b)^*\rangle \\
x_b \to b
\end{array}$$

Let $f = a\langle ab\rangle$ and suppose we want to check whether there is a derivation of f w.r.t. R and x_a. We can proceed in a bottom-up manner. It is easy to see that $(a, x_a) \in \mathcal{D}eriv_{x_a}$ and $(b, x_b) \in \mathcal{D}eriv_{x_b}$. Since $x_a x_b \in [\![(x_a|x_b)^*]\!]$ we have that $(ab, x_a x_b) \in \mathcal{D}eriv_{(x_a|x_b)^*}$. It follows that $(a\langle ab\rangle, x_a\langle x_a x_b\rangle) \in \mathcal{D}eriv_{x_a}$.

Example 5.4. Let R_2 be the set of following productions:

$$\begin{array}{lll}
(1)\ x_\top \to a\langle x_\top^*\rangle & (4)\ x_1 \to a\langle x_\top^*(x_1|x_a)x_\top^*\rangle & (6)\ x_b \to b\langle x_\top^*\rangle \\
(2)\ x_\top \to b\langle x_\top^*\rangle & (5)\ x_a \to a\langle x_b x_c\rangle & (7)\ x_c \to c\langle x_\top^*\rangle \\
(3)\ x_\top \to c\langle x_\top^*\rangle & &
\end{array}$$

Let t be the tree textually represented by the following XML document:

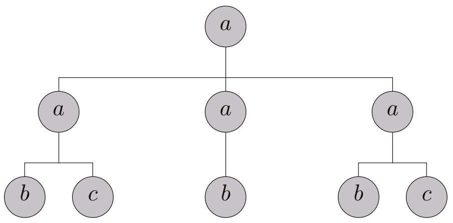


Fig. 5.4 The tree representation of t from Example 5.4.

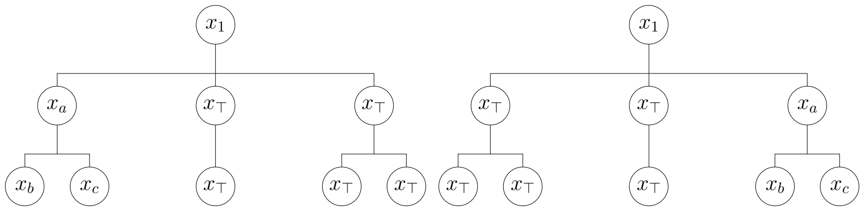


Fig. 5.5 Possible derivations of t from Example 5.4.

```
<a>
  <a><b/><c/></a>
  <a><b/></a>
  <a><b/><c/></a>
</a>
```

The tree t is graphically presented in Figure 5.4. Two possible derivations of t w.r.t. R and the regular expression $x_1|x_a$ are depicted in Figure 5.5.

The *meaning* $[\![R]\!]$ of a set of productions R assigns sets of trees to non-terminals $x \in X$ and sets of forests to regular expressions $r \in \mathcal{R}_X$ as follows:

$$
\begin{aligned}
&t \in [\![R]\!]\, x \text{ iff there is } t' \in \mathcal{T}_X \text{ with } (t, t') \in \mathcal{D}eriv_{R,x}\\
&f \in [\![R]\!]\, r \text{ iff there is } f' \in \mathcal{F}_X \text{ with } (f, f') \in \mathcal{D}eriv_{R,r}
\end{aligned}
$$

If $t \in [\![R]\!]\, x$ or $f \in [\![R]\!]\, r$ we say that t can be derived from x or f can be derived from r, respectively.

Example 5.5. Let R be the set of productions from Example 5.3. It is easy to see that $[\![R]\!]\, x_b$ is the set consisting only of the tree b. The set $[\![R]\!]\, x_a$ consists of all trees, the internal nodes of which are all labeled a, and the leaves of which are labeled either a or b.

The *regular forest language* specified by a forest grammar $G = (R, r_0)$ is the set of forests $\mathcal{L}_G = [\![R]\!]\ r_0$.

Example 5.6. Consider the grammar $G = (R_2, x_1|x_a)$ over $\{a, b, c\}$ with the productions R_2 as defined in Example 5.4.

$\mathcal{L}_G$ is the set of documents in which there is a path from the root to a node labeled a, whose children are a node labeled b and a node labeled c , and whose ancestors are all labeled a. The first three productions make $x_\top$ account for trees with arbitrary content. As specified by production (5), x_a stands for the a element with the b and the c children. Productions (6) and (7) say that these children can have arbitrary content. Finally, production (4) specifies that the a specified by (5) can be at arbitrary depth in the input, and all its ancestors must be labeled a.

5.3.1.2 *Practical Extensions*

To use forest grammars as a specification language in a practical setting, such as XML processing, a couple of useful extensions need to be made as presented below.

Text Nodes The grammar formalism as introduced is not yet able to handle XML documents in which elements have text content. Let $\mathcal{U}$ be the set of Unicode characters [78], the symbols allowed in text and mixed-content nodes of XML documents. Our definition of trees t and forests f in Section 5.2.2 can be adapted in order to allow XML text nodes as well as follows:

$$\begin{array}{lll} t \in \mathcal{T}_\Sigma & \text{iff} & t = a\langle f\rangle \text{ with } a \in \Sigma \text{ and } f \in \mathcal{F}_\Sigma \text{ or } t = \alpha^* \\ f \in \mathcal{F}_\Sigma & \text{iff} & f = \varepsilon \text{ or } f = tf_1 \text{ with } t \in \mathcal{T}_\Sigma \text{ and } f_1 \in \mathcal{F}_\Sigma \end{array}$$

where α^* denotes an arbitrary sequence of characters $\alpha \in \mathcal{U}$.

External Predicates To handle text nodes one can extend the definition of forest grammars to include a set of *external predicates* P. The purpose of external predicates is to express properties which cannot be captured via content models. In particular, an external predicate can test whether a node is a text node.

An external predicate $p \in P$ is a boolean function of type $\mathcal{T}_\Sigma \mapsto \mathcal{B}$ which takes a tree as argument and returns one of the two boolean values in $\mathcal{B}$, *true* or *false*. A *forest grammar with external predicates* is a tuple $G = (\Sigma, X, P, R, r_0)$ with Σ, X and r_0 as before and $R \subseteq (X \times \Sigma \times \mathcal{R}_X) \cup (X \times P)$. As before, we denote a production $(x, a, r) \in (X \times \Sigma \times \mathcal{R}_X)$

or $(x,p) \in (X \times P)$ as $x \to a\langle r\rangle$ or $x \to p$, respectively. Intuitively, a production $x \to p$ is applicable in a derivation of the tree t when the predicate p is true for t.

Formally, the *tree derivation* relation $\mathcal{D}eriv_{R,x} \in \mathcal{T}_\Sigma \times \mathcal{T}_X$ and the *forest derivation* relation $\mathcal{D}eriv_{R,r} \in \mathcal{F}_\Sigma \times \mathcal{F}_X$ defined by set of productions R together with a distinguished non-terminal $x \in X$ and a regular expression $r \in \mathcal{R}_X$ respectively, are correspondingly extended as follows:

$$\begin{array}{ll}
(a\langle f\rangle, x\langle f'\rangle) \in \mathcal{D}eriv_{R,x} & \text{iff } x \to a\langle r\rangle \in R \text{ and } (f,f') \in \mathcal{D}eriv_{R,r} \\
(t,x) \in \mathcal{D}eriv_{R,x} & \text{iff } x \to p \text{ and } p(t) = true \\
(t_1 \dots t_n, t'_1 \dots t'_n) \in \mathcal{D}eriv_{R,r} & \text{iff } x_1 \dots x_n \in [\![r]\!] \text{ and } (t_i,t'_i) \in \mathcal{D}eriv_{R,x_i} \\
 & \quad \text{for } i = 1, \dots, n \\
(\varepsilon,\varepsilon) \in \mathcal{D}eriv_{R,r} & \text{iff } \lambda \in [\![r]\!]
\end{array}$$

The *meaning* $[\![R]\!]$ of R is similarly given by:

$$\begin{array}{l}
t \in [\![R]\!]\, x \text{ iff there is } t' \in \mathcal{T}_X \text{ with } (t,t') \in \mathcal{D}eriv_{R,x} \\
f \in [\![R]\!]\, r \text{ iff there is } f' \in \mathcal{F}_X \text{ with } (f,f') \in \mathcal{D}eriv_{R,r}
\end{array}$$

Finally, the language of G is as before $\mathcal{L}_G = [\![R]\!]\, r_0$.

Thereby, one can express that a node within a content model is a text node by referring it via a new terminal x_{text} for which a production $x_{text} \to p$ exists where p is a predicate testing whether its argument is a sequence of Unicode characters, i.e.:

$$p(t) = \begin{cases} true\ , & \text{if } t = \alpha^* \\ false, & \text{otherwise} \end{cases}$$

In general, predicates can be used to specify arbitrary properties of subtrees which are not expressible with the original forest grammars formalism. For example, one can use them to express that some text nodes have a required datatype, as needed in XML schema languages like XML Schema [82] or RelaxNG [65].

Wild Card Symbols In a practical specification language one is often interested in merely indicating the occurrence of some entity without further specifying it. For this purpose special place-holder symbols, also called *wildcards*, have to be provided. In the forest grammar formalism we use the wildcard "*" to denote an arbitrary label of the node and "." to denote an arbitrary node. Furthermore, we use "_" as an abbreviation for ".*", to denote an arbitrary sequence of nodes.

Example 5.7. Let R be the following set of productions:

$$\begin{aligned} x_1 &\to a\langle\_\,(x_1|x_2)\,\_\rangle \\ x_2 &\to *\langle x_b\ .\rangle \\ x_b &\to b\langle\_\rangle \end{aligned}$$

The grammar $G = (R, x_1|x_2)$ specifies the language of documents in which there is a path from the root to a node the ancestor nodes of which are all labeled with a, and which has two children, the first a `b` node and the second an arbitrary node.

5.3.1.3 *Forest Grammars and XML Schema Languages*

As previously mentioned, XML schema languages basically specify forest grammars. There are however some differences in terms of expressiveness that we address here. Understanding the different expressive powers of the XML schema languages is one advantage of a language theoretic approach thereto. A taxonomy of XML schema languages obtained in this way can be consulted in [51]. Here we restrain only to addressing the main differences between forest grammars and the common XML schema languages.

DTDs, as opposed to forest grammars, do not allow the specification of context-dependent content models for elements – as presented in the following example:

Example 5.8. Consider the modification of the productions from Example 5.2 as follows:

$$\begin{array}{lcl} x_{book} & \to & \texttt{BOOK}\langle x_{title}\ x^?_{subtitle}\ x^+_{chapter_1}\ x^?_{appendix}\rangle \\ x_{chapter_1} & \to & \texttt{CHAPTER}\langle x_{title}\ (x_{chapter_2}|x_{par})^+\rangle \\ x_{chapter_2} & \to & \texttt{CHAPTER}\langle x_{title}\ x^+_{par}\rangle \\ x_{appendix} & \to & \texttt{APPENDIX}\langle x^+_{chapter_1}\rangle \end{array}$$

Note that we specify different content models for chapters on the top-level and chapters occurring inside other chapters. Rather than allowing arbitrarily nested chapters as in Example 5.2, the new productions only allow top-level chapters to contain sub-chapters. This is not possible to express with a DTD, where all elements with the same name must be associated with the same content model.

Another limitation of DTDs as compared to forest grammars is the requirement of content models to be *unambiguous*, i.e. that the corresponding finite string automata, as obtained by the Berry-Sethi construction, are deterministic. The restriction ensures that every word can be unambiguously parsed using a lookahead of one symbol.

Similarly to forest grammars and as opposed to DTDs, XML Schema [82] and RelaxNG [65] allow both to specify context-dependent and non-deterministic content models. In contrast to forest grammars, they also allow the specification of the datatype of the text nodes more precisely, for example whether it should represent an integer, a float or a date. This goes beyond the basic capabilities of forest grammars. However, it is possible to define this kind of requirements using forest grammars with external predicates, by using a new non-terminal and a corresponding external predicate for each basic type needed, which tests whether the text can be converted into a value of the corresponding required type.

Another feature provided by schema languages (DTDs and XML Schema, not RelaxNG) is specifying uniqueness and reference constraints. Uniqueness constraints are used to ensure that there are no two elements with the same property, e.g. with an identical value of an attribute with a given name. Reference constraints are meant to ensure that a property of an element identifies an existing property of another element, e.g. that the value of an attribute of an element identifies another element which contains the same value in another attribute. This kinds of constraints cannot be expressed using forest grammars. While this is an important feature, it does not actually belong to the structural constraints and can be handled in applications after checking conformance to the schema.

Other features such as the ability of XML Schema to directly specify a minimum or maximum number of times a certain element type should occur do not add to the theoretical expressivenes but are very convenient in practice.

5.3.2 *Recognizing Regular Forest Languages*

In this section we briefly review how the structural constraints specified via forest grammars can be efficiently checked.

Neumann showed that the expressive power of forest grammars is equal with that of regular tree grammars [53]. That is, for every forest grammar G there is exactly one regular tree grammar G' such that the ranked tree language specified by G' is exactly the image of the forest language specified by G through an encoding function which maps every forest to a ranked tree. One such encoding can be obtained by representing the arbitrarily long sequences of sibling nodes in a similar way to how lists are represented in functional programming languages, via a binary constructor *cons* and a nullary constructor *nil*.

Therefore, testing the membership of a forest in a regular forest language (specified by a forest grammar G) is equivalent to testing the membership of its ranked encoding in the corresponding regular (ranked) tree language (specified by the regular tree grammar G'). It is well known that regular tree languages are recognized by bottom-up tree automata [28]. Hence, recognizing regular forest languages could be in principle solved using the classic bottom-up tree automata. In fact, most of the research literature handling XML processing use this to ignore the unrankedness of XML trees.

Nevertheless, this has a few drawbacks. Firstly, in a practical setting it requires a supplementary overhead for the encoding step. Secondly, some natural one-to-one correspondences between the XML data model and the tree representation, as for example tree relationships, are not directly recognizable in the encoded tree. In contrast, constructions using the original unranked representations are more straightforward and easy to realize in practice. Therefore, we prefer to use a unranked variant of tree automata. One straightforward approach to recognizing regular forest languages is to use bottom-up forest automata [53, 55] (or, equivalently, unranked tree automata [13]). However, their implementation may be very expensive [53,55].

As expressive as bottom-up automata but much more concise and efficient to implement in practice are the *pushdown forest automata* [53, 55]. Any implementation of a bottom-up automaton has to choose a traversal strategy for the input tree. The idea of a pushdown forest automaton (PA) is based on the observation that, when reaching a node during the traversal, the information gained from the already visited part of the tree can be used at the transitions of the automaton at that node. This supplementary information allows a significant reduction in the size of the states and in the number of possible transitions to be considered by a deterministic PA as compared to the equivalent deterministic bottom-up automaton. Intuitively, in the case of a depth-first, left-to-right traversal, the advantage is that information gained by visiting the left siblings as well as the ancestors and their left siblings can be taken into account before processing the current node. The name of the automata (pushdown forest automata) is due to the fact that information from the visited part of the tree is stored on the stack (pushdown) which is implicitly used for the tree traversal.

Another advantage of PAs over bottom-up automata is that they can visit the elements in an XML input exactly in the order in which these are read from the input. Consequently, they do not need to materialize the tree representation of the input in memory, as they can handle the XML elements as they come, in an event-driven manner. This makes PAs suitable for applications in which the tree cannot be built in main memory, as for example in the case of very large XML documents. We take up again this topic in more detail in Section 5.6.

5.3.2.1 *Pushdown Forest Automata*

In addition to the tree states of classic tree automata, a PA also has *forest states*. Intuitively, a forest state contains the information gained from the already visited part of the tree (*context information*) at any point during the tree traversal. Let us consider a depth-first, left-to-right traversal. The following notations are essentially those introduced in [53].

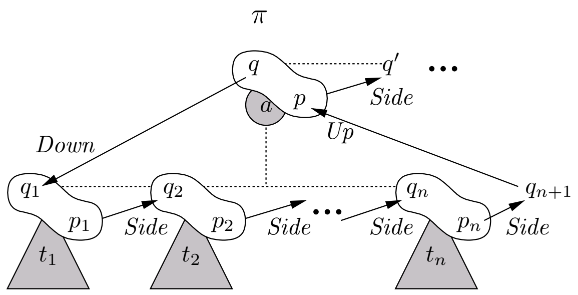


Fig. 5.6 The processing model of a pushdown forest automaton.

The behavior of a *left-to-right pushdown forest automaton* (LPA) is depicted in Figure 5.6, the notations of which are used in the following explanation. When arriving at some node π labeled a, the context information is available in the forest state q by which the automaton reaches the node. The automaton has to traverse the content of π and compute a tree state p, which describes π within the context q. In order to do so, the children of π are recursively processed. The context information for the first child, q_1, is obtained (via a *Down* transition) by refining q by taking into account that the father is labeled a. Subsequently the q_2 context information for the second child is obtained (via a *Side* transition) from q_1 and the information p_1 gained from the traversal of t_1. Proceeding in this manner, after visiting all children of π, enough context-information is collected in q_{n+1} in order to compute p (via an *Up* transition). After processing π, the context information for the subsequent node is updated into q'.

Formally, an LPA $A = (P, Q, I, F, \mathit{Down}, \mathit{Up}, \mathit{Side})$ over an alphabet Σ consists of a finite set of *tree states* P, a finite set of *forest states* Q, a set of *initial states* $I \subseteq Q$, a set of *final states* $F \subseteq Q$, a *down-relation* $Down \subseteq Q \times \Sigma \times Q$, an *up-relation* $Up \subseteq Q \times \Sigma \times P$ and a *side-relation* $Side \subseteq Q \times P \times Q$. Based on *Down*, *Up* and *Side*, the behavior of A is described by the relations $\delta^A_{\mathcal{F}} \subseteq Q \times \mathcal{F}_\Sigma \times Q$ and $\delta^A_{\mathcal{T}} \subseteq Q \times \mathcal{T}_\Sigma \times P$ as follows, where the notations correspond to those in Figure 5.6:

(1) $(q, a\langle t_1 \ldots t_n\rangle, p) \in \delta^A_{\mathcal{T}}$ iff $(q, a, q_1) \in Down$, $(q_1, t_1 \ldots t_n, q_{n+1}) \in \delta^A_{\mathcal{F}}$ and $(q_{n+1}, a, p) \in Up$ for some $q_1, q_{n+1} \in Q$.
(2) $(q_1, t_1 f, q_{n+1}) \in \delta^A_{\mathcal{F}}$ iff $(q_1, t_1, p_1) \in \delta^A_{\mathcal{T}}$, $(q_1, p_1, q_2) \in Side$ and $(q_2, f, q_{n+1}) \in \delta^A_{\mathcal{F}}$ for some $p_1 \in P, q_2 \in Q$
(3) $(q_1, \varepsilon, q_1) \in \delta^A_{\mathcal{F}}$ for all $q_1 \in Q$

The language accepted by the automaton A is given by:

$$\mathcal{L}_A = \{f \in \mathcal{F}_\Sigma \mid q_1 \in I, q_2 \in F \text{ and } (q_1, f, q_2) \in \delta^A_{\mathcal{F}}\}$$

An LPA performs a depth-first, left-to-right traversal of the input. Similarly, if we consider a depth-first, right-to-left traversal we obtain a *right-to-left pushdown forest automaton* (RPA). An RPA $A = (P, Q, I, F, Down, Up, Side)$ is similar to an LPA but, as it proceeds on a forest from the right to the left, case (2) from above is replaced by:

(2') $(q_{n+1}, t_1 f, q_1) \in \delta^A_{\mathcal{F}}$ iff $(q_{n+1}, f, q_2) \in \delta^A_{\mathcal{F}}$, $(q_2, t_1, p_1) \in \delta^A_{\mathcal{T}}$ and $(q_2, p_1, q_1) \in Side$ for some $q_2 \in Q, p_1 \in P$.

If the *Down*, *Up* and *Side* transitions of a PA are functions rather than relations and there is exactly one start state, the PA is called *deterministic*. Otherwise, it is called *non-deterministic*. If a PA is deterministic we write $Down(q, a) = q_1$, $Side(q_1, p_1) = q_2$ and $Up(q_{n+1}, a) = p$ rather than $(q, a, q_1) \in Down$, $(q_1, p_1, q_2) \in Side$ and $(q_{n+1}, a, p) \in Up$, respectively.

5.3.2.2 *From Forest Grammars to Pushdown Forest Automata*

A compilation schema from a forest grammar $G = (R, r_0)$ into a deterministic LPA (DLPA) accepting the same regular forest language that we briefly review here has been given in [53]. The idea is that the DLPA keeps at any time track of all possible content models of the elements whose content has not yet been seen in its entirety. The forest is accepted at the end if the sequence of top-level nodes conforms to r_0.

Let $r_1, \ldots, r_l$ be the regular expressions occurring on the right-hand sides in the productions R, where l is the number of productions. For $0 \leq j \leq l$, let $A_j = (Y_j, y_{0,j}, F_j, \delta_j)$ be the non-deterministic finite automaton

(NFA) accepting the regular string language defined by r_j, as obtained by the Berry-Sethi construction (presented in Section 5.2.1). Recall that Y_j is the set of NFA states, $y_{0,j}$ the start state, F_j the set of final states and $\delta_j \in Y_j \times \Sigma \times Y_j$ is the transition relation.

By possibly renaming the NFA states we can always ensure that $Y_i \cap Y_j = \varnothing$ for $i \neq j$. Let $Y = Y_0 \cup \cdots \cup Y_l$ and $\delta = \delta_0 \cup \cdots \cup \delta_l$. A DLPA $A_{\vec{G}}$ accepting $\mathcal{L}_G$ can be defined as $A_{\vec{G}} =(2^X, 2^Y, \{q_0\}, F, \mathit{Down}, \mathit{Up}, \mathit{Side})$, where X is the set of non-terminals in G. A tree state synthesized for a node is the set of non-terminals from which the node can be derived. A forest state consists of the NFA states reached within the possible content models of the current level and can be computed as follows.

We start with the content model r_0, i.e.:

$$q_0 = \{y_{0,0}\}$$

We accept the top level sequence of nodes if it conforms to r_0, i.e.:

$$F = \{q \mid q \cap F_0 \neq \varnothing\}$$

The possible content models of a node are computed from the content models in which the node may occur:

$$\mathit{Down}(q, a) = \{y_{0,j} \mid y \in q,\ (y, x, y_1) \in \delta,\ x \to a\langle r_j\rangle\}$$

When finishing a sequence of siblings we consider only the fulfilled content models in order to obtain the non-terminals from which the father node may be derived:

$$\mathit{Up}(q, a) = \{x \mid x \to a\langle r_j\rangle \text{ and } q \cap F_j \neq \varnothing\}$$

The possible content models are updated after finishing visiting the next node in a sequence of siblings:

$$\mathit{Side}(q, p) = \{y_1 \mid y \in q, x \in p \text{ and } (y, x, y_1) \in \delta\}$$

The resulting $A_{\vec{G}}$ is obviously deterministic, since it has one initial state and its transitions are functions rather than relations.

Example 5.9. The NFAs for the regular expressions occurring in grammar G with the set of rules specified in Example 5.4 (on page 219) are depicted in Figure 5.7. Consider as input the XML document depicted in Figure 5.4 (on page 220). The run of $A_{\vec{G}}$ on the tree representation of the input is shown in Figure 5.8, where the sets containing x-s are tree states and the sets containing y-s are forest states. The order in which the tree and forest states are computed is denoted by the subscripts at their right. Observe that the input tree, which is in the regular forest language specified by G, is accepted by $A_{\vec{G}}$ as it stops in the state $\{y_1\}$, which is a final state of the LPA.

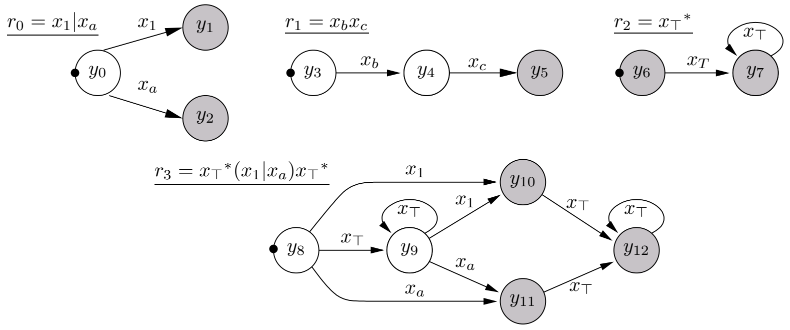


Fig. 5.7 NFAs obtained by Berry-Sethi construction for regular expressions in Example 5.6.

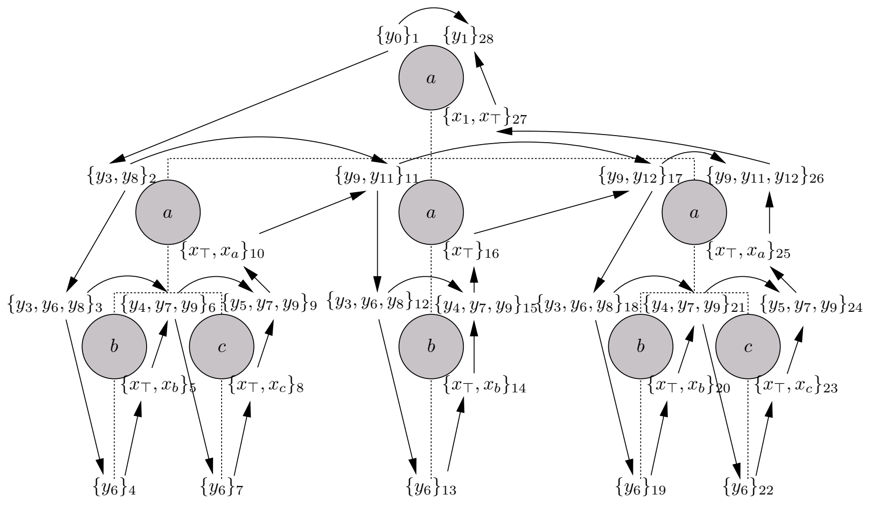


Fig. 5.8 The run of $A_{\vec{G}}$ on an input tree.

5.3.3 *Bibliographic Notes*

Originally, Neumann and Seidl have used *μ-formulas* [56] and later *constraint systems* [55] to specify regular forest languages. Forest grammars, as a more comprehensible mean of specifying regular forest languages, have been introduced in [53], as an adaption of tree grammars from the ranked to the unranked tree case. An overview on how results from the ranked tree-theory carry over in general to the unranked case is presented by Brüggemann-Klein *et al.* in [13].

The correlation between the most popular available schema languages and regular forest languages has been studied by Murata *et al.* [51]. For each considered schema language a corresponding restriction of regular forest languages is identified. An algorithm for checking the conformance to a schema is presented for each such subclass.

Checking conformance to a schema which is a regular forest language can be very efficiently performed by a pushdown forest automaton as presented in this chapter. Pushdown forest automata have been introduced by Neumann and Seidl in [55]. They show that every non-deterministic pushdown forest automaton can be made deterministic and that they are much more concise when compared to bottom-up automata. They also give a compilation schema from constraint systems to deterministic pushdown forest automata. The compilation of forest grammars to deterministic pushdown forest automata was introduced in [53].

5.4 Grammar Queries

In this section we present how forest grammars can be used to specify powerful XML queries. Forest grammars queries are suitable for the implementation of XML pattern languages as it will be presented in Section 5.5. Most of the attention in the study of XML query languages has been drawn by *unary queries*, which locate individual nodes from the input tree. In contrast, in Section 5.4.1, we present a formalism which can express *k-ary queries*, which are able to locate k nodes which simultaneously satisfy a specific property, and discuss their expressiveness in Section 5.4.2. In Section 5.4.3 we review an efficient construction based on pushdown automata which can be used to find matches of unary grammar queries. An efficient construction for locating binary queries is presented in Section 5.4.4. Finally, implementing k-ary queries in general is addressed in Section 5.4.5.

5.4.1 *Specifying Queries*

One possible way of identifying nodes of interest, as required by the task of querying, is to label them with special symbols. In this respect, derivations according to a forest grammar, which, as seen in the previous chapter, are relabelings of input forests, can be used as a means of specifying queries. The definitions of grammar queries, given in the remainder of this section pursue this observation.

5.4.1.1 *Unary Queries*

As previously suggested, given a grammar G, a non-terminal x of it specifies a query by identifying all nodes π in the input f for which there is a derivation f' w.r.t. G in which π is labeled with x. More generally, a *unary grammar query* Q is a pair (G,T) consisting of a forest grammar $G = (R, r_0)$ and a set of *target non-terminals* $T \subseteq X$ where X is the set of non-terminals in R. The *matches* of Q in an input forest f are given by the set $\mathcal{M}_{Q,f} \subseteq N(f)$ as follows:

$$\pi \in \mathcal{M}_{Q,f} \text{ iff } \exists (f,f') \in \mathcal{D}eriv_G, \exists x \in T \text{ and } lab(f'[\pi]) = x$$

We say that π is a match of Q in f w.r.t. the derivation f'.

Example 5.10. Consider the grammar G from Example 5.6 (on page 221). The query $Q_1 = (G, \{x_b\})$ locates nodes b having only a ancestors and only one sibling c to the right. The leftmost b in the input tree depicted in Figure 5.4 (on page 220) is a match, as one can see by definition by looking at the first derivation in Figure 5.5 (on page 220). Similarly, the rightmost b is a match as defined by the second derivation w.r.t. G.

The query $Q_2 = (G, \{x_a\})$ locates the a nodes which have a child b followed by a child c. These are the leftmost and the rightmost a nodes.

In general, as suggested in the example above, a single grammar can be flexibly used to specify many similar yet different queries, one for each non-terminal. This flexibility is in contrast with pattern languages, where for each query a significantly different pattern has to be specified.

Note that we decided for an *all-matches* semantic of our queries, i.e. all nodes π as in the definition are to be reported as matches. This is reasonable, because a user query typically is aimed at finding *all* locations with the specified properties, as for instance in XPath . Furthermore, we do not want to place on the user the burden of specifying the query via an unambiguous grammar, therefore the definition above refers to *any* derivation.

5.4.1.2 *K-ary Queries*

The definition of queries given in the previous section can be straightforwardly extended in order to identify k-tuples of nodes related via structural constraints, as imposed by forest grammars.

A *k-ary grammar query* is a pair $Q = (G, T)$ consisting of a forest grammar $G = (R, r_0)$ and a k-ary relation $T \subseteq X^k$ where X is the set non-terminals in R. The *matches* of Q in an input forest f are given by the k-ary relation $\mathcal{M}_{Q,f} \subseteq N(f)^k$:

$$(\pi_1, \ldots, \pi_k) \in \mathcal{M}_{Q,f} \text{ iff } \exists (f, f') \in \mathcal{D}eriv_G, \exists (x_1, \ldots, x_k) \in T \text{ and}$$
$$lab(f'[\pi_i]) = x_i \text{ for } i = 1, \ldots, k$$

We say that $(\pi_1, \ldots, \pi_k)$ is a match of Q in f w.r.t. the derivation f'. For $k = 1$ and $k = 2$ we obtain unary and binary queries, respectively.

Example 5.11. Consider the grammar G from Example 5.6 (on page 221). The binary query $Q_2 = (G, \{(x_b, x_c)\}$ locates pairs of nodes b and c having as father the same node a, and only a ancestors. The leftmost b and c in the input depicted in Figure 5.4 (on page 220) form a match pair, as one can see by definition by looking at the first derivation in Figure 5.5. Similarly, the rightmost b and c form a match pair as defined by the second derivation w.r.t. G.

5.4.2 *Expressive Power of Grammar Queries*

As noted in Section 5.3.2, forest grammars are as expressive as regular tree grammars. The proof by Neumann [53] shows that for every forest grammar G there is exactly one regular tree grammar G' s.t. the language specified by G' is the image of the language specified by G through a bijective function *enc* mapping every unranked tree (or forest) to a unique ranked tree representation.

In particular, *enc* can be chosen s.t. an arbitrarily long sequence of sibling nodes is represented similarly to the way that lists are represented in functional programming languages via two constructor nodes *cons* and *nil*, with arity 2 and 0 respectively. This mapping ensures that every node in a forest f corresponds to exactly one node in $enc(f)$. Moreover, the construction presented in [53] ensures that for every non-terminal x in G there is exactly one non-terminal x' in G' such that a (forest) derivation of some input forest f labeling a node with x exists iff a (tree) derivation of the ranked encoding $enc(f)$ exists labeling the corresponding node in the encoding with x'.

According to the definition of k-ary grammar queries, this implies that a tuple $(\pi_1, \pi_2, \ldots, \pi_k)$ of nodes $\pi_i \in N(f)$ is a match of a query $(G, (x_1, x_2, \ldots, x_k))$ iff the corresponding tuple of nodes $(\pi'_1, \pi'_2, \ldots, \pi'_k)$ of nodes $\pi'_i \in N(enc(f))$ is a match of a query $(G', (x'_1, x'_2, \ldots, x'_k))$. Therefore, the expressive power of forest grammar queries is equal to that of regular tree grammar queries.

It is well known that the class of languages specified by regular tree grammars (the regular ranked tree languages) is exactly the same as the class of languages specified by formulas of monadic second order logic (MSO) on trees without free variables [77]. Using this result, and casting the problem of finding matches of regular tree grammar queries into a language recognition problem, one can show that the expressive power of k-ary tree grammar queries is equal with that of MSO formulas with k free variables. A proof can be consulted in [64]. We conclude that the expressive power of our k-ary grammar queries is equal to that of MSO formulas with k free variables.

Queries specified directly via MSO formulas are not practicable due to their high evaluation complexity, yet they have been used as convenient benchmarks for comparing XML query languages [61] due to their large expressive power. Indeed MSO queries subsume many of the fundamental features of the query languages which have been proposed for XML (as it will be presented in Section 5.4.7). Grammar queries have thus the same expressive power as MSO queries, while being efficiently implementable, at least in the unary and binary case, as we show in the next sections.

5.4.3 *Recognizing Unary Queries*

A construction for answering unary grammar queries using pushdown forest automata has been presented in [53, 55]. In the present section we briefly review this construction. Knowing this construction helps understanding its generalization for binary and k-ary queries which is presented in Section 5.4.4 and Section 5.4.5, respectively.

Specifying which are the subtrees of interest in a query typically consists of two conceptual parts, as described in Figure 5.9. The contextual part constrains the surrounding context of the subtrees of interest, whereas the structural part describes the properties of the subtrees themselves.

Example 5.12. Supposing we have an XML document which represents a conference article, where sections and subsections are encoded as XML elements, we might be interested in *subsections containing the word "automata"* occurring *in sections whose title contain the word "forest"*. The

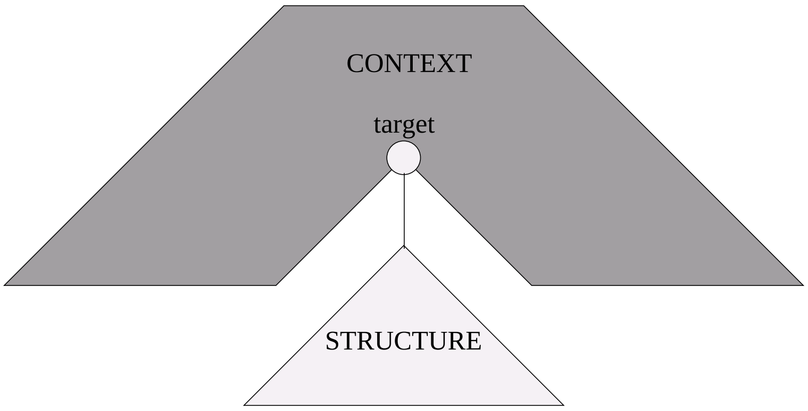


Fig. 5.9 The contextual and the structural part of a query.

two emphasized parts denote the structural and the contextual part of the query, respectively.

Example 5.13. As seen in Example 5.10 (on page 231), the query $Q_1 = (G, \{x_b\})$ locates the b nodes (structure) which have only a ancestors and a right c sibling (context).

When specifying a query as a grammar $G = (R, r_0)$ together with a distinguished non-terminal x one specifies at once the desired structure and context of some subtree t in a forest f. The structure is described by the productions which can be used in order to derive a tree t starting from x. The remaining productions of the grammar, which constrain the locations where x can occur in a derivation of f from r_0, capture the context part of the specification.

As argued in Section 5.3.2.1 a PA uses its forest states to remember information from the already visited part of the input. Therefore, by looking into the forest state of the PA after visiting a subtree t it should be possible to check a structural property of t as well as whether a contextual property can be satisfied considering the part of the context seen so far.

Example 5.14. Let Q_1 be the unary query from Example 5.10 (on page 231), identifying b nodes which have only a ancestors and only one c sibling to the right. Consider the run of the corresponding LPA on the input as depicted in Figure 5.8 (on page 229). One can see that by the time the automata has seen any of the b nodes, each of them fulfills the structural part (it is a b node) and the upper-left contextual part (all ancestors are a

nodes). This is reflected in the forest states of the LPA when it leaves each of the b nodes, depicted at the upper right of each of them, respectively. In each of these forest states, the NFA state y_4, which is reached after reading an x_b, denotes that a derivation of the input forest may exist in which the respective node is labeled x_b.

However, since the right part of the context has not yet been seen, the LPA cannot decide at the time it leaves the b nodes whether they are indeed matches.

In order to decide whether a node is a match, in general, the remaining part of the context also has to be seen. The idea is to *remember* for each node the information collected after seeing only a part of the context and to let a second automaton proceed from the opposite direction (i.e. to perform a depth-first, right-to-left traversal if the first PA does a left-to-right traversal) in order to account for the remaining context.

Before proceeding in Section 5.4.3.2 to the construction based on the two PA runs for the evaluation of a grammar queries, we introduce in Section 5.4.3.1 a couple of useful notations which allow us to speak about the states of the PAs at a certain location.

5.4.3.1 *Pushdown Forest Automata as Relabelings*

A run of a deterministic PA on an input forest f can be seen as a *relabeling* of each node in f with the triple of states involved in the transitions at that node during the run[2].

Consider a DLPA A as defined in Section 5.3.2.1 (on page 226). The relabeling of f performed by A is a mapping $\overrightarrow{\lambda} : N(f) \to Q \times P \times Q$, $\overrightarrow{\lambda}(\pi i) = (\overrightarrow{q}_{\pi(i-1)}, \overrightarrow{p}_{\pi i}, \overrightarrow{q}_{\pi i})$, where, for the node πi, $\overrightarrow{q}_{\pi(i-1)}$, $\overrightarrow{p}_{\pi i}$ and $\overrightarrow{q}_{\pi i}$ are the forest state in which the node is reached, the tree state synthesized for the node and the forest state in which the node is left respectively, by A, i.e.:

$$\begin{aligned}
\overrightarrow{q}_{\lambda 0} &= \overrightarrow{q}_0 \text{ (the initial state)}\\
\overrightarrow{q}_{\pi 0} &= \mathit{Down}(\overrightarrow{q}_\pi, a)\\
\overrightarrow{p}_\pi &= \mathit{Up}(\overrightarrow{q}_{\pi n}, a), \text{ if } n = \mathit{last}_f(\pi)\\
\overrightarrow{q}_{\pi i} &= \mathit{Side}(\overrightarrow{q}_{\pi(i-1)}, \overrightarrow{p}_{\pi i})
\end{aligned}$$

where $a = \mathit{lab}(f[\pi])$.

[2]For a visualization, observe Figure 5.6 on page 226 where for the node denoted π, the above mentioned states correspond to q, p and q'.

Similarly, a deterministic RPA (DRPA) B can be seen as a relabeling $\overleftarrow{\lambda}(\pi i) = (q_{\pi(i-1)}, p_{\pi i}, q_{\pi i})$, where $q_{\pi i}$, $p_{\pi i}$ and $q_{\pi(i-1)}$ are the forest state in which the node is reached, the tree state synthesized for the node and the forest state in which the node is left, respectively.

5.4.3.2 *Locating Unary Matches*

The state in which a DLPA leaves a node π synthesizes all the information collected after seeing the upper left context and all the content of π. Given this information, a second (DRPA) automaton, proceeding from right to left, will have at every node the information necessary in order to decide whether the node fulfills the structural and contextual requirements of a query.

Consider a unary query (G, T). Let $A_{\overrightarrow{G}}$ be the DLPA accepting the language of grammar G, constructed as in Section 5.3.2.2 (on page 227). We now present how to construct the second DRPA $B_{\overleftarrow{G}}$ for the given grammar G. In the following we use notations as introduced in Section 5.4.3.1. That is, given a node π, we denote by $\overrightarrow{p}_\pi$ and $\overrightarrow{q}_\pi$ the tree state synthesized for π and the forest state in which π is left by $A_{\overrightarrow{G}}$, respectively. For $B_{\overleftarrow{G}}$, we denote by q_π and p_π, the forest state in which π is reached and the tree state synthesized for π by the DRPA, respectively.

By remembering $\overrightarrow{q}_\pi$ one can locally decide at each node during a second traversal of the input by $B_{\overleftarrow{G}}$ whether the node is a match of a query. Also, to avoid unnecessary re-computations by $B_{\overleftarrow{G}}$, $\overrightarrow{p}_\pi$ is remembered so as to account for the structure information collected at π.

The automaton $B_{\overleftarrow{G}}$ runs thus on an *annotation* $\overrightarrow{f}$ of the input forest f by $A_{\overrightarrow{G}}$, $\overrightarrow{f} \in \mathcal{F}_{\Sigma \times P \times Q}$ with $N(\overrightarrow{f}) = N(f)$ and $lab(\overrightarrow{f}[\pi]) = (lab(f[\pi]), \overrightarrow{p}_\pi, \overrightarrow{q}_\pi)$ for all $\pi \in N(f)$.

The construction of $B_{\overleftarrow{G}}$ is similar to that of $A_{\overrightarrow{G}}$ but follows the NFA transitions in reverse and considers corresponding NFA final states at rightmost siblings, as the input to the NFAs is seen from the right to the left. Additionally, $B_{\overleftarrow{G}}$ takes into account information collected by $A_{\overrightarrow{G}}$ in order to avoid considering NFA transitions which were not relevant for the conformance check performed by $A_{\overrightarrow{G}}$. The automaton $B_{\overleftarrow{G}} = (2^X, 2^Y, \{F_0\}, \varnothing, Down^{\leftarrow}, Up^{\leftarrow}, Side^{\leftarrow})$, where X, Y and F_0 are as in the definition of $A_G^{\rightarrow}$, is given by:

$$\begin{aligned}
\mathit{Down}^{\leftarrow}(q,(a,\overrightarrow{p},\overrightarrow{q})) &= \{y_2 \mid \ y \in q \cap \overrightarrow{q},\ (y_1,x,y) \in \delta, \\
&\qquad\qquad x \to a\langle r_j\rangle \text{ and } y_2 \in F_j\} \\
\mathit{Up}^{\leftarrow}(q,(a,\overrightarrow{p},\overrightarrow{q})) &= \overrightarrow{p} \\
\mathit{Side}^{\leftarrow}(q,p,(a,\overrightarrow{p},\overrightarrow{q})) &= \{y \mid (y,x,y_1) \in \delta, y_1 \in q \cap \overrightarrow{q}, x \in p\}
\end{aligned}$$

where we also provide the $\mathit{Side}^{\leftarrow}$ transition with the label $(a,\overrightarrow{p},\overrightarrow{q})$ of the node over which it is executed.

Note that $p_\pi = \overrightarrow{p}_\pi$ for all π. When it is clear from the context which is the label $(a,\overrightarrow{p},\overrightarrow{q})$ at a transition we will omit this argument.

The following proposition by Neumann [53] shows how for every node π, the forest state q_π in which $B_G^{\leftarrow}$ arrives at π, containing information from the right context can be combined with the information for the remaining part of the input given in the annotation $\overrightarrow{q}_\pi$ in order to find matches of a unary query. A node is a match if both the forest states in which $A_G^{\rightarrow}$ leaves the node and in which $B_G^{\leftarrow}$ arrives at the node contain an NFA state reachable after seeing a target non-terminal from T.

Theorem 5.15. *Let $Q = (G,T)$ be a unary query and $f \in \mathcal{L}_G$. With $A_G^{\rightarrow}$ and $B_G^{\leftarrow}$ as above, $\pi \in \mathcal{M}_{Q,f}$ iff $y_1 \in q_\pi \cap \overrightarrow{q}_\pi$ and $(y,x,y_1) \in \delta$ for some $y, y_1 \in Y$ and $x \in T$.*

Proof. This theorem is proven in [53] as Theorem 7.1. □

Directly from Theorem 5.15 follows the corollary:

Corollary 5.16. *$(f,f') \in \mathcal{D}eriv_G$ and $lab(f'[\pi]) = x$ iff $y \in q_\pi \cap \overrightarrow{q}_\pi$, $(y_1,x,y) \in \delta$ for some $y, y_1 \in Y$.*

This further implies that:

Corollary 5.17. *If $(f,f') \in \mathcal{D}eriv_G$ and $lab(f'[\pi]) = x$, then $x \in p_\pi$.*

Proof. By Corollary 5.16 there are $y \in q_\pi \cap \overrightarrow{q}_\pi$, $(y_1,x,y) \in \delta$. Since $y \in \overrightarrow{q}_\pi$, it follows by the definition of *Side* in $A_G^{\rightarrow}$ that there is $(y',x_1,y) \in \delta$ for some $x_1 \in p_\pi$. By the Berry-Sethi construction, all incoming transitions into an NFA state y are labeled with the same symbol. Therefore, $x_1 = x$ and thus $x \in p_\pi$. □

Example 5.18. Consider the run of $A_G^{\rightarrow}$ depicted in Figure 5.8 (on page 229). The run of $B_G^{\leftarrow}$ on the tree annotated by $A_G^{\rightarrow}$ is presented in Figure 5.10. The order in which the tree and forest states are computed is

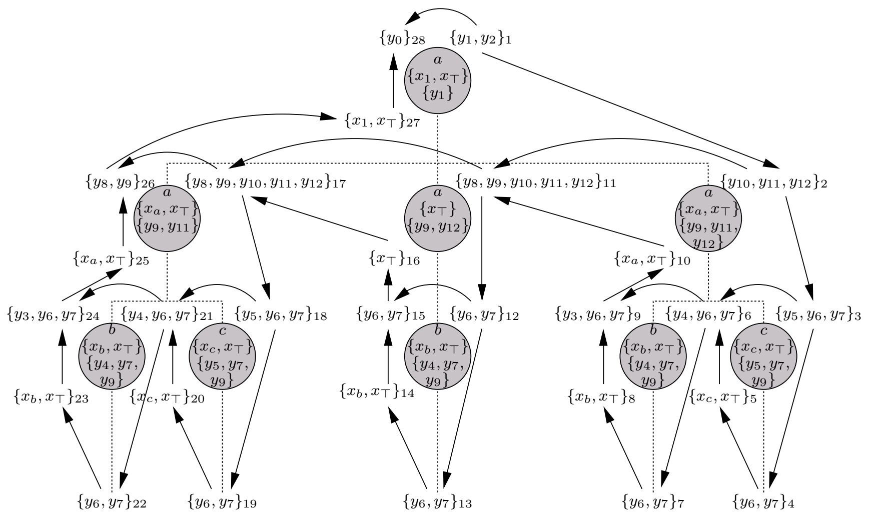


Fig. 5.10 The run of the $B_G^{\leftarrow}$ on the input document annotated by the $A_G^{\rightarrow}$ in Example 5.9.

denoted by the subscripts at their right. Note how the rightmost b node is recognized as a match of the query $Q_1 = (G, \{x_b\})$. As noted in Example 5.14, the NFA state y_4 (having an incoming transition labeled x_b) in the annotation done by $A_G^{\rightarrow}$ denotes the node as a potential match after accounting for its upper left context and its content. The conformance of the right context is also fulfilled as the forest state in which $B_G^{\leftarrow}$ arrives at the node contains y_4 as well. Similarly, the leftmost b node is a match. On the contrary, the node b in the middle is not a match, as its right context does not contain a c sibling as required by the query.

Complexity

Let n be the size of the input forest f, i.e. the number of nodes in f. The complexity of answering a binary query is given by the complexities of running $A_G^{\rightarrow}$ and $B_G^{\leftarrow}$ and that of detecting the matches as stated by Theorem 5.15.

The automaton $A_G^{\rightarrow}$ executes at each node one *Down*, one *Side* and one *Up* transition. As one can see in the definitions of the transitions, the time cost of each of these transitions does not depend on f. The run of $A_G^{\rightarrow}$ requires thus time $\mathcal{O}(n)$. Similarly, the run of $B_G^{\leftarrow}$ needs time $\mathcal{O}(n)$.

As detecting each match as by Theorem 5.15 is not dependent on n, this leads to the overall $\mathcal{O}(n)$ complexity.

We have thus proven the following theorem.

Theorem 5.19. *The complexity of answering unary gramar queries is linear in the size of the input document.*

5.4.4 *Recognizing Binary Queries*

In this section we present a construction that allows the efficient evaluation of binary queries. The construction is based on the technique introduced in the previous section for the evaluation of unary queries. As opposed to unary queries, where the decision whether a node is a match can be taken when the node is visited by the second automaton, finding a match pair of a binary query requires postponing the decision at least until both nodes in the pair have been visited. We thus need a supplementary construction which allows the remembering of information distributed upon the tree, and the use of this information to detect matches.

We introduce the necessary construction in Section 5.4.4.1 and show how it can be used to efficiently evaluate a slightly restricted class of binary queries. In Section 5.4.4.2 we show how the approach works for general binary queries.

5.4.4.1 *Recognizing Simple Binary Queries*

Let $Q = (G, B)$ be a binary query. For convenience, we will first assume that $B = \{(x^1, x^2)\}$ for some $x^1, x^2 \in X$, where X is the set of non-terminals from G. We call such a query a *simple binary query*.

According to the definition, a pair (π_1, π_2) is a match for an input f iff there is a derivation f' of f w.r.t. G and $f'[\pi_1] = x^1$, $f'[\pi_2] = x^2$.

Observe that this implies that π_1 and π_2 are matches of the unary queries (G, x^1) and (G, x^2), respectively. Thereby, (π_1, π_2) is a binary match for Q iff:

(p) π_1 is a match of the unary query (G, x^1) and
(s) π_2 is a match of the unary query (G, x^2) and
(r) π_1 and π_2 are unary matches w.r.t. the same derivation f'.

We call the nodes fulfilling (p) and (s) *primary* and *secondary* matches, or, for short, *primaries* and *secondaries*, respectively.

We have already seen how unary matches can be located. Thus, testing (p) and (s) can be done by an automata construction as in Section 5.4.3. In order to implement binary queries, however, one must additionally be able to test (r).

Construction

In the following we show that binary queries can be efficiently answered by using a run of a DLPA $A_G^{\rightarrow}$ followed by a run of a DRPA $B_G^{\leftarrow}$, in a way which is similar to the case of unary queries. The $A_G^{\rightarrow}$ and $B_G^{\leftarrow}$ automata are defined exactly as in Section 5.4.3.2. Primary and secondary matches can be thus recognized in the same way as in Section 5.4.3.2 and we keep the same notations as there.

In order to locate binary matches, we have to remember during the run of $B_G^{\leftarrow}$ which of the already visited nodes are primary or secondary matches, as potential components of binary matches. We accumulate these primaries and secondaries in set attributes l_1 and l_2, respectively, with which we equip each element of the tree and forest states of $B_G^{\leftarrow}$.

For a tree state p at node π and $x \in p$, $x.l_1$ contains primary matches and $x.l_2$ secondary matches which are found below π and are defined w.r.t. derivations which label $f[\pi]$ with x.

Similarly, for a forest state q at node π and $y \in q$, $y.l_1$ contains primary and $y.l_2$ secondary matches collected from the already visited right-sibling subtrees of $f[\pi]$. These are the matches defined w.r.t. derivations in which the word of non-terminals on the current level is accepted by an NFA reaching the current location in state y.

Similarly to attribute grammars, the values of the l_1 and l_2 attributes are defined by a set of local rules, as follows:

- For the elements of a forest state in which $B_G^{\leftarrow}$ arrives at a node π which has no right-siblings (i.e. π is the rightmost node among its siblings), the sets of primaries and secondaries collected from the right sibling subtrees are obviously empty. This is the case for the initial state F_0 at the root and for the states obtained by executing a $Down^{\leftarrow}$ transition:

$$\text{If } y \in F_0 \text{ or } y \in Down^{\leftarrow}(q, (a, \overrightarrow{p}, \overrightarrow{q})), \text{ then } y.l_1 = \varnothing,\ y.l_2 = \varnothing$$

- After finishing visiting the children of a node π, the sets of primaries and secondaries found below π are propagated and possibly updated with π if π is a primary or secondary match, respectively:

$$\text{If } x \in \mathit{Up}^{\leftarrow}(q, (a, \overrightarrow{p}, \overrightarrow{q})), \text{ then}$$

$$x.l_1 = \begin{cases} \{\pi\} \cup \bigcup\{y.l_1 \mid y \in q, y = y_{0,j}, x \to a\langle r_j\rangle\}, & \text{if } x = x^1 \\ \bigcup\{y.l_1 \mid y \in q, y = y_{0,j}, x \to a\langle r_j\rangle\} & \text{, otherwise} \end{cases}$$

$$x.l_2 = \begin{cases} \{\pi\} \cup \bigcup\{y.l_2 \mid y \in q, y = y_{0,j}, x \to a\langle r_j\rangle\}, & \text{if } x = x^2 \\ \bigcup\{y.l_2 \mid y \in q, y = y_{0,j}, x \to a\langle r_j\rangle\} & \text{, otherwise} \end{cases}$$

- At side transitions over a node π labeled with $(a, \overrightarrow{p}, \overrightarrow{q})$, the list of primaries and secondaries found so far are obtained by combining the matches below π with the matches from the already visited part to the right:

$$\text{If } y \in \mathit{Side}^{\leftarrow}(q, p, (a, \overrightarrow{p}, \overrightarrow{q})), \text{ then}$$

$$y.l_1 = \bigcup\{y_1.l_1 \cup x.l_1 \mid (y, x, y_1) \in \delta, y_1 \in q \cap \overrightarrow{q}, x \in p\}$$
$$y.l_2 = \bigcup\{y_1.l_2 \cup x.l_2 \mid (y, x, y_1) \in \delta, y_1 \in q \cap \overrightarrow{q}, x \in p\}$$

Note that the rules allow a bottom-up, right-to-left evaluation of the attributes. Therefore, they can be evaluated directly along the run of $B_G^{\leftarrow}$, which performs a depth-first, right-to-left traversal. Moreover, the information used for the evaluation of attributes at a node π is the same as the information needed to compute the transitions at π. In the practical implementation (which will be addressed in Section 5.4.6), where transitions are computed as they are needed during the run of $B_G^{\leftarrow}$, the attributes can be thus computed at minimal costs.

Example 5.20. Consider the binary query $Q_2 = (G, \{(x_b, x_c)\}$ from Example 5.11 (on page 232), locating the b and the immediately following c children of a node a whose ancestors are exclusively a nodes, on the tree depicted in Figure 5.4 (on page 220). Figure 5.11 depicts how the l_1 and l_2 attributes are computed along the run of $B_G^{\leftarrow}$ on the input annotated by the run of $A_G^{\rightarrow}$ (shown in Figure 5.8 on page 229). The order of computation performed by the second automaton is the same as in the unary case (which was depicted in Figure 5.10 on page 238). Note that nodes are identified by ordinal numbers rather than by paths in order to increase readability. The attributes l_1, l_2 for an element x are depicted as ${}^{l_1}_{l_2}x$. Attributes with value $\varnothing$ are omitted.

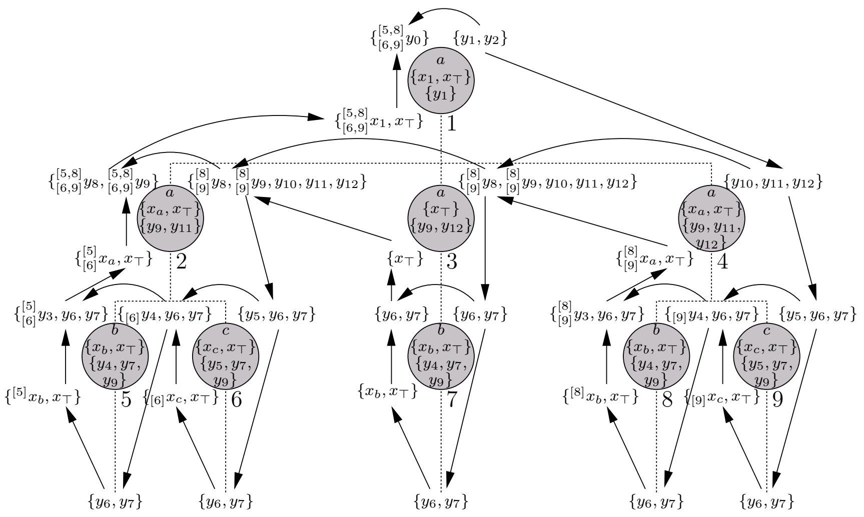


Fig. 5.11 Evaluation of the l_1 and l_2 attributes.

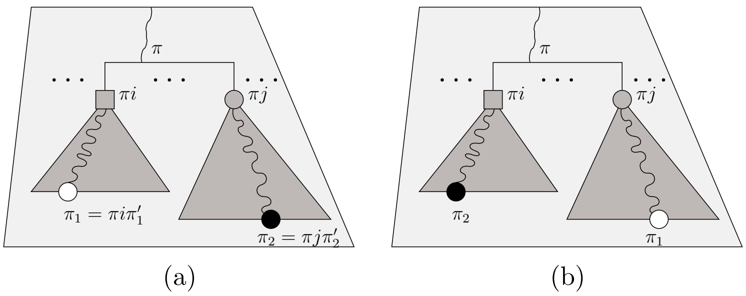
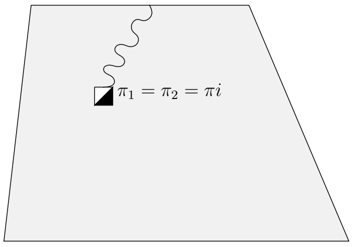


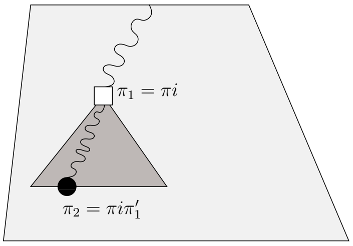
Fig. 5.12 Relative positions of matches: π is nearest common ancestor or λ.

Locating Binary Matches Figure 5.12 (a) and (b), and Figure 5.13 (c), (d) and (e) show all possible relative positions of the primary (depicted in white) and the secondary component (depicted in black) of one binary match (π_1, π_2). In all five situations, due to the construction above, π_1 and π_2 belong to the attributes of one of the tree state $p_{\pi i}$ or forest state $q_{\pi i}$ in which the automaton reaches node πi (depicted by a square). This is where the binary match (π_1, π_2) will be detected at the $\mathit{Side}^{\leftarrow}(q_{\pi i}, p_{\pi i})$ transition.

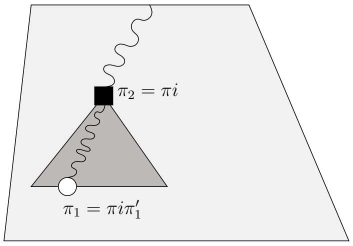
To see how, we need to observe that our construction ensures the following invariants:



(c)



(d)



(e)

Fig. 5.13 Relative positions of matches: equal, or one is a proper ancestor of the other.

(i_1) A node π_1 belongs to the l_1 or l_2 attribute of an element x of a tree state computed for a node πi iff π_1 is below πi and there is a derivation of the input forest which labels πi with x and π_1 with x^1 or x^2, respectively.

(i_2) A node π_2 belongs to the l_1 or l_2 attribute of an element y of a forest state in which $B_G^{\leftarrow}$ arrives at a node πi iff π_2 is in some right sibling subtree and there is a derivation of the input forest which labels πi with x, the label of the NFA transitions coming into y, and π_2 with x^1 or x^2, respectively.

This is formally expressed by the following theorem in which the involved nodes are named as in Figure 5.12 (a) (or (b)):

Theorem 5.21. *(Invariants ensured by the construction)*

(i_1) If $y \in \overrightarrow{q}_{\pi i} \cap q_{\pi i}$, $x \in p_{\pi i}$, $(y', x, y) \in \delta$ *for some* y', x *then*
$\pi_1 \in x.l_1$ *(or* $\pi_1 \in x.l_2$*)* *iff*
$\pi_1 = \pi i \pi'_1$, $\exists f_1$ *s.t.* $(f, f_1) \in \mathcal{D}eriv_G$, $lab(f_1[\pi i]) = x$ *and*
$lab(f_1[\pi_1]) = x^1$ *(or* $lab(f_1[\pi_1]) = x^2$*, respectively).*

(i_2) $y \in \overrightarrow{q}_{\pi i} \cap q_{\pi i}$, $x \in p_{\pi i}$, $(y', x, y) \in \delta$ *and* $\pi_2 \in y.l_2$ *(or* $\pi_2 \in y.l_1$*)*
iff
$\pi_2 = \pi j \pi_2'$, $j > i$, $\exists f_2$ *s.t.* $(f, f_2) \in \mathcal{D}eriv_G$, $lab(f_2[\pi i]) = x$ *and* $lab(f_2[\pi_2]) = x^2$ *(or* $lab(f_2[\pi_2]) = x^1$*, respectively)*

Proof. A formal proof is given in Section 5.8.1. The idea is presented forthwith. □

Let $x \in p_{\pi i}$, $y \in \overrightarrow{q}_{\pi i} \cap q_{\pi i}$, $(y', x, y) \in \delta$. Let $\pi_1 \in x.l_1$ and $\pi_2 \in y.l_2$. It is easy to see that (i_1) directly implies (p) and (i_2) implies (s). Less obvious but still true is that (i_1) and (i_2) also imply (r). It follows that every pair formed with $\pi_1 \in x.l_1$ and $\pi_2 \in y.l_2$ is a binary match.

To see why (i_1) and (i_2) imply (r), let us define a function which given a forest f, a node π and a tree t constructs a forest f_1 by replacing in f the subtree located at π with t, formally $f_1 = f/^{\pi}\, t$ where:

$$\begin{aligned}(t_1 \dots t_i \dots t_n)/^{i}\, t &= t_1 \dots t \dots t_n\\ (t_1 \dots t_i \dots t_n)/^{i\pi}\, t &= t_1 \dots a\langle f/^{\pi}\, t\rangle \dots t_n, \text{ if } t_i = a\langle f\rangle\end{aligned}$$

If $f_1 = f/^{\pi}\, t$, we say that f_1 is obtained by *grafting* t into f at π.

The following theorem observes that given two derivations of a forest f which label a node π with the same symbol, a new derivation can be obtained by doing a relabeling of f in which the nodes below π are labeled as in one of the derivations and the rest of nodes as in the other.

Theorem 5.22. *If* $(f, f_1) \in \mathcal{D}eriv_G$*,* $(f, f_2) \in \mathcal{D}eriv_G$ *and* $lab(f_1[\pi]) = lab(f_2[\pi])$ *then* $(f, f_1/^{\pi}\, f_2[\pi]) \in \mathcal{D}eriv_G$ *and*

$$lab((f_1/^{\pi}\, f_2[\pi])[\pi_1]) = \begin{cases} lab(f_2[\pi_1]), & \text{if } \pi_1 = \pi\pi_2 \text{ for some } \pi_2\\ lab(f_1[\pi_1]), & \text{otherwise}\end{cases}$$

Proof. The proof is given in Section 5.8.2. □

Using the notations of Theorem 5.21, let $f' = f_2/^{\pi i}\, f_1[\pi i]$. It follows by Theorem 5.22 that $(f, f') \in \mathcal{D}eriv_G$, $f'[\pi_1] = x^1$ and $f'[\pi_2] = x^2$, thus (r) also holds for (π_1, π_2). It follows that (π_1, π_2) is a binary match.

Example 5.23. Consider the side transition at node 8 in Figure 5.11 (page 242). The element ${}_{[9]}y_4$ in the forest state in which node 8 is reached denotes that node 9 is a secondary match in the part of the tree already visited. The element ${}^{[8]}x_b$ in the tree state synthesized at node 8 denotes that 8 is a primary match found in the subtree 8. The fact that 8 and 9 are defined with respect to the same derivation can be seen from the fact that x_b is the label of the incoming transitions into y_4. Thus $(8, 9)$ is a binary match.

Similarly, $(5, 6)$ is detected as a match at the side transition at node 5.

Thereby, we obtain how binary matches can be detected (where cases (a)-(e) correspond to the situations depicted in Figures 5.12 and 5.13):

(a) Every pair (π_1, π_2) with $\pi_1 \in x.l_1$, $\pi_2 \in y.l_2$ is a binary match, as presented above.
(b) Similarly, one can show that every pair (π_1, π_2) with $\pi_1 \in y.l_1$, $\pi_2 \in x.l_2$ is a binary match.
(c) If $x = x^1 = x^2$ it is easy to see in invariant (i_1) that by definition $(\pi i, \pi i)$ is a binary match.
(d) If $x = x^1$ we also have by (i_1) that every pair $(\pi i, \pi_2)$ with $\pi_2 \in x.l_2$ is a binary match.
(e) Similarly, if $x = x^2$ we have by (i_1) that every pair $(\pi_1, \pi i)$ with $\pi_1 \in x.l_1$ is a binary match.

To see that *all* binary matches are detected as above, let, conversely, (π_1, π_2) be a binary match. If $\pi_1 = \pi i \pi_1'$ and $\pi_2 = \pi j \pi_2'$, $j > i$ then there is f' with $(f, f') \in \mathcal{D}eriv_G$, $f'[\pi i \pi_1'] = x^1$ and $f'[\pi j \pi_2'] = x^2$. Let $f'[\pi i] = x$. It follows by Corollary 5.16 (page 237) that there are $y' \in q_{\pi i} \cap \overrightarrow{q}_{\pi i}$, $(y_1', x, y') \in \delta$. By Corollary 5.17 (page 237) we have that $x \in p_{\pi i}$. By (i_1) it follows that $\pi_1 \in x.l_1$. By (i_2) there are $y \in \overrightarrow{q}_{\pi i} \cap q_{\pi i}$, $x \in p_{\pi i}$, $(y_1, x, y) \in \delta$ and $\pi_2 \in y.l_2$. It follows that there is πi, $x \in p_{\pi i}$, $y \in \overrightarrow{q}_{\pi i} \cap q_{\pi i}$, $(y_1, x, y) \in \delta$, $\pi_1 \in x.l_1$ and $\pi_2 \in y.l_2$.

Similarly, for $\pi_2 = \pi i \pi_2'$, $\pi_1 = \pi j \pi_1'$, $j > i$, or $\pi_1 = \pi_2$, or $\pi_2 = \pi_1 i \pi_2'$, or $\pi_1 = \pi_2 i \pi_1'$ we obtain the converse of (b), (c), (d) or (e), respectively.

We have thus proven the following theorem:

Theorem 5.24. *A pair (π_1, π_2) is a binary match iff there is $\pi \in N(f)$, $x \in p_\pi$, $y \in q_\pi \cap \overrightarrow{q}_\pi$, $(y', x, y) \in \delta$ and either:*

(a) $\pi_1 \in x.l_1$, $\pi_2 \in y.l_2$ *or*
(b) $\pi_1 \in y.l_1$, $\pi_2 \in x.l_2$ *or*
(c) $\pi_1 = \pi_2 = \pi$, $x = x^1 = x^2$ *or*
(d) $\pi_1 = \pi$, $x = x^1$, $\pi_2 \in x.l_2$ *or*
(e) $\pi_2 = \pi$, $x = x^2$, $\pi_1 \in x.l_1$.

Complexity

Let n be the size of the input forest f, i.e. the number of nodes in f. The complexity of answering a binary query is given by the complexities of running $A_G^{\rightarrow}$ and $B_G^{\leftarrow}$, computing the l_1 and l_2 attributes and that of locating binary matches.

The automaton $A_G^{\rightarrow}$ executes at each node one *Down*, one *Side* and one *Up* transition. As one can see in the definitions of the transitions, the time cost of each of these transitions does not depend on f. The run of $A_G^{\rightarrow}$ requires thus time $\mathcal{O}(n)$. Similarly, the run of $B_G^{\leftarrow}$ needs time $\mathcal{O}(n)$.

The l_1 and l_2 attributes have to be computed for each component of the states obtained after a $Side^{\leftarrow}$ and $Up^{\leftarrow}$ transition. For the complexity assessment let us suppose that m is the larger of the numbers of primary and secondary matches in f.

Consider now an $Up^{\leftarrow}$ transition. The set $x.l_1$ of primaries for each component is computed as the union of the sets $y.l_1$ of primaries. As the number of sets $y.l_1$ does not depend on f, and a set union can be computed in time $\mathcal{O}(m)$, the time for computing $x.l_1$ is in $\mathcal{O}(m)$. Similarly, $x.l_2$ is computed in time $\mathcal{O}(m)$. As the number of elements in the computed state does not depend on f either, executing $Up^{\leftarrow}$ can be done in time $\mathcal{O}(m)$. The sets $y.l_1$ and $y.l_2$ computed at $Side^{\leftarrow}$ transition for each component of the state are similarly computed in time $\mathcal{O}(m)$. It follows that the attributes can be computed in time $\mathcal{O}(n \cdot m)$.

As for the complexity of locating matches, let p be the number of binary matches in f. Note that each of the binary matches is located at exactly one of the $Side^{\leftarrow}$ transitions, namely at the $Side^{\leftarrow}$ transitions over the ancestor of one of the primary or secondary, which is a sibling of an ancestor of the other. As remembering each binary match only requires constant time, locating binary matches has the overall time cost in $\mathcal{O}(p)$.

The total time cost of answering binary queries is thus in $\mathcal{O}(n \cdot m + p)$. Since $p \leq m^2$ and $m \leq n$, the theoretical worst cost is in $\mathcal{O}(n^2)$. We have thus proven the following theorem.

Theorem 5.25. *The theoretical worst case complexity of answering binary gramar queries is quadratic in the size of the input document.*

This theoretical worst case corresponds to the case in which every pair of nodes from f is a binary match. In practice, however, the number of primary, secondary and binary matches tend to be irrelevant as compared to the input size. In this case, the time consumed is rather linear in the input size and binary queries can be answered almost as efficiently as unary queries.

5.4.4.2 *Recognizing General Binary Queries*

Let $Q = (G, T)$, where $T \subseteq X^2$, be a binary query. The construction is similar to that for simple binary queries but has to keep a set attribute for each non-terminal occurring in T.

Formally, let $X_1 = \{x \mid (x, x') \in T \text{ or } (x', x) \in T\} = \{x_1, \ldots, x_n\}$.

Rather than with two attributes as in the case of simple binary queries, we equip each element of a state in which $B_G^{\leftarrow}$ visits the input with n attributes $l_1, \ldots, l_n$. The attributes l_i are computed as follows:

- If $y \in F_0$ (the initial state of $B_G^{\leftarrow}$) or $y \in \mathit{Down}^{\leftarrow}(q, (a, \overrightarrow{p}, \overrightarrow{q}))$ then $y.l_i = \varnothing$
- If $x \in \mathit{Up}^{\leftarrow}(q, (a, \overrightarrow{p}, \overrightarrow{q}))$ then

$$x.l_i = \begin{cases} \{\pi\} \cup \bigcup\{y.l_i \mid y \in q, y = y_{0,j}, x \to a\langle r_j \rangle\}, & \text{if } x = x_i \\ \bigcup\{y.l_i \mid y \in q, y = y_{0,j}, x \to a\langle r_j \rangle\} & , \text{ otherwise} \end{cases}$$

- If $y \in \mathit{Side}^{\leftarrow}(q, p, (a, \overrightarrow{p}, \overrightarrow{q}))$ then

$$y.l_i = \bigcup\{y_1.l_i \cup x.l_i \mid (y, x, y_1) \in \delta, y_1 \in q \cap \overrightarrow{q}, x \in p\}$$

for $i = 1, \ldots, n$.

As in the case of simple binary queries, matches are found at $\mathit{Side}^{\leftarrow}$ transitions of $B_G^{\leftarrow}$. Let $\mathit{Side}^{\leftarrow}(q_\pi, p_\pi)$ be such a transition and let $x \in p_\pi$, $y \in q_\pi \cap \overrightarrow{q}_\pi$, $(y_1, x, y) \in \delta$. In order to find binary matches, one has to look for every $(x_i, x_j) \in T$ into the l_i and l_j attributes. Finding the match pairs is achieved similarly to finding match pairs in the case of simple binary matches.

Theorem 5.26. *A pair (π_1, π_2) is a binary match iff there is $\pi \in N(f)$, $(x_i, x_j) \in T$, $x \in p_\pi$, $y \in q_\pi \cap \overrightarrow{q}_\pi$, $(y_1, x, y) \in \delta$ and either:*

(a) $\pi_1 \in x.l_i$, $\pi_2 \in y.l_j$ *or*
(b) $\pi_1 \in x.l_j$, $\pi_2 \in y.l_i$ *or*
(c) $\pi_1 = \pi_2 = \pi$, $x = x_i = x_j$ *or*
(d) $\pi_1 = \pi$, $x = x_i$, $\pi_2 \in x.l_j$ *or*
(e) $\pi_1 = \pi$, $x = x_j$, $\pi_2 \in x.l_i$.

Proof. By definition, (π_1, π_2) is a binary match iff there is $(x_i, x_j) \in T$ and (π_1, π_2) is a simple binary match for $(G, (x_i, x_j))$. The proof follows immediately from Theorem 5.24 by observing that the attributes l_1 and l_2 from the construction for $(G, (x_i, x_j))$ equal l_i and l_j, respectively. □

In a similar manner as in the case of simple binary queries one obtains that the complexity of answering binary queries is quadratic in the input size in the worst case and rather linear in the average case.

5.4.5 *Recognizing K-ary Queries*

The construction introduced for the evaluation of binary queries can be in principle extended to work for k-ary queries for arbitrary k's. In order to locate matches of a query $(G,(x_1,\ldots,x_k))$ the construction has to keep a separate set attribute for each non-empty subset $A \subset \{x_1,\ldots,x_k\}$. The set attribute for A then contains all tuples of nodes which form a partial match corresponding to the elements in A. This is necessary because a complete match can be obtained by considering any pair of complementary partial matches. For example, for a query $(G,(x_1,x_2,x_3))$ one needs to consider putting together the partial matches corresponding to $\{x_1\}$ and $\{x_2,x_3\}$, or $\{x_2\}$ and $\{x_1,x_3\}$, or $\{x_3\}$ and $\{x_1,x_2\}$, respectively. However, the complexity of the construction grows exponentially with k which makes it impracticable for large k's.

In the XML practice however many queries are expressed via XPath select patterns which conceptually are binary relations (namely, between the context node for the evaluation of the pattern and the set of nodes selected in that context). Therefore, binary queries can be satisfactorily used to cover a wide range of actual XML applications.

Nevertheless, it is possible to implement k-ary queries very efficiently if one adopts a *disambiguating policy* for grammars. Our queries so far consider *all* possible derivations w.r.t. the given input grammar. Following all these derivations in parallel is the source of the exponential blowup in the evaluation complexity. A disambiguating policy is a set of rules which allows the choice of exactly *one* derivation from among the different derivations.

One disambiguating policy could be obtained for instance by requiring one to (1) always consider *left-longest* sequences in fulfillments of content models, i.e., in NFA runs, to prefer NFA transitions corresponding to symbols which are as left as possible in the corresponding regular expressions; and (2) always choose the first applicable production in the input grammar. A similar policy was in essence originally adopted in XDuce [36,37], in the context of its functional style pattern matching for XML documents (a comparison of XDuce with our approach is provided in Section 5.4.7). Similarly, one can adopt a *right-longest* policy, or non-deterministically choose one of the possible derivations, if one is for example interested in just one match, as in the case of a *one-match* policy.

The implementation in the presence of such a disambiguating policy can use the same first traversal of $A_G^{\rightarrow}$ to annotate the input tree. Only the second automaton $B_G^{\leftarrow}$ has to proceed differently. The adopted policy allows the maintenance of its states as singletons, rather than sets, by indicating exactly one NFA transition (y_1, x, y), and exactly one grammar production $x \rightarrow a\langle r_j \rangle$ to be considered at the $Side^{\leftarrow}$ and $Down^{\leftarrow}$ transitions, respectively. The x's considered at the NFA transitions are then the labels of the sought-after derivation. The k match nodes can be thus directly read from the annotation by the second automata, getting thus even linear time complexity.

5.4.6 *Implementation Issues*

The constructions presented in the previous section for the evaluation of unary and binary queries can be put to work in the XML practice as it will be addressed in Section 5.5. Their efficient implementation is supported by the consideration of a couple of practical aspects as follows (presented in more detail in Neumann [53]).

Lazy Evaluation The pushdown automata are efficiently implemented by computing their transitions only as they are needed. Transitions which are not required for the traversal of the input are not computed. This avoids the computation of possibly exponentially large transition tables. The number of transitions that are actually computed is at most linear in the size of the input document.

Caching Moreover, the automata do not need to compute transitions at every node, as many transitions are repeatedly executed. The first time a transition is needed, its computed value is cached, and the cached value is simply looked up for its subsequent uses. In practice only few transitions need to be computed even for large XML documents.

Pre-processing Further, information which is repeatedly used for the computation of transitions, and which does not depend on the input document can be computed by a preprocessor of the query and directly accessed when needed. For example, a transition $Down(q, a)$ is computed (only when the automaton $A_G^{\rightarrow}$ arrives in forest state q at a node labeled a, and only if the transition was not already computed) using the definition:

$$Down(q,a) = \{y_{0,j} \mid y \in q,\ (y,x,y_1) \in \delta,\ x \to a\langle r_j \rangle \text{ for some } x, y_1\}$$

To do so it can use the following pre-processed information:

$$\begin{array}{l} y_0s\_for\_y\ y = \{y_{0,j} \mid (y,x,y_1) \in \delta,\ x \to a\langle r_j \rangle\} \text{ for all } y \in Y \\ y_0s\_for\_a\ a = \{y_{0,j} \mid x \to a\langle r_j \rangle\} \text{ for all } a \text{ occurring in } G \end{array}$$

Therefore:

$$Down(q,a) = \begin{cases} y_0s\_for\_a\ a \cap \bigcup_{y \in q} y_0s\_for\_y\ y, & \text{if } a \text{ occurs in } G \\ \varnothing & , \text{ otherwise} \end{cases}$$

Similar information is computed by the preprocessor for supporting the other transitions of the pushdown automata.

5.4.7 *Bibliographic Notes*

In this section we briefly survey existing query approaches for tree-structured data and relate them with the previously introduced ones where possible. In particular the research on XML processing has been extremely prolific in the recent years. No claim can be made with regard to the exhaustiveness of the presentation.

We consider the following comparison criteria:

Expressiveness As a benchmark for assessing the expressive power of query languages for trees we use the monadic second order logic (MSO logic), which has proven to be particularly convenient in the context of tree-automata and logic-based approaches [62].

Extensibility to the k-ary case Most of the proposals only consider the unary case. We indicate where a query approach can be straightforwardly extended to express k-ary queries.

Evaluation complexity An important factor for practical implementations is how efficient are the proposed query languages. Despite the relatively large number of proposals, there are not many for which practical algorithms have been introduced, and even fewer practical query languages are actually implemented.

Some of the approaches have been summarized in a survey done by Neven and Schwentick [62], which mainly addresses only the unary queries. A somewhat dated overview on practical XML query languages can be found in [25]. We start with a very brief summary of our grammar queries.

5.4.7.1 *Forest Grammar Queries*

The specification of unary queries using forest grammars has been introduced in [53]. The definition of a match node of a query is given there *locally*, recursively in terms of the queries matched by the father node and the structural constraints fulfilled by the sibling nodes. In order to make our unary query definition extensible to the k-ary case we gave it in the global terms of derivations, which denote *global* computations over the input [8,9]. As presented in Section 5.4.2, the expressive power of our k-ary queries equals MSO formulas with k free variables. Evaluating unary grammar queries is done in time linear in the size of the input. Binary queries can be always evaluated in quadratic time in the worst case, yet most queries are answered in linear time as discussed in Section 5.4.4.1. Unary and binary grammar queries are completely implemented in the practical XML querying tool Fxgrep [54]. The complexity of evaluating k-ary queries generally grows exponentially with k. Nevertheless, by adopting a disambiguating policy to chose one among the different derivations allowed by a grammar, it is possible to evaluate k-ary queries in linear time, as discussed in Section 5.4.5.

5.4.7.2 *XPath Evaluation and Expressiveness*

XPath has established itself since its standardization by the W3C Consortium as the most prominent XML query language. XPath will be introduced and compared with our grammar queries in more detail in Section 5.5.5.

Gottlob *et al.* [29] find out by experimenting with practical XPath processors that their evaluation time might grow exponentially with the size of the considered XPath pattern. This inefficiency is ascribed to a naive implementation which literally follows the XPath specification as a succession of selection and filtering steps. As opposed to this evaluation strategy, the authors propose a *bottom-up* evaluation, that is, an evaluation in which expressions are computed by first evaluating their sub-expressions and then combining their results. They show that the theoretical evaluation time might be bound by the class $O(n^3 \cdot m^2)$ where n is the size of the document and m the size of the query. The practicability of the approach is limited given its space complexity bounded by $O(n^4 \cdot m^3)$, which is due to the tabulation of the results of evaluating each subexpression considered during the evaluation. In principle, the reason for the space inefficiency is the same as for bottom-up automata, namely their ignorance of the upper context (opposedly to pushdown automata), which should nevertheless be available in a practical implementation. To alleviate this problem, Gottlob *et al.* give a translation of their bottom-up algorithm into a top-down one. It is shown that this has the same time-complexity and argued that it may compute less intermediate results.

The authors identify a fragment of XPath, called *Core XPath* which can be evaluated more efficiently. Core XPath is XPath without arithmetical or data value operations, and so basically expressible by grammar queries as presented in Section 5.5.5. Core XPath is not first-order complete [45]. It has been shown that Core XPath can be made first-order complete by extending it with the ability to specify conditions that need to be satisfied by every node on a path between two specific nodes [44]. The complexity of the evaluation scheme for Core XPath proposed in [29] is $O(n \cdot m)$. As opposed to Fxgrep where at most two traversals are needed both for unary and binary queries, the number of traversals of the input document is bounded by the number of steps in the pattern. Since addressing implementation strategies for XPath, [29] only considers unary queries.

5.4.7.3 *Query Automata*

Query automata (QA) defined by Neven and Schwentick [59] are *two-way tree automata*, i.e. automata which can perform both up and down transitions, together with a distinguished set of *selecting* states. A query automata is a unary query locating the nodes of some input which are visited at least once in a selecting state. Simple QA on unranked trees are less expressive even than first order logic. Intuitively, the reason for this limitation is due to their inability to pass information from one sibling to another. To achieve the expressive power of MSO logic, QA have to be extended with *stay transitions*, at which a two way string transducer reads the string of states of the children of some node and outputs for each child a new state. Queries of arity k are not considered. The complexity of query evaluation is linear in the size of the input [62].

5.4.7.4 *Selecting Tree Automata*

Similarly to query automata, Frick *et al.* [27] use tree automata extended with a set of selecting states (called *selecting tree automata*) to specify unary queries. The semantics of queries is defined in terms of *runs* of the tree automata, either in an *existential* or a *universal* setting. In the existential setting, a node in some input is a match, if *some* accepting *run* on the input visits the node in a selecting state. In the universal one, a node is a match if *every* accepting *run* on the input visits the node in a selecting state. It is shown that the existential and the universal queries are equally expressive.

Runs of tree automata are in fact the same as derivations w.r.t. tree grammars, with automata states corresponding to non-terminals in grammars. The existential queries are thus basically unary forest grammar queries restricted to the ranked case. The proof that the expressiveness of selecting tree automata equals MSO logic unary queries comes therefore as no surprise.

For the evaluation of queries an algorithm is proposed which performs a bottom-up followed by a top-down traversal of the algorithm, the complexity of which is $O(m \cdot n)$, where m is the size of the encoding of the automaton and n the size of the input.

The selecting automata defined in [27] are only used to express unary queries. As previously argued, the grammar queries formalism presented in Section 5.4.1.2 and introduced in [8] and [9] is in fact a generalization of the existential query formalism based on selecting automata, to k-ary queries on unranked trees. Recently, Niehren *et al.* [64] consider both existential and universal k-ary queries defined via selecting tree automata and show that they are equally expressive also in the k-ary case. Furthermore, [64] provide a proof that the k-ary queries defined by selecting tree automata capture precisely the MSO expressible queries, and that this result carries over to unranked trees. From this follows that grammar queries have the same expressiveness as MSO queries, as already mentioned in Section 5.4.2. Another implication is that forest grammar queries, defined using an existential semantics, also capture universal queries.

As a particularly efficiently implementable case, Niehren *et al.* identify k-ary queries specified via *unambiguous tree automata*, i.e. tree automata for which for every input tree there is at most one successful run. The proposed construction identifies the unique run in a bottom-up followed by a top-down traversal of the input and has a time-complexity linear in the input size. Specifying queries by unambiguous tree automata is similar to specifying a grammar query together with a disambiguating policy for derivations as mentioned in Section 5.4.5.

5.4.7.5 *Attribute Grammar Queries*

Neven and Van den Bussche [58] used attribute grammars (AGs) to specify queries on derivation trees of context-free grammars, hence they deal with queries on ranked trees. In particular they consider *boolean-valued attribute grammars* with propositional logic formulas as semantic rules (BAG). A BAG together with a designated boolean attribute defines a unary query which retrieves all nodes at which the attribute has the value *true*. It is shown that BAG unary queries have the same expressiveness as MSO logic.

In order to deal with unranked trees, Neven extends the BAG query formalism [57]. He introduces *extended attribute grammars* which work on extended context free grammars (ECFGs) rather than on CFGs. Since an ECFG production has a regular expression r on the right hand side, the number of children of a node defined via the production may be unbounded. To be able to define an attribute for each child, the semantic rules identify them via the unique position in r to which the child has to correspond. A semantic rule for a given position defines attribute values for all children corresponding to that position. The semantic rule is basically given as a regular expression r_1 containing a special place-marker symbol "#". The attribute value of a child is assigned true if the sequence of children containing the place-marker "#" in front of the child under consideration matches the regular expression r_1. This allows the specification of the left and right context of a node. Neven shows that the query formalism based on extended BAGs is equally expressive as MSO logic.

BAGs can only express unary queries. No considerations are made regarding the complexity of query evaluation.

One possibility for expressing k-ary queries with AGs has been investigated by Neven and Van den Bussche [58] only for ranked trees. This is achieved via *relation-valued attribute grammars* (RAG). A RAG defines a query as some designated attribute at the root. The matches of the query are given by the relation computed as the value of this attribute. They show that RAGs are more expressive than MSO for queries of any arity.

Attribute grammars have been considered also in the context of XML stream processing [40, 52, 72]. We briefly review these approaches in Section 5.6.3.

5.4.7.6 *Regular Hedge Expressions*

In [50], Murata considers providing a specification language to allow for the specification of context of a node more precisely than by simply expressing conditions on the path from the root to the node. The query formalism proposed is similar to the original proposal by Neumann and Seidl using μ-formulas [56]. A query is specified as a pair consisting of two expressions for specifying the structure and the context of a match node, respectively. Murata's *hedge regular expressions*, used to express the structure of a match, are able to express regular forest languages. To express the contextual condition, the formalism for specifying structure is extended by using a special symbol to denote the desired node (obtaining *pointed* formulas). Murata, previously used pointed trees for specifying contextual conditions on ranked trees in [49].

The formalism presented in [50] is targeted at unary queries. The evaluation time of the queries is proven to be linear in the size of the input. The expressive power of the selection queries using regular hedge expressions is equal with MSO logic.

5.4.7.7 *Tree-walking Automata*

Tree-walking Automata (TWAs) are sequential automata working on trees. In contrast to classic tree automata, in which the control state is distributed at more than one node, a TWA always considers exactly one node of the input. Depending on the label of the node and its location the automaton changes its state and moves to a neighbor node[3]. A TWA specifies the language of trees on which the automaton starts at the root and ends also at the root in an accepting state. TWAs can be used to specify unary queries in a similar way as the selecting tree automata, by defining a set of selecting states. Recently it was proven that TWAs cannot define all regular tree languages [12]. This implies that TWAs are less expressive than unary MSO queries.

Brüggemann-Klein and Wood [14] proposed *caterpillars* as a technique for specifying context in queries. A caterpillar is a sequence of symbols from a *caterpillar alphabet* denoting movements and node-tests. The symbols specify a movement to the parent, left- or right-sibling, left- or right-most child of a node or testing that a node is root, leaf, the first or the last among its siblings. A caterpillar sequence specifies a unary query as the nodes starting from which the sequence of movements and tests can be successfully performed. More generally the query can be specified via *caterpillar expressions* which are regular expressions over a caterpillar alphabet. Evaluating the matches specified by a caterpillar expression can be done in $O(m \cdot n)$, where m is the size of the input and n is the number of transitions in the finite-state automaton corresponding to the caterpillar

[3]Pushdown forest automata can be thus also seen as a kind of tree-walking automata enhanced with a pushdown.

expression. The formalism can express only unary queries.

Expressing binary queries on trees via TWAs was considered by van Best in [79]. A tree-walking automaton can compute a binary relation on tree nodes by starting at one node and finishing at the other. It is argued that ordinary TWAs cannot compute all MSO definable binary queries. To achieve the expressiveness of MSO binary queries *tree-walking marble/pebble automata* are introduced. A tree-walking marble/pebble automaton is a TWA enhanced with a number of *marbles* and one pebble. The automaton can place and pick-up the marbles and the pebble while visiting the nodes of the input, with the restriction that the last one placed must be the first one to be picked-up. Furthermore, when placing a marble on a node, the automaton is restricted to walk only below this node. It is shown that these automata can be used to compute binary relations defined by MSO logic formulas. The complexity of query evaluation is not assessed in [79].

To be able to deal with XML text nodes and attributes (generically called *data values*) and to allow the use of equality tests on them in queries, two extensions of TWAs are suggested by Neven *et al.* in [62, 63]. The extensions are given for string automata but they carry over also to tree automata. In the first approach the automata are extended with a finite number of registers and can check whether the data value of a node equals the content of some register when performing its transitions. The expressive power of this register automata is comparable neither with FO logic nor with MSO logic. That is, there are FO queries which are not expressible by the automata, but also queries expressible by these automata which are not captured by any MSO query. In the second approach, the automata are equipped with a number of pebbles which they can use according to a stack policy (last dropped first removed). The expressiveness of these pebble automata is shown to lie between FO and MSO logic. The complexity of query evaluation for both register and pebble automata is shown to be in PTIME.

5.4.7.8 *Pure Logic Formalisms*

In logical formalism queries are expressed directly as logic formulas. A query containing a free node variable expresses a unary query. Going from unary to k-ary queries is in principle easily done by using formulas with k free variables instead of formulas with one free variable. Queries on tree-structured data can be expressed using MSO logic on trees. Many of the query approaches mentioned so far have exactly the same expressiveness as MSO logic. Nevertheless expressing queries as MSO logic formulas is not practicable due to the prohibitively high (non-elementary) evaluation complexity.

Neven and Schwentick [60, 73] consider a restriction of MSO logic as a query mechanism. The logic, called FOREG, is an extension of FO logic which allows the formulation of constraints on children of nodes and on nodes lying on a given path, called *horizontal* and *vertical path formulas*, respectively. Horizontal and vertical path formulas are regular expression over formulas which are to be satisfied by the children of a node, or the pairs of end nodes of the edges on a path, respectively. It is shown that the expressiveness of FOREG lies precisely between FO and MSO logic.

Still, the evaluation complexity of FOREG queries is not practicable. Neven and Schwentick present syntactic restrictions of FO, FOREG and MSO logic which are still as expressible as the original logics but for which unary queries can be more efficiently evaluated. The restriction, basically allows the formulation of properties only on paths from the root to a node, rather than on arbitrary paths. It is shown that unary queries expressed in the restriction of FOREG (called *guarded FOREG*) and MSO (*guarded MSO*) can be evaluated in time $O(n \cdot 2^m)$ and $O(n \cdot 2^{m^2})$, respectively, where n is the size of the input and m the size of the formula.

Going from unary to k-ary queries in the guarded logic formalism is not directly possible due to the restricted use of variables in the guarded case. Nevertheless, Schwentick shows [73] that expressing queries of arbitrary arity is possible by suitable combinations of unary queries as above and an additional kind of horizontal path formula (called *intermediate path formula*) which is able to talk about the sequence of siblings between the ancestors of two arbitrary nodes. While the expressiveness of the logic formalism is not modified by the addition of intermediate path formulas, it is shown that an algorithm exists which checks in time $O(n \cdot 2^m)$ whether a tuple of nodes verifies a formula on some input. Answering queries using this algorithm implies generating all the k-tuples of nodes from the input, incurring $\mathcal{O}(n^k)$ time. This gives the evaluation of k-ary queries the $\mathcal{O}(n^{k+1} \cdot 2^m)$ complexity. In particular, binary queries can be answered thus in time $\mathcal{O}(n^3 \cdot 2^m)$, which is less efficient when compared with the complexity of our algorithm.

5.4.7.9 *Numerical Document Queries*

Seidl *et al.* [75] introduce *Presburger tree automata*, which are able to check numerical constraints on children of nodes expressible by Presburger formulas. Whether the children of a node fulfill a Presburger constraint is independent of their relative order. Correspondingly, the tree automata checking Presburger constraints can be considered automata on unordered trees. Unary queries are defined similarly to selecting automata by specifying a distinguished set of states of a Presburger tree automaton. It is shown that unary query evaluation has linear time complexity in the size of the input. To capture the expressiveness of the Presburger tree automata, the authors introduce an extension of MSO logic with Presburger predicates on children of nodes (called PMSO). They show that, on unordered trees, PMSO is equivalently expressive with Presburger tree automata.

Furthermore, they consider expressing both numerical and order constraints on children. For this they extend the Presburger tree automata by allowing regular expressions of Presburger constraints in transitions. It is shown that expressiveness of these automata is captured by PMSO on ordered trees and that evaluating unary queries can be performed in polynomial time. As a case of special interest, they further consider expressing either numerical- or order-constraints on children, depending on the label of the father node. It turns out that the corresponding tree automata have the same expressiveness as the corresponding PMSO logic and that the definable unary queries can be evaluated in linear time.

5.4.7.10 *Tree Queries*

To the best of our knowledge, none of the querying approaches mentioned so far, with the exception of grammar queries (available in Fxgrep), have been implemented in practical XML querying languages.

A practically available system for XML querying is X^2 proposed by Meuss *et al.* [47]. The basic capabilities of the query language of X^2 allow it to specify and relate query nodes via child and descendant relations, labels of nodes or tokens occurring in text nodes. Supplementary constraints make it possible to specify immediate vicinity or relative order of siblings and also to mark nodes as leftmost or rightmost children. Formally, the tree queries expressible by X^2 are *conjunctive queries over trees.* The nodes identifiable via these queries are subsumed by Core XPath (as shown by Gottlob *et. al* in [31]) and therefore also by Fxgrep. Nevertheless, in contrast to XPath, X^2 is able to express k-ary queries, since one answer to a X^2 query is a variable assignment mapping each node in the query to a node in the input (hence, k is the number of nodes in the query). Because the number of answer mappings can be exponentially large, a novel data structure, the complete answer aggregate (CAA) is introduced by Meuss and Schulz [46]. CAAs represent the answer space in a condensed form and allow its visual exploration, while guaranteeing that all single answers can be reconstructed. The computation of a CAA for a query is in $O(n \cdot log(n) \cdot h \cdot m)$, where n and h are the size and the maximal depth of the input, respectively, and m is the size of the query.

A main concern of X^2 is supporting the specification of queries and navigation within the answer space via a graphical user interface. Among other visual interfaces to XML query languages which have been proposed are XML-GL [15], BBQ [48], Xing [23], or visXcerpt [5].

There are a few more practical XML tools built upon recent research work. Most of them are strictly speaking not query languages but transforming languages. Anyhow, most of the available tools use XPath as a query language. Furthermore, a few more tools emerging from research on XML stream processing are mentioned in Section 5.6.3.

5.5 Practical Application: A Pattern Language for XML Querying

An important design purpose of tools which are to be used by people with various backgrounds, as in the case of XML query languages, is that they are as intuitive and easy to learn as possible. The small number of constructs needed to build complex regular expressions makes them a simple yet powerful way of specifying patterns of symbols. Consequently, they have been intensively used already for searching in flat (i.e. not hierarchically structured) documents.

The classic regular expressions can be used also to search for string patterns in hierarchically structured documents, yet they are not able to exploit the supplementary structure information in order to provide more precise patterns to be recognized. To account for this, the XML query language Fxgrep [54][4] was designed to extend the convenience of using regular expressions from strings to trees and implemented based on the

[4]Fxgrep is an acronym for "the functional XML grep", where "functional" denotes that Fxgrep is written in a functional programming language (SML) and "grep" is an allusion to the Unix string search tool grep [33].

constructions introduced in the previous section. This section is intended to be a primer on Fxgrep .

Fxgrep receives as input a query and an XML input document and responds with the locations in the XML input identified by the query. Queries can be concisely specified via *patterns*, the construction of which is introduced in the next subsections.

5.5.1 *Paths*

A quite familiar representative of hierarchically structured information is a file system, in which directories and files are organized in a tree structure. In this context, a query is simply a file name denoting the path to be followed to a file or directory of interest. It is thus sensible to use paths as a syntactical basis for a query language for XML documents, as XPath [83,84] does, the most prominent XML query language yet. Fxgrep builds upon the same analogy and is correspondingly *syntactically* similar to XPath.

▷ Completely analogously to a file name, the pattern `/a/b/c` returns XML elements labeled `c` which are children of `b` elements contained in the root element `a`.

As opposed to file names, patterns may identify more then one node in the input tree. Also, paths do not need to be completely specified. The *deep-match* construct “`//`” may be used to denote an arbitrary number of steps in a path.

▷ The pattern `/a/b//c` returns XML elements labeled `c` which are descendants of `b` elements contained in the root element `a`.

Besides by their name, the element names in a path may be specified as regular expressions. The regular expression is to be fulfilled by the referred element names and must be single or double quoted and enclosed between the “`<`” and “`>`” symbols, as in the case of an XML element tag.

▷ The pattern `//<'(sub)*section'>` locates all `section`-s, `subsection`-s and `subsubsection`-s elements in the input.

The last step in a path may not only be an element name, but could also be a regular expression specifying a model for a text node.

▷ The pattern `//section/title/"automat(a|on)"` locates the `title`-s of `section`-s containing the words `"automata"` or `"automaton"`.

5.5.1.1 *Regular Path Expressions*

Regular path expressions are known in the literature as regular expressions to be satisfied by the string of node labels on the path from the root to the node of interest. Fxgrep provides regular path expressions.

▷ The pattern `(king/)+ person` stands for `king/king/.../king/person` and identifies `person` nodes that have only `king` ancestors.

As we will see in Section 5.5.2, nodes occurring in Fxgrep patterns can be specified much more precisely as only by their label, by further qualifying them with structural and contextual constraints. By using qualifiers in regular path expressions one significantly exceeds the expressiveness of the ordinary regular path expressions.

5.5.1.2 *Boolean Connectives for Paths*

Conjunctions To specify that the path to a match should simultaneously fulfill several path models, these can be connected via the "`&`" symbol.

▷ The pattern `((//king//)&(//count/duke//)) person` locates `person` elements that have *both* an ancestor `king` and an ancestor `duke` whose father node is an element `count`, as specified by the conjunction of the two connected (incomplete) path patterns `//king//` and `//count/duke//`, respectively.

Disjunctions One can choose between several alternative paths to the match node by connecting them via the "`||`" operator.

▷ The pattern `//((king//)||(count/duke//)) person` locates `person` elements that have either an an ancestor `king` or an ancestor `duke` whose father node is an element `count`.

Negations To specify that the path to a match must not satisfy a path pattern, this must be preceded by the "`!`" symbol.

▷ The pattern `!(//king/king//)person` locates `person`-s who do not have two consecutive `king` ancestors.

5.5.2 *Qualifiers*

Nodes in paths can be qualified with both structural and contextual constraints. The structural constraints talk about the content of a node, while the contextual constraints are concerned with the surroundings of the node.

5.5.2.1 *Structure Qualifiers*

Similarly to XPath, nodes occurring in a pattern can be specified more precisely than by their name by indicating a supplementary condition to be fulfilled by a node provided within brackets following the node. Unlike in XPath, an Fxgrep qualifier is a regular expression to be fulfilled by the children of the subject node, as follows. Rather than a string regular expression, a qualifier is a regular expression over patterns. The children of a node fulfill a pattern regular expression if there is a contiguous sequence of them and an equally long sequence of patterns fulfilling the pattern regular expression, and every pattern in the sequence leads to at least one match when evaluated on the corresponding child. The following examples should clarify the idea.

▷ The pattern `//section[(subsection)+ conclusion]/title` locates `title`-s of `section` elements. The qualifier of the `section` element in the path requires that the sought sections have one or more `subsection`-s (each of them fulfilling the simple pattern `subsection`) followed by a `conclusion` element (fulfilling the simple pattern `conclusion`).

▷ `//section[(subsection/title/"part")+ conclusion]/title`
locates `title`-s of `section`-s having one or more `subsection`-s, the `title` of which contains the substring `"part"` (each of them fulfilling the pattern `subsection/title/"part"`), followed by a `conclusion` element.

Note that any node in a pattern can be qualified, in particular also the nodes occurring in qualifiers.

▷ The pattern `//section[(subsection[(theorem proof)+]/title/"part")+ conclusion]/title` locates the titles of sections as in the previous example, but supplementary requires that the subsections contain a non-empty sequence of `theorem`-s followed by `proof`-s.

Start and end markers The symbols "`^`" and "`$`" can be used to denote the start and the end of a sequence to be matched by a regular expression, both for string and pattern regular expressions.

▷ The pattern `//section[^intro]` locates sections whose first child is an `intro` element. Compare with `//section[intro]` which locates sections which have some `intro` child element.

▷ The pattern `//section[^(theorem proof)+$]` locates sections consisting exclusively of `theorem`-s followed by `proof`-s. Compare with the pattern `//section[(theorem proof)+]` which locates sections containing a sequence of `theorem`-s followed by `proof`-s.

Attribute qualifiers A special form of qualifiers are *attribute qualifiers* by which one can require that an element has an attribute with a specified name and possibly a specified value. An attribute qualifier consists of the symbol "`@`" followed by the specification of an attribute name and possibly succeeded by the symbol "`=`" and the specification of the attribute value. The name as well as the value of the attribute can be specified using regular expressions.

▷ The pattern `subsection[@title="Results"]` identifies subsections having an attribute `title` with value `"Results"`.

Wildcards Often, one needs only to talk about the existence of a node or a sequence of nodes, without further specifying the appearance of the nodes. To denote an arbitrary node or an arbitrary sequence of nodes one can use the symbols "`.`" and "`_`", respectively.

▷ The pattern `//././section` locates sections having at least two ancestors.

▷ The pattern `//.[theorem _ proof]` locates elements which contain a `theorem`, followed by an arbitrary sequence of nodes, followed by a `proof`.

Boolean connectives for structure qualifiers ***Conjunctions:*** A node can have more than one qualifier, each of them being supplied in square brackets following the node. If a node has more than one qualifier than it must fulfill all the conditions defined by them.

▷ The pattern

```
//section[(title/"soups")][(subsection/title/"tomatoes")]
```

identifies sections having the word `"soups"` in the title and a subsection whose title contains the word `"tomatoes"`.

Negations Sometimes it is easier to specify what is disallowed, rather than what model is allowed for a node. A qualifier preceded by the symbol "`!`" specifies a condition which must not be fulfilled by the subject node.

▷ The pattern `//section[!conclusion subsection]` identifies sections that do not have a `conclusion` before a `subsection` element.

5.5.2.2 *Context Qualifiers*

Besides constraints on children, it is possible to specify constraints on siblings of a node, provided the node's father is explicitly denoted in a path. To do so, one specifies two pattern regular expressions l and r which are to be satisfied by the node's left and right siblings, respectively. Such a constraint is called a *context qualifier* and is given between brackets following the node's father in the form `[l#r]`. The symbol "`#`" denotes the subject node, which is the subtree where the path continues. The idea should become clear in the following examples.

▷ The pattern `//section[definition # theorem]/example` locates the `example`-s directly under a `section` element if they are enclosed between a definition and a theorem. The symbol "`#`" denotes the child of the `section` element where the path continues, in this case the `example` element.

▷ The pattern `//section[definition # theorem]//example` locates the `example` elements located arbitrarily deep in a section under an element enclosed by a definition and a theorem. The symbol "`#`" denotes the child of the `section` element where the path continues, in this case the child element of the section which has a descendant `example`.

▷ The pattern `//section[#$]/subsection` locates the `subsection`-s which are the last element in their section.

5.5.3 *Binary Patterns*

Rather than locating individual nodes in the input, we are sometimes interested in identifying tuples of nodes which are in some specified relationship.

A particularly useful case is locating pair of nodes from some specified binary relation.

For example, rather than locating `person` elements which have some ancestor `king` we might be interested in having both the `person` and the corresponding `king` ancestors reported. To specify a binary relation in Fxgrep one must write a (unary) pattern as presented before, in which both elements of the relation are explicitly denoted. The first element of the relation has to be the target node in the pattern (the last node in the top level path). Specifying the second element of the relation is as easy as placing the special symbol, "%", in front of the node denoting the second element of the relation. This should be better understood by looking at the following examples.

▷ The pattern to locate `person` elements which have some ancestor `king` is `//king//person`. To have reported both the `person` and her `king` ancestors we place "%" in front of the `king` node in the pattern The binary pattern for the above query is thus `//%king//person`.

▷ The following unary pattern identifies all book titles whose author's names end in "escu": `//book[(author/"escu$")]/title`. Suppose we want to identify the titles as above, but together with the authors of the books with these titles. The binary query which simultaneously reports the authors having names ending in "escu" and the titles of their books is `//book[(%author/"escu$")]/title`.

Strictly speaking, a match of a binary pattern is a pair. The first node in the pair is called *primary match.* The second node in the pair is a node related to the first node as specified by the "%" symbol and is called *secondary match.* In practice, however, rather than reporting each primary and secondary match separately, if there are more secondary matches related to a same primary match, they are reported at once together with the primary match.

Consider for example the XML input file `library.xml`:

```
<library>
  <book>
    <author>Mihai Eminescu</author>
    <author>Ion Ionescu</author>
    <title>Făt Frumos din tei</title>
  </book>
  <book>
    <author>Oscar Wilde</author>
    <title>A woman of no importance</title>
    <price>10</price>
  </book>
</library>
```

Evaluating the pattern `//book[(%author/"escu$")]/title` on `library.xml` produces:

```
<match>
  <primary>
    <position>[library.xml:5.5]</position>
    <node><title>Făt Frumos din tei</title></node>
  </primary>
  <secondary>
    <position>[library.xml:3.5]</position>
    <node><author>Mihai Eminescu</author></node>
  </secondary>
  <secondary>
    <position>[library.xml:4.5]</position>
    <node><author>Ion Ionescu</author></node>
  </secondary>
</match>
```

It is possible to locate a primary match together with more than one set of related nodes. Each set of related nodes is specified by a "`%`" symbol preceding the corresponding node in the pattern. The sets of secondary matches are in this case reported together with a number denoting to which set of related nodes a match node belongs, as the ordinal number of the corresponding occurrence of the "`%`" symbol in the pattern, given as the value of an `ord` attribute.

For example evaluating the pattern:

```
//book[(%price)?][(%author)]/title["importance"]
```

on `library.xml` delivers the title of books containing the word `"importance"` together with the optional `price`, followed by the `author` of the book:

```
<match>
  <primary>
    <position>[library.xml:9.5]</position>
    <node><title>A woman of no importance</title></node>
  </primary>
  <secondary ord="1">
    <position>[library.xml:10.5]</position>
    <node><price>10</price></node>
  </secondary>
  <secondary ord="2">
    <position>[library.xml:8.5]</position>
    <node><author>Oscar Wilde</author></node>
  </secondary>
</match>
```

5.5.4 *From Patterns to Grammar Queries*

As previously presented, Fxgrep allows the specification of queries also by using a more intuitive pattern language, rather than via grammar queries. Internally, patterns are automatically translated to grammar queries. In the following we show via a few examples how patterns can be automatically translated to grammar queries. A more formal and detailed translation schema for most of the Fxgrep pattern language can be found in [53].

Basically, given a pattern, the corresponding grammar has a nonterminal for each symbol in the pattern denoting some XML node. For example, the pattern `/a/b/c` is translated to the query $(G, \{x_c\})$ where $G = (R, x_a)$, and R is the set productions:

$$\begin{array}{lcl} x_a & \to & \texttt{<a>}\_\,x_b\,\_\texttt{</a>} \\ x_b & \to & \texttt{<b>}\_\,x_c\,\_\texttt{</b>} \\ x_c & \to & \texttt{<c>}\_\texttt{</c>} \end{array}$$

The `a` element denoted by x_a has a child denoted by x_b which is preceded and followed by an arbitrary number of siblings. Given the same grammar G, the query (G, x_b) is equivalent to the pattern `/a/b[c]` and $(G, \{x_a\})$ to `a[b[c]]`.

As presented in Chapter 5.5, binary queries can be specified via binary Fxgrep patterns, by using a symbol "`%`" which may be placed anywhere in front of a node inside a pattern to indicate the secondary match position. Binary patterns are similarly automatically translated into binary grammar queries. The grammar productions are obtained as in the case of unary patterns. The primary target non-terminal is obtained as the non-terminal corresponding to the target node of the unary pattern. The secondary target non-terminal is the non-terminal corresponding to the node preceded by the "%" symbol. For example, the pattern `a/%b/c` is translated to the grammar query $(G, (x_c, x_b))$ with G as above.

Structural constraints for a node are directly reflected in the rule corresponding to the node. For example `/a/b[c+ d*]` is queried by $((R, x_a), \{x_b\})$ with the productions:

$$\begin{array}{lcl}
x_a & \to & \texttt{<a>}\_\,x_b\,\_\texttt{</a>} \\
x_b & \to & \texttt{<b>}\_\,x_c^+ x_d^*\,\_\texttt{</b>} \\
x_c & \to & \texttt{<c>}\_\texttt{</c>} \\
x_d & \to & \texttt{<d>}\_\texttt{</d>}
\end{array}$$

More than one structural constraint for a node is reflected in the grammar by productions with conjunctions of content models. Intuitively, a rule containing a conjunction specifies that each content model in the conjunction has to be fulfilled by a node derived via that production and can be used to simultaneously specify more structural constraints. Note that, in a pattern, a structural constraint is given either explicitly in square brackets following the concerned node, or implicitly as the continuation of the path in which the node occurs. For example `/a[b]/c[d][e]` is matched by $((R, x_a), \{x_c\})$ with the productions:

$$\begin{array}{lcl}
x_a & \to & \texttt{<a>}\_\,x_b\,\_ \wedge \_\,x_c\,\_\texttt{</a>} \\
x_b & \to & \texttt{<b>}\_\texttt{</b>} \\
x_c & \to & \texttt{<c>}\_\,x_d\,\_ \wedge \_\,x_e\,\_\texttt{</c>} \\
x_d & \to & \texttt{<d>}\_\texttt{</d>} \\
x_e & \to & \texttt{<e>}\_\texttt{</e>}
\end{array}$$

When a pattern contains a regular vertical path, the translation is guided by the finite automaton recognizing the regular path. To every state of the automaton for the vertical context we associate a new non-terminal, whereas the transitions correspond to the productions of the grammar accounting for context. An additional non-terminal is associated to final states in order to account for the structure. Consider, e.g., the pattern `(a/)+b` for which the automaton is depicted in Figure 5.14. The corresponding grammar query is $((R, x_1), \{x_3\})$ with the productions:

$$\begin{array}{lll} x_1 & \to & \texttt{<a>}\_x_2 \mid x_3\_\texttt{</a>} \\ x_2 & \to & \texttt{<a>}\_x_2 \mid x_3\_\texttt{</a>} \\ x_3 & \to & \texttt{<b>}\_\texttt{</b>} \end{array}$$

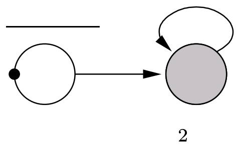


Fig. 5.14 Finite automaton for a vertical regular path.

5.5.5 *Comparison with XPath*

XPath has established itself since its standardization by the W3C Consortium as the most prominent XML query language. It is used both standalone or as part of other important languages like the XML Schema Language [82], XSLT [86, 87] or XQuery [85]. As mentioned, Fxgrep shares with XPath the idea of using paths as a framework for expressing queries. In spite of these syntactic similarities, the two query languages are quite different as we present next.

5.5.5.1 *XPath's Paths*

Like Fxgrep, XPath is conceptually based on an analogy of locating subdocuments in a document with locating files in a directory tree. Nodes can be thus addressed by specifying the path from the root to them. The pattern `/book/sec/title`, as in the case of Fxgrep, locates the `title` elements, the father of which is a `sec` element whose father is the root element labeled `book`.

Fxgrep and XPath have quite different semantic models. XPath is defined in terms of an operational model. An XPath pattern specifies a number of successive selection steps. Each such step selects in turn nodes which find themselves in a specified tree relationship (called *axis* in XPath terminology) with a node selected by the previous step (the *context node*). Initially, the set of selected nodes contains only the root of the input.

The axes can be divided into *forward* and *reverse axes*, depending on whether they select nodes which are after or before the context node in document order. The forward axes are `self`, `child`, `descendant`, `descendant-or-self`, `following` and `following-sibling`. The reverse axes are `parent`, `ancestor`, `ancestor-or-self`, `preceding` and `preceding-sibling`.

The slash symbols in a pattern are thus to be seen as delimiters between the selection steps. Besides specifying a tree relationship, each selection step further specifies a so-called *node test*, i.e., the type of node to be selected (e.g., text node, processing instruction node, element with a specified name). This can be seen as a predicate required to filter the set of selected nodes. Thus, in general, a selection step has the form `treeRelationshipName::nodeTest`. For example `/book/sec/title` is an abbreviation for `/child::book/child::sec/child::title`.

Example 5.27. Consider the XPath pattern:

```
/child::book/descendant::sec/parent::node()/child::text()
```

According to the processing model of XPath this pattern is evaluated on a given input as follows. In the first step, `child::book` selects all the children of the root (i.e. the top-level processing instructions and the root element) and retains from them the element nodes named `book` (i.e. the root element if it has type `book`). Then, `descendant::sec` selects all the descendants of the `book` root element and retains the element descendants named `sec`. Next, the fathers of these `sec` elements are selected and retained whatever node type they have. Finally, the children of these father nodes are selected and those being text nodes are identified by the pattern. In the alternative, abbreviated syntax provided by XPath, the name of the `child` axis can be omitted, "//" stands for `descendant` and "`..`" for `parent`. Hereby, the pattern presented can be also expressed as: `/book//sec/../text()`.

The presence of both forward and reverse axes allows the location steps to arbitrarily navigate in the input. Arbitrary navigation in the input in particular might prevent efficient stream-based implementations as it requires the input tree to be built up in memory. In [67] a set of equivalences are defined which can be used to transform absolute XPath patterns (i.e. whose initial context node for the evaluation is the root node) into equivalent patterns without reverse axes. In general however, XPath patterns are interpreted relative to other nodes selected from the input. In particular, this is the case for XPath select patterns that are used in XSLT and XQuery.

5.5.5.2 *XPath's Qualifiers*

Besides by node tests, the set of nodes selected in an XPath step can be further filtered by specifying a set of *predicates*. The predicates are given between square brackets following the node test. A predicate can be an arbitrary XPath expression. The predicate is evaluated for each of the nodes in the set and only those nodes for which it returns true are retained. In particular, predicates can be as follows.

Arithmetic expressions An XPath qualifier can be an arbitrary arithmetic expression. The qualifier is fulfilled if the node to be filtered is on the position specified by the expression in the input document, when considered in document order on the set of nodes selected by the current step. One simple use case is counting or indexing of matches.

▷ The XPath pattern `//book[42]` locates the 42nd `book` node in the input, in document order.

Patterns An XPath pattern occurring in a predicate denotes for each node in the set of nodes to be filtered, the set of nodes obtained by evaluating the pattern in that node's context. Such a predicate expresses a form of existential quantification as presented below.

- **Static data value comparisons:** If an XPath pattern occurs in a predicate in a comparison with some simple value, then the predicate is true if the denoted set of nodes contains at least one node with that value. We call this kind of comparisons *static data value comparisons* as one term of the comparison is known statically, before the input is read.

  ▷ The XPath pattern `book[author="Kafka"]` identifies `book` nodes having an `author` child, the text value of which is `"Kafka"`.

- **Dynamic data value comparisons:** If the predicate consists of a comparison of two XPath patterns, then the predicate is evaluated to true if there exists a node in the set denoted by the first pattern and a node in the set denoted by the second pattern such that the result of performing the comparison on the string-values of the two nodes is true. We call this kind of comparisons *dynamic data value comparisons* as both terms in the comparison are only known when the XML input document is read.

  ▷ The XPath pattern `//book[author=title]` locates the `book`-s whose proud `author`-s have chosen as `title` their own names.

- **Stand alone patterns:** If the predicate consists only of an XPath pattern, the pattern is evaluated in the context of each node in the set of nodes to be filtered. A node from the set of nodes to be filtered is retained if the set of nodes obtained by evaluating the XPath pattern given as predicate in its context is not empty.

  ▷ The XPath pattern `//sec[subsec]` identifies `sec` elements that have one or more `subsec` children. The pattern `//sec[subsec/theorem]` selects the `sec` elements having a `subsec` child which has a `theorem` child.

5.5.5.3 *Differences in Expressiveness*

XPath is not directly comparable with Fxgrep, that is, there are queries expressible by Fxgrep but not expressible by XPath, and also queries expressible by XPath but not expressible by Fxgrep.

Fxgrep cannot express the non-regular features of XPath, mainly: (1) indexing and counting of matches as possible in XPath via arithmetic expressions as qualifiers, and (2) dynamic data value comparisons.

On the other side, XPath is not able to express most of the regular features of Fxgrep. In XPath, structure qualifiers for a node may only contain one pattern, being thus only able to refer to one child of the node, as opposed to Fxgrep where a structure qualifier may impose a regular condition on a sequence of children.

Regular structural and contextual conditions are in general not expressible in XPath even though some simple regular conditions can be expressed by using, for example, counting of matches.

▷ The Fxgrep pattern `//sec[^subsec*$]`, locating `sec` elements whose children are all `subsec` elements, could be expressed in XPath as `//sec[count(subsec)=count(*)]`, where the qualifier requires that the number of `subsec` children equals the number of all children.

Expressing simple contextual conditions on nodes is possible by using the explicit navigation allowed in XPath and the fact that a step in a path might navigate in arbitrary directions in the input document (e.g. by navigating to the node's left or right siblings and imposing some constraints on them). The explicit navigation however makes the specification more difficult and error-prone.

▷ Locating `theorem` elements which are preceded by a `lemma` and followed by a `corollary` element, achieved in Fxgrep by using the pattern `//.[lemma#corollary]/theorem`, can be performed in XPath by the pattern

```
//theorem[name(preceding-sibling::*[1])="lemma"]
         [name(following-sibling::*[1])="corollary"]
```

Regular contextual conditions are also in general not expressible even though some simple regular conditions can be expressed in a rather cumbersome manner by using counting of matches.

▷ The Fxgrep pattern `//b[^c*#d*$]/a`, locating `a` elements, the father of which is a `b` element, and the left and right siblings of which are all `c` and `d` elements, respectively, can be expressed in XPath as:

```
//b/a[count(preceding-sibling::*)=count(preceding-sibling::c)]
     [count(following-sibling::*)=count(following-sibling::d)]
```

The examples evidence one fundamental conceptual difference between the two pattern languages. Fxgrep patterns specify properties of the nodes to be identified in a declarative way. In contrast, XPath patterns adhere to a rather *operational* style of specification, which basically consists of a succession of navigation steps.

Furthermore, no regular path expressions can be expressed in XPath. The only kind of deep-matching allowed by XPath is "`//`" (arbitrary descendant). Also, XPath can only locate individual nodes in the input as opposed to the binary patterns of Fxgrep.

5.5.6 *Bibliographic Notes*

The research interest in developing query languages for hierarchically structured data has been very vivid since the introduction of XML. Technically speaking, many approaches to querying have been proposed, using different formalisms, for example logic-, automata- or grammar-based, as presented in Section 5.4.7. Nevertheless, most of these formalisms are too complex to be directly usable by non-specialist users. Instead, given the spreading of XML in different application domains, an XML query language has to be concise and easy to learn by providing a small number of constructs, while being able to fulfill the various domain-specific requirements. We refer to such a language as a *pattern-language* as opposed to more sophisticated, yet for the non-specialist less understandable languages. Typically, the XML query tools would provide a pattern language and automatically translate the patterns into the internally used querying setting, as Fxgrep does.

Most of the proposals made are targeted at the XPath pattern language, which became the de facto industry standard XML query language. Many of them are generally able to implement different subsets of XPath. Most of them are subsumed by an XPath fragment called *Core XPath* [30], mainly featuring location paths and predicates using location paths but not arithmetics and data value comparisons. Some of the research works extend XPath with simple regular path expressions [1, 20, 21, 26, 32].

Even though formalisms for expressing queries similarly powerful with the grammar formalism underlying Fxgrep exist (see Section 5.4.7), to the best of our knowledge, no other pattern language has been provided which allows one to express in a concise and declarative way the regular and contextual constraints as in the pattern language of Fxgrep.

The pattern language of Fxgrep was originally designed by Neumann and Seidl [53] and contained simple paths with deep matching, structure qualifiers with boolean conectives and context qualifiers as presented above. We extended the Fxgrep pattern language in order to test the practical suitability of the concepts presented in this work by regular path expressions, regular expressions for element and attribute names, boolean connectives for paths as well as the possibility of expressing binary queries.

The task of searching tree-structured documents was present long before the arrival of XML in application-domains such as linguistics. Linguistic queries are typically performed on collections of natural language texts manually annotated with syntactical information, called *corpora*. The corpora are conceptually labeled trees, similar to XML documents, and can also be encoded in some XML format. The linguistic queries typically identify syntactical constructs by specifying tree relationships, similarly to XPath, or for this matter Fxgrep.

One of the most popular query-tools for linguistic corpora are *tgrep* [69], published as early as 1992, long before the appearance of XPath, and its successor *tgrep2* [70]. tgrep patterns are build together from dominance and precedence relationships among nodes, being thus similar to XPath, but using a different syntax. Finding matches of patterns requires a preprocessing phase in which the corpus to be searched is annotated with index information. Like other proposed linguistic query languages, tgrep is tied to the specific data format of the searched document and cannot be used for general tree-structured documents.

A natural choice is to represent linguistic data in XML and use general-purpose XML query languages like XPath or Fxgrep to search for linguistic patterns. XPath lacks however some features needed for linguistic queries, where not only vertical but also precise horizontal relationships among nodes need to be specified. For example, linguitic queries often need to refer to *immediately following* nodes [11], i.e. nodes on the path from the following sibling to its leftmost descendant. One solution to this problem is extending XPath with more axes for horizontal navigation derived from a basic *immediately following* axis, as in *LPath* [11]. The immediately-following relationship can be easily expressed in the underlying grammar formalism of Fxgrep. Given its ability to specify accurate horizontal and vertical constraints , Fxgrep is suitable for application domains like linguistics where such precise contextual specifications are needed.

Since web pages are in fact document trees, querying hierarchically structured documents is a natural enhancement of plain keyword based search on the Web. On the one hand this allows the identification of interesting portions of documents rather than whole documents. On the other hand even when whole documents are to be identified, the specification of the documents of interest can be done more precise by requiring each keyword in a query to occur in a specific (structural) context. Most of the proposed query languages [80] are subsumed by the grammar queries and Fxgrep for this matter. Rather than one document at a time as considered by us here, the web search use case requires to simultaneously consider multiple documents. An important issue thereby is to rank the documents w.r.t. the query at hand. A survey of some of the proposed structured query languages for web search can be found in [80].

5.6 Online Querying

XML processing can be classified into two main categories, which correspond to the two main approaches to XML parsing, DOM [22] and SAX [71]. In the first approach, used by most existing XML processors, the tree which is textually represented by the XML input is effectively constructed in memory and subsequently used by the XML application.

In the second approach, the XML input is transformed into a stream of events which are transmitted to a listening application. An event contains a small piece of information linearly read from the input, e.g. a *start-tag* or an *end-tag*. The order of the events in the stream corresponds to the document order of the input, i.e. to the sequential order in which the information is read from the input. It is up to the listening application to decide how it processes the stream of events. In particular, it can construct the XML tree in memory and subsequently process it, as in the first approach, being thus at least as expressive.

The advantage of the event-based approach is that it allows one to buffer only the relevant parts of the input, thereby saving time and memory. The increased flexibility allows the handling of very large documents, the size of which would be prohibitive if the XML input tree was to be entirely built in memory. Also, the event-based processing naturally captures real-life applications in which the document is received linearly via some communication channel, rather than being completely available in advance.

The research interest in querying XML streams has been very vivid recently and there is a very rich literature on this topic. The related work is reviewed in Section 5.6.3. The proposed query languages are generally able to implement different subsets of XPath. Most of them are subsumed by *Core XPath* .

In this section we present a solution for efficient event-based evaluation of queries which go beyond the capabilities of many languages for which this problem was previously addressed. Most of these languages can be expressed using first-order logic (FO) possibly extended with regular expressions on vertical paths, but are less expressive than monadic second order logic (MSO). In contrast, our solution evaluates *grammar queries*, which are equivalent to MSO queries as mentioned in Section 5.4.2.

Grammar queries can be implemented using pushdown forest automata as presented in Section 5.4.3. The original construction as introduced by Neumann and Seidl [55] generally requires the construction of the whole input tree in memory and the execution of two traversals of it. A one-pass query evaluation, suitable for an event-based implementation, is addressed only for a restricted class of queries. These are the so called *right-ignoring* queries for which all the information needed to decide whether a node is a match has been seen by the time the end-tag of the node is encountered. The term right-ignoring is coined by the fact that all the nodes to the right of the match node in the tree representation are irrelevant for the match.

Rather than a-priori (i.e. statically) handling only a restricted subset of queries, we show in this section how *arbitrary* grammar queries can be evaluated on XML streams using pushdown forest automata.

Example 5.28. Consider an XML document, the tree representation of

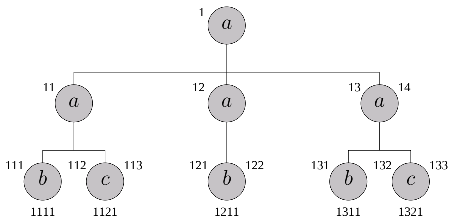


Fig. 5.15 Input tree.

which is depicted in Figure 5.15. Each location in the tree corresponds to an event in the corresponding stream of events. The stream of events together with the corresponding locations are denoted below. Nodes too can be identified by the location corresponding to their start-tag.

```
 1
<a>
        11  111 1111  112 1121  113
        <a> <b> </b>  <c> </c> </a>

        12  121 1211  122
        <a> <b> </b> </a>

        13  131 1311  132 1321  133
        <a> <b> </b>  <c> </c> </a>
 14
</a>
```

It should be clear that the amount of memory necessary to answer an arbitrary query inherently depends on the query and on the input document at hand. Consider for example the (XPath or Fxgrep) pattern `//a/b` locating `b` nodes which have as father an `a` node. The node 111 is a match in our input. This can be detected as early as at the location 111, as the events following 111 cannot change the fact of 111 being a match.

The pattern `//a[c]/b` locates `b` nodes which have a node `a` as father and a `c` sibling. The node 111 is again a match but this becomes clear only after seeing that the `a` parent has also a child `c` at location 112. One has thus to remember 111 as a potential match between the events 111 and 112. As the events to the right of 112 cannot change the fact of 111 being a match, 111 can be reported and discarded at 112.

Finally, as an extreme case consider the (MSO expressible) XPath pattern `/*[not(d)]//*` locating all descendant nodes of the root element if this has no child node `d`. Any node in the input is a potential match until seeing the last child of the root element. In our example all nodes have to be remembered as potential matches up to the last event 14. Note thus, that any algorithm evaluating a query needs in the worst case an amount of space linear in the input size. However, most of the practical queries require a quite small amount of memory as compared to the size of the input.

This section addresses the following issues.

- We introduce a way of defining the earliest detection location of a match for some given query and input tree.
- We prove that matches of grammar queries are recognizable at their earliest detection location and hereby demonstrate the following theorem:

  Theorem 5.29. *Matches of MSO definable queries are recognizable at their earliest detection location.*

- Based on the construction used for proving Theorem 5.29 we give an efficient algorithm for grammar query evaluation, which reports matches at their earliest detection point. As a consequence potential matches are remembered only as long as necessary, meaning that our construction implicitly adapts its memory consumption to the strict requirements of the query on the input at hand.

This section is organized as follows. In section Section 5.6.1 we briefly present how a pushdown automaton can be used to answer right-ignoring queries on streams. The generalized query evaluation for XML streams is given in Section 5.6.2 where the algorithm is presented and its correctness, optimality, complexity and performance are addressed. Related work is discussed in Section 5.6.3.

In the following let $Q = (G, T)$ be an arbitrary query on input f_1 with $G = (\Sigma, X, R, r_0)$. Further, let $A_{\vec{G}}$ be the LPA accepting $\mathcal{L}_G$ constructed as in Section 5.3.2.2.

5.6.1 *Right-ignoring Queries*

In this section we briefly present the ideas [53, 55] which allow the evaluation of a right-ignoring query $Q = (G, T)$ using the $A_{\vec{G}}$ LPA (defined in Section 5.3.2.2 on page 227). Let us investigate what are the requirements for Q to be right-ignoring. Consider a match node π of Q as depicted in

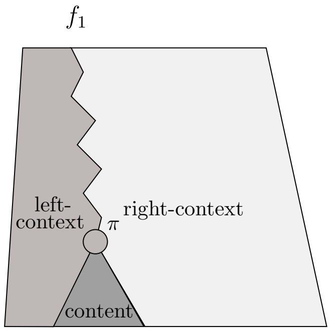


Fig. 5.16 The context and the content of a match.

Figure 5.16. Since the query is right-ignoring, all the nodes from the right-context are irrelevant for the decision as to whether π is a match. That is, π is a match however the right-siblings of π and of every ancestor of π might look like.

Let us consider that π is the k-th out of m siblings, with $m \geq k$. Since π is a match, according to the definition, there is a derivation labeling the sequence of siblings containing π with $x_1 \dots x_k \dots x_m$ and $x_k \in T$. There is thus a content model r_j s.t. $x_1 \dots x_k \dots x_m \in [\![r_j]\!]$. The fact that the right siblings of π might be any trees implies that x_i must be able to derive any tree, i.e. $[\![R]\!]\, x_i = \mathcal{T}_X$ for all $i = k+1, \dots, m$. Also, as the number of right-siblings might be arbitrary, all of the above must hold for all $m \in \mathbb{N}$ with $m \geq k$.

To ensure the above there must exist an NFA state $y_k \in Y_j$ reached after seeing the left siblings of π with $y_k \in F_j$ and s.t. for all $p \in \mathbb{N}$, there are $y_{k+1}, \dots, y_p \in F_j$ with $(y_i, x_\top, y_{i+1}) \in \delta_j$ for $i = k+1, \dots, p$, where $x_\top \in X$ and $[\![R]\!]\, x_\top = \mathcal{T}_\Sigma$. We call such a y_k a *right-ignoring NFA state.* The non-terminal $x_\top$ is to be seen as a wild-card non-terminal which can derive any tree and which is made available in any forest grammar. The necessity of the above follows from the fact that no other non-terminal x in the grammar can be s.t. $[\![R]\!]\, x = \mathcal{T}_\Sigma$, as in general the alphabet Σ is neither finite nor known in advance[5].

Given an NFA state y, we use the predicate $rightIgn(y)$ to test whether y is a right ignoring state. Testing whether $rightIgn(y)$ holds, can be done statically by checking in the NFA whether there are cycles visiting y and

[5]We do not consider optimizations possible when the schema of the XML data is available.

consisting only of $x_\top$ edges, needing thus time linear in the size of the NFA.

Similar considerations have to be made due to the right-ignorance for all the nodes lying on the path from the root to π. Therefore we need to consider all the non-terminals with which a derivation defining a match may label the nodes lying on the path from the match to the root. These are the so-called *match-relevant* non-terminals, defined by:

$$x \text{ is match-relevant iff } x \in T \text{ or } \begin{array}{l} x \to a\langle r_j \rangle, (y_1, x_1, y) \in \delta_j \\ \text{and } x_1 \text{ is match-relevant} \end{array}$$

We call a query Q *right-ignoring* iff all $y \in Y$ with $(y_1, x, y) \in \delta$ and x match-relevant are right-ignoring. As presented above, testing whether a query is right ignoring can be done completely statically.

The right-ignorance of Q ensures thus that if the left-context of a match is fulfilled, then the right-context is also always satisfied. Hence, to check whether a node is a match, it suffices to look into the forest state in which $A_{\vec{G}}$ leaves a node, which synthesizes the information gained after visiting the left-context and the content of the node, depicted in dark grey in Figure 5.16:

Proposition 5.30. *Let q_π be the forest state in which $A_{\vec{G}}$ leaves a node π. If Q is right-ignoring then $\pi \in \mathcal{M}_{Q,f}$ iff $y \in q_\pi$, $(y_1, x, y) \in \delta$ for some $y, y_1 \in Y$ and $x \in T$.*

Proof. The theorem is proven as Theorem 7.4 in [53]. □

To answer queries on XML streams without building the document tree in memory, it remains to show how a left-to-right pushdown automaton can be implemented in an event-based manner.

Event-driven Runs of Pushdown Forest Automata

Consider an LPA $A_{\vec{G}} = (2^X, 2^Y, \{q_0\}, F, \mathit{Down}, \mathit{Up}, \mathit{Side})$ as defined in Section 5.3.2.1 and its processing model as depicted in Figure 5.6 (on page 226). The order in which $A_{\vec{G}}$ visits the nodes of the input is the order of a depth-first, left-to-right search, which corresponds exactly to the document-order.

Compare Figures 5.6 and 5.17. At every node π, $A_{\vec{G}}$ executes one *Down* transition at the moment when it proceeds to the content of π and one *Up* followed by one *Side* transitions at the moment when it finishes visiting the content of π. These moments correspond to the start and end tags, respectively, of the node π. The algorithm implementing the event-driven run of $A_{\vec{G}}$ is depicted in Figure 5.18.

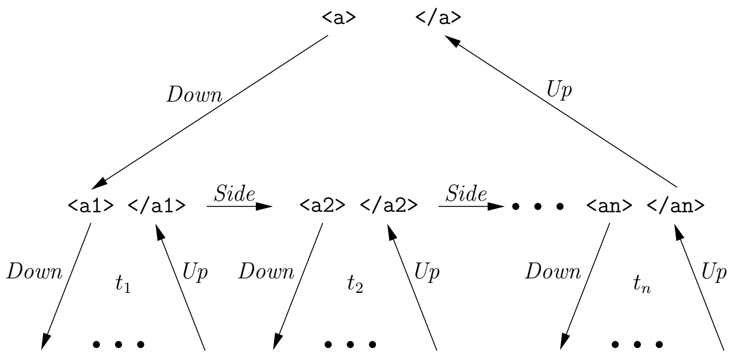


Fig. 5.17 Event-driven run of a pushdown forest automaton.

We handle the following events:

(1) `startDoc`, which is triggered before starting reading the stream;
(2) `endDoc`, which is triggered after finishing reading the stream;
(3) `enterNode`, which is triggered when a start-tag is read;
(4) `leaveNode`, which is triggered when an end-tag is read.

The stack declared in line 1 is needed in order to remember the forest states used for the traversal of the content of the elements opened and not yet closed. The variable q declared in line 2 stores the current forest state during the traversal of the document.

At the beginning, `startDocHandler` is called and it sets the current state to the initial state of the automaton (line 5). A start tag triggers a call of `enterNodeHandler` which remembers the current state on the stack (line 9) and updates the current state as result of executing the *Down* transition (line 10). An end-tag triggers the corresponding *Up* transition (line 14), followed by the *Side* transition which uses as forest state the last remembered state on the stack, i.e. the forest state before entering the element now ending (line 15).

The number of elements on the stack always equals the depth of the XML element currently handled. Hence the maximal height of the stack is the maximal depth of the handled XML document, which is in general rather small, even for very large documents.

Depending on the purpose of the pushdown automaton, other actions can be performed in the events handler. For the purpose of validation, it must be checked at the end of the document whether the current state is a final state (line 18).

```
Stack s;
ForestState q;

startDocHandler(){
  $q$ := $q_0$;
}

enterNodeHandler(Label a){
  s.push($q$);
  $q$ := $Down(q, \mathrm{a})$ ;
}

leaveNodeHandler(Label a){
  TreeState $p$ = $Up(q, \mathrm{a})$ ;
  $q$ := $Side(\mathrm{s.pop()}, \mathrm{a})$ ;
}

endDocHandler(){
  if $q \in F$ then output("Input accepted.")
  else output("Input rejected.");
}
```

Fig. 5.18 Skeleton for the event-driven run of a pushdown forest automaton.

For the purpose of answering right-ignoring queries it must be checked whether the forest state obtained after the side transition has the property stated in Proposition 5.30. Using the above presented implementation, $A_G^{\rightarrow}$ is thus able to answer right-ignoring queries on XML streams.

5.6.2 *Arbitrary Queries*

The previous section only shows how right-ignoring queries can be answered on XML streams. In this section we lift this limitation by showing how *arbitrary* queries can be answered on XML streams.

In the case of non right-ignoring queries, the decision as to whether a node is a match cannot be taken locally, i.e. at the time the node is left, because there is still match-relevant information in the part of the input not yet visited. The decision can only be taken after seeing all of the match-relevant information.

The general situation is depicted schematically in f_1 in Figure 5.19 (i). The node π is a potential match considering its left context and its content

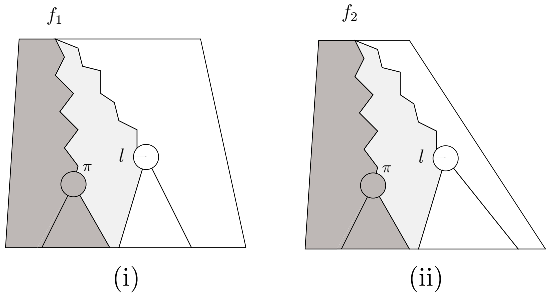


Fig. 5.19 Right completion of a forest.

(depicted in dark grey) which can be checked by the time the end-tag of the node is seen. The decision as to whether this is indeed a match node must be postponed until seeing the relevant part of the right context (depicted in light grey), which was empty in the particular case of right-ignoring queries. The location which must be reached in order to recognize π as a match is denoted as l. We call such a location l, *earliest detection location* of the match π, formally defined below.

Earliest Detection Locations

A forest f_2 is a *right-completion* of a forest f_1 at location $l \in L(f_1)$ iff f_1 and f_2 consists of the same events until l. (The tree representation of f_1 and f_2 are depicted in Figure 5.19). Formally:

$$f_2 \in \mathcal{R}ightCompl_{f_1,l} \text{ iff } prec_{f_1}(l) = prec_{f_2}(l) \text{ and } lab(f_2[\pi']) = lab(f_1[\pi']) \text{ for all } \pi' \in prec_{f_1}(l).$$

with $prec_f(l)$ denoting the *preceding nodes* of a location $l \in L(f)$ in a forest f, defined as $prec_f(l) = \{\pi \mid \pi \in N(f), \pi < l\}$, where "$<$" denotes lexicographical comparison.

A location l is an *early detection location* of a match node π for a query Q in input f_1 iff $\pi \in \mathcal{M}_{Q,f_2}$ for all right-completions f_2 of f_1 at l.

A location l is the *earliest detection location* of a match node π iff l is the smallest early detection location of π in lexicographic order.

Example 5.31. Reconsider Example 5.28 and the accompanying input depicted in Figure 5.15 (on page 277). Given the query `//a/b`, the earliest detection location of node 111 is 111. As for the query `//a[c]/b` the earliest detection location of node 111 is 112. Finally, for the query `/*[not(d)]//*`, there is no early detection location for any of the match nodes. This matches cannot be detected until the last location in the input has been reached.

5.6.2.1 *Idea*

We proceed now to the description of the computation performed by our algorithm for evaluating grammar queries on XML streams. This can be seen as a run of a pushdown automaton changing its state on every XML event.

For the purpose of evaluation we use the stack to remember the following information for a location l at some nesting level :

(1) q, denoting the progress within the content models to be considered on the level containing l. This is exactly the forest state in which $A_G^{\rightarrow}$ (the DLPA accepting $\mathcal{L}_G$) reaches l;
(2) ri, needed for the early detection of matches as presented below;
(3) m, storing potential matches which might be confirmed on the current level, as well as the potential matches accumulated while traversing the current level up to l.

For 1, we remember the states of the finite automata corresponding to the content models which are reached considering the content seen so far on the current level. They are obtained as by performing the transitions of $A_G^{\rightarrow}$.

For 2, we need to know which of the content models considered on the current level occur in right-ignoring contexts. A content model for an element e occurs in a right ignoring context iff there is no content model of an enclosing element whose fulfillment depends on the right context of e. We call such content models *right-ignoring content models.*

For 3, we associate potential matches with NFA states from q. A potential match is associated with a state y at location l iff the match may be defined w.r.t. derivations in which the word of non-terminals on the current level is accepted by the NFA run reaching l in state y. The information m can be thus represented as a partial mapping from NFA states y to the corresponding potential matches $m(y)$.

Consider our query $Q = (G, T)$ with a forest grammar $G = (R, r_0)$. Let $r_1, \ldots, r_p$ be the regular expressions occurring on the right-hand sides in the productions R, where p is the number of productions. For $0 \leq j \leq p$, let $A_j = (Y_j, y_{0,j}, F_j, \delta_j)$ be the non-deterministic finite automaton (NFA) accepting the regular string language defined by r_j as obtained by the

Berry-Sethi construction. By possibly renaming the NFA states we can always ensure that $Y_i \cap Y_j = \varnothing$ for $i \neq j$. Let $Y = Y_0 \cup \cdots \cup Y_p$ and $\delta = \delta_0 \cup \cdots \cup \delta_p$.

Initial State Initially, we start with the NFA start state $y_{0,0}$ of the start content model r_0. The content model r_0 is right-ignoring as there are no enclosing elements. Also, there are no potential matches detected yet, thus the information initially remembered on the stack consists of:

$$q_0 = \{y_{0,0}\},\ ri_0 = \{r_0\},\ m_0 = \varnothing$$

Start-Tag Transitions On a start-tag event `<a>` at location l, new information (q_1, ri_1, m_1) is pushed on the stack, depending on the information in the current top of the stack (q, ri, m) as follows.

The possible content models of the current element are computed from the content models in which the element may occur (as in the case of a *Down* transition in $A_{\vec{G}}$). Before seeing any of the children of the current element we are in the initial NFA state of these content models:

$$q_1 = \{y_{0,j} \mid y \in q,\ (y, x, y_1) \in \delta,\ x \to a\langle r_j \rangle\}$$

A content model r_j considered for the current element l is right-ignoring if (1) the surrounding content model r_k is right ignoring and (2) r_k is fulfilled independently of how the right siblings of l might look like. Condition (1) can be looked up in ri. To ensure (2) there must be an right-ignoring NFA state y_1 reachable after seeing the left siblings of l. Thus:

$$ri_1 = \{r_j \mid y \in q,\ (y, x, y_1) \in \delta_k,\ x \to a\langle r_j \rangle, r_k \in ri, rightIgn(y_1)\}$$

As for the potential matches which might be confirmed while visiting the content of l, these are the matches propagated so far for which the content of the current level is fulfilled whatever follows after l. We add l to these potential matches if it can be derived from a target non-terminal considering its left-context, and its right-context is irrelevant (that is, l's confirmation as a match depends thus only on its content).

$$\begin{aligned} m_1(y_{0,j}) = & \bigcup \{m(y) \mid y \in q,\ (y, x, y_1) \in \delta_k,\ x \to a\langle r_j \rangle, r_k \in ri, rightIgn(y_1)\} \\ & \cup \\ & \{l \mid y \in q,\ (y, x, y_1) \in \delta_k,\ x \to a\langle r_j \rangle, r_k \in ri, rightIgn(y_1), x \in T\} \end{aligned} \tag{5.1}$$

End-Tag Transitions An end-tag event `</a>` at location $\pi i(n{+}1)$ signals that the processing of the sequence of children $\pi i1, \ldots, \pi in$ is completed and the computation has to return to the nesting level and advance over the father node πi. The top two elements on the stack at this moment: (q, ri, m) and (q_1, ri_1, m_1) store the state of the computation after seeing the children and the left siblings of πi, respectively. (q, ri, m) and (q_1, ri_1, m_1) are consumed from the stack and used to compute the new top of the stack (q_2, ri_2, m_2), reflecting the state after finishing seeing πi, as follows.

A content model r_j is fulfilled by the children of πi iff there is some $y_2 \in q \cap F_j$, i.e. a NFA final state for r_j is reached after traversing them. It follows that πi can be derived from symbols x for which there is a production $x \to a\langle r_j\rangle$. The advance in the content models on the level of πi, after seeing πi is obtained by considering NFA transitions with symbols x from which πi may be derived. This is completely similar to an *Up* transition followed by a *Side* transition in $A_G^{\rightarrow}$ and is summarized by:

$$q_2 = \{y_1 \mid y_2 \in q \cap F_j, x \to a\langle r_j\rangle, y \in q_1, (y, x, y_1) \in \delta\}$$

As the set of right ignoring content models only depends on the surrounding content models, it remains unchanged for a whole nesting level, that is :

$$ri_2 = ri_1$$

As for the potential matches, we have to aggregate the potential matches from the left-context of πi with those from its content. More precisely, potential matches defined by an NFA run on the children level are joined with potential matches from the left context associated with NFA states which are reached in nesting NFA runs after seeing the father node. The father node, πi is added as a potential match if it can be derived from a target non-terminal:

$$\begin{aligned} m_2(y_1) = \{m(y_2) \cup m_1(y) \mid y_2 \in q \cap F_j, x \to a\langle r_j\rangle, y \in q_1, (y, x, y_1) \in \delta\} \cup \\ \{\pi i \mid y_2 \in q \cap F_j, x \to a\langle r_j\rangle, y \in q_1, (y, x, y_1) \in \delta, x \in T\} \end{aligned} \tag{5.2}$$

5.6.2.2 *Recognizing Matches*

The construction above allows the location of matches as stated by the following theorem:

Theorem 5.32. *A location l is an early detection location for a match node π iff $\pi \in m(y)$, $r_j \in ri$ and $rightIgn(y)$ for some $y \in q \cap Y_j$ with (q, m, ri) being the top of the stack at event l.*

Proof. The complete proof is given in Section 5.8.3. The idea of the proof is described next.

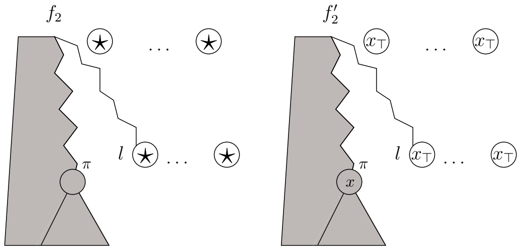


Fig. 5.20 Right completion at l and corresponding derivation.

Let l be an early detection location for π. Let f_2 be a right-completion obtained from f_1 by adding on every level from the root to l an arbitrary number of right siblings $\star\langle\rangle$ (as depicted in Figure 5.20), where $\star$ is a symbol not occurring in any of the rules in the grammar. By the definition of early detection locations there is a derivation f_2' of f_2 in which π is labeled x for some $x \in T$. Also, since $\star$ does not occur in any rule, f_2' must label all the $\star$ nodes with $x_\top$. The y with the properties as required by this theorem is the NFA state in which the location l is reached within the NFA accepting run corresponding to f_2'.

Conversely, let (q, m, ri) be the top of the stack at event l and let $\pi \in m(y)$, $r_j \in ri$ and $rightIgn(y)$ for some $y \in q \cap Y_j$. From $\pi \in m(y)$ it follows that there is a relabeling of the nodes visited so far in which π is labeled with some $x \in T$ and which might be completed to a whole derivation according to the grammar G using $x_\top$ symbols for the not yet visited nodes. The existence of the completion on the current level follows from $rightIgn(y)$, while the existence of the completions on the enclosing levels is ensured by the condition $r_j \in ri$. □

As locations are visited in lexicographic order, testing the condition in Theorem 5.32 ensures that every match π is detected when reaching its earliest detection location. This proves Theorem 5.29.

The algorithm implementing the event-driven evaluation of the queries as above is given in Figure 5.21 (on page 289). We assume that `enterNodeHandler` and `leaveNodeHandler` receive as an argument, besides the label of the currently read node, also the currently reached location. For the case in which the current location is not provided by the event-based parser, note that it can be easily propagated along the event handlers in the internal parse-state. The algorithm basically follows the computation rules given above while sharing the commonalities in the rules for q, ri and m. As an abbreviation we use the operator $\oplus$ to add a new entry or update an existing set entry in a mapping m via set union.

Matches are detected when their earliest detection location is reached, i.e. at the event-handler executed at the immediately preceding location. This might be the case either at start or at end tags. At start tags (line 12) we report potential matches for which we know that (a) the right siblings at the current level are irrelevant (condition $rightIgn(y_1)$ tested in line 9); (b) the right siblings of the ancestors are irrelevant (condition $r_k \in ri$ tested in line 9) and (c) the content is irrelevant (condition $rightIgn(y_{0,j})$ in line 11).

At end tags (line 32) we report potential matches for which the content was fulfilled (condition $y_2 \in F_j$ in line 30) and the upper-right context is irrelevant (condition $r_j \in ri$).

Note that there is no need to propagate a confirmed match π beyond its earliest detection location l where it is reported. (see tests in lines 11 and 32).

Also, potential matches are discarded implicitly precisely as soon as enough information is seen in order to reject them. Potential matches $m(y)$ at a location l are no longer propagated when y is not involved in the NFA transitions. This happens at end tag events if there is no transition (y, x, y_1) in any of the possible content models. Also at end tag events, potential matches in $m(y)$ are discarded if y is not a final state in any of the considered content-models on the finished level. Thereby matches are remembered only as long as the strictly necessary portion of the input has been seen in order to confirm them.

Finally, at the end of the input potential matches not yet confirmed and conforming to the top-level content model (condition $y \in q \cap F_0$ in line 48) are reported as matches in line 49.

5.6.2.3 *Complexity*

Let $|D|$ be the size of the input data, i.e. the number of nodes in it. The size of a query Q can be estimated as the number of NFA states $|Y|$ plus the number of non-terminals $|X|$. Let pot_{max} be the maximum number of potential match nodes at any given time during the traversal.

For every node in the input `enterNodeHandler` and `leaveNodeHandler` is called once. In `enterNodeHandler` at πi, the loop starting at line 7 is executed for every $y \in q$, for every outgoing NFA transition (y, x, y_1) and

```
Stack s;

enterNodeHandler(Location $l$, Label a){
  ($q$,$ri$,$m$) := s.top();
  $q_1$ := $ri_1$ := $m_1$ := $\varnothing$;

  forall $y \in q$ with $(y, x, y_1) \in \delta_k$ and $x \to a\langle r_j \rangle$
    $q_1$ := $q_1 \cup \{y_{0,j}\}$
    if $rightIgn(y_1)$ and $r_k \in ri$ then
      $ri_1$ := $ri_1 \cup \{r_j\}$;
      if $rightIgn(y_{0,j})$ then
        reportMatches($m(y)$);
        if $x \in T$ then reportMatches($\{l\}$) endif
      else
        $m_1$ := $m_1 \oplus \{y_{0,j} \mapsto m(y)\}$;
        if $x \in T$ then $m_1$ := $m_1 \oplus \{y_{0,j} \mapsto \{l\}\}$ endif
      endif
    endif
  endfor

  s.push(($q_1$,$ri_1$,$m_1$));
}

leaveNodeHandler(Location $ln$, Label a){
  ($q$,$ri$,$m$) := s.pop();
  ($q_1$,$ri_1$,$m_1$) := s.pop();
  $q_2$ := $m_2$ := $\varnothing$;
  $ri_2$ := $ri_1$;

  forall $y \in q_1$,$y_2 \in q$,$y_2 \in F_j$,$x \to a\langle r_j \rangle$ and $(y, x, y_1) \in \delta_k$
    $q_2$ := $q_2 \oplus \{y_1\}$;
    if $r_j \in ri$ then reportMatches($m(y_2)$)
    else
      $m_2$ := $m_2 \oplus \{y_1 \mapsto m(y_2)\}$;
      $m_2$ := $m_2 \oplus \{y_1 \mapsto m(y)\}$;
      if $x \in T$ then $m_2$ := $m_2 \oplus \{y_1 \mapsto \{l\}\}$ endif
    endif
  endfor

  s.push(($q_2$,$ri_2$,$m_2$));
}

startDocHandler(){s.push(($q_0$,$\{r_0\}$,$\varnothing$));}

endDocHandler(){
  ($q$,$ri$,$m$) := s.pop();

  forall $y \in q \cap F_0$
    reportMatches($m(y)$);
  endfor

}
```

Fig. 5.21 Algorithm for event-driven query answering.

for all content models r_j for x. The size of q is in $O(|q_{max}|)$, where q_{max} is the forest state q with the maximum number of elements. Let cm_{max} be the maximum number of content models considered on a level and let

out_{max} be the maximum number of outgoing NFA transitions from an NFA state. The loop is executed thus up to $|q_{max}| \cdot out_{max} \cdot cm_{max}$ times.

The set union in line 8 can be computed in time $O(|q_{max}|)$. The set union in line 10 needs time $O(cm_{max})$. Reporting the confirmed matches additionally requires time $O(pot_{max})$. Finally the set unions in lines 15 and 16 necessitate again $O(pot_{max})$ time. A call to `enterNodeHandler` amounts thus to $O(|q_{max}| \cdot out_{max} \cdot cm_{max} \cdot (|q_{max}| + cm_{max} + pot_{max}))$ time.

In `leaveNodeHandler`, the loop starting at line 30 is executed in the worst case, for every $y \in q_1$ and every $y_2 \in q$, i.e. up to $|q_{max}|^2$ times. The set union in line 31 is computed in time $O(|q_{max}|)$. Reporting the confirmed matches possibly adds $O(pot_{max})$ time. The set unions in lines 34, 35 and 36 need $O(pot_{max})$ time. A call to `leaveNodeHandler` amounts thus to $O(|q_{max}|^2 \cdot (|q_{max}| + pot_{max}))$ time.

As `leaveNodeHandler` and `enterNodeHandler` are called each once for every node, the overall time complexity of event driven evaluation of queries is thus in $O(|D| \cdot (|q_{max}| \cdot out_{max} \cdot cm_{max} \cdot (|q_{max}| + cm_{max} + pot_{max}) + |q_{max}|^2 \cdot (|q_{max}| + pot_{max})))$. The values of $|q_{max}|$, out_{max} and cm_{max} are bounded by values which do not depend on $|D|$. Experimental evidence show them to be small, and correspondingly the algorithm scales well with the size of the query as presented in the next section.

The worst complexity in the size of the document is obtained for $pot_{max} = |D|$, in the case where all the nodes are potential matches until the end of the document. In general, however, the number of potential matches is much less than the total number of nodes ($pot_{max} \ll |D|$) and can be assimilated with a constant. In this case we obtain a time linear in the size of the document, as suggested by experimental results [6].

As for the space complexity, let d be the depth of the input document. During the scan of the document we store at each location the (q, ri, m) tuples for all ancestor locations up to the root, which correspond to the opened and not yet closed elements at the current location. For every level, q has up to $|q_{max}|$ elements, m stores up to $|q_{max}| \cdot pot_{max}$ locations and ri up to cm_{max} content models. As all these elements can be stored in constant space and the height of the stack is at most d, we obtain the worst case space complexity $O(d \cdot (|q_{max}| + |q_{max}| \cdot pot_{max} + cm_{max}))$. Most of the practical queries need only a small amount of memory, as the information relevant to whether a node is a match is typically located in the relative proximity of the node (that is pot_{max} is small).

5.6.3 *Bibliographical Notes*

A basic task in XML processing is XML validation. The problem of validating XML streams is addressed by Segoufin and Vianu in [74] and Chitic and Rosu in [18]. As mentioned in Section 5.3.1.3, XML schema languages are basically regular forest languages, hence conformance to such a schema can be checked by a pushdown forest automaton. As presented in this chapter this can be performed efficiently on XML streams in the event-based manner.

Many research works deal with querying of XML streams. Most of them consider subsets of XPath. Some of them deal with XQuery, which in fact is more than a querying language as it allows the transformation of the input. In the following we are mainly interested in the querying capabilities of the considered languages.

Conventional attribute grammars (AG) and compositions thereof are proposed by Nakano and Nishimura in [52] as a means of specifying tree transformations. An algorithm is presented which allows an event-driven evaluation of attribute values. Specifying transformations, or in particular queries, using AG is however quite elaborate even for simple context-dependent queries and AG are restricted to use attributes of non-terminal symbols at most once in a rule. Also as no stack is used input trees have to be restricted to a maximum nesting depth.

More suited for XML are attribute grammars based on forest grammars as considered in XML Stream Attribute Grammars (XSAGs) [40] and TransformX [72][6]. A restricted form of attribute forest grammars is considered which allows the evaluation of attributes on XML streams. The attribute grammars have to be L-attributed, i.e. to allow their evaluation in a single pass in document-order. Another necessary restriction is that the regular expressions in productions are *unambiguous*, as in the case of DTDs. This ensures that every parsed element corresponds to exactly one symbol in the content model of the corresponding production, which allows the unambiguous specification and evaluation of attributes. While XSAGs are targeted at ensuring scalability and have the expressiveness of deterministic pushdown transducers, the TransformX AGs allow the specification of the attribution functions in a Turing-complete programming language (Java). In both cases, for the evaluation of the attribute grammars pushdown transducers are used. The pushdown transducers used in TransformX [72] validate the input according to the grammar in a simi-

[6]In these works forest grammars are called.

lar manner to the pushdown forest automata. Additionally, a sequence of attribution functions is generated as specified by the attribute grammar. A second transducer uses this sequence and performs the specified computation. For the identification of the non-terminals from which nodes are derived in the (unique) parse tree, as needed for the evaluation of the AGs in [40, 72], pushdown forest automata can be used. The unambiguousness restriction of the attribute forest grammars allows one to proceed as in the case of right-ignoring queries presented in Section 5.6.1. That is, the non-terminal corresponding to the current node can be directly determined from the (single) NFA state in the current forest state, as it does not depend on the events after the current one.

A number of approaches handle the problem of querying XML streams in the context of selective dissemination of information (SDI), also known as XML message brokering [1,2,16,17,20,21,32,34]. In this scenario a large number of users subscribe to a dissemination system by specifying a query which acts like a filter for the documents of interest. Given an input document, the system simultaneously evaluates all user queries and distributes it to the users whose queries lead to at least one match. Strictly speaking, the queries are not answered. The documents which contain matches are dispatched but the location of the matches is not reported. XFilter [1] handle simple XPath patterns, i.e. without nested XPath patterns as filters. These can be expressed with regular expressions, hence they are implemented using finite string automata. YFilter [20] improves on XFilter by eliminating redundant processing by sharing common paths in expressions. In [21], the querying capabilities are extended to handle filters comparing attributes or text data of elements with constants and nested path expressions are allowed to occur basically only for the last location step. Green *et al.* [32] consider regular path expressions without filters. It is shown that a lazy construction of the DFA resulting from multiple XPath expressions can avoid the exponential blow-up in the number of states for a large number of queries. XPush [34] also handles nested path expressions and addresses the problem of sharing both path navigation and predicate evaluation among multiple patterns. XTrie [16] considers a query language which allows the specification of nested path expressions and, besides, an order in which they are to be satisfied. Even though Fxgrep is not targeted at SDI, note that it basically exceeds the essential capabilities of all previously mentioned query languages.

There are a number of approaches in which queries on XML streams are answered by constructing a network of transducers [24, 42, 66, 68]. A query is there compiled into a number of interconnected transducers, each of them taking as input one or more streams and producing one or more output streams by possibly using a local buffer. The XML input is delivered to one start transducer and the matches are collected from one output transducer. The query language of XSM [42] handles only XPath patterns, without filters and deep matching (//), but allows instead value-based joins. XSQ [68] deals with XPath patterns in which at most one filter can be specified for a node and filters cannot occur inside another filter. The filters only allow the comparison of the text content of a child element or an attribute with a constant. SPEX [66] basically covers Core XPath. Each transducer in the network processes the input stream and transmits it augmented with computed information to its successors. The number of transducers is linear in the query size. The complexity of answering queries depends on whether filters are allowed and is polynomial in both the size of the query and of the input. XStreamQuery [24] is an XQuery engine based on a pipeline of SAX-like event handlers augmented with the possibility of returning feedback to the producer. The strengths of this construction are its simplicity and the ability to ignore irrelevant events as soon as possible. However, the approach only handles the child and descendant axes.

FluXQuery [41] extends a subset of XQuery with constructs which guide an event-based processing of the queries using the DTD of the input. FluXQuery is used within the StreamGlobe project which is concerned with query evaluation on data streams in distributed, heterogeneous environments [76]. STX [4] is basically a restriction of the XSLT transformation language to what can be handled locally by considering only the visited part of the tree and selecting nodes from the remaining part of the tree. Sequential XPath [19] presents a quite restricted subset of XPath, handling only right-ignoring XPath patterns, which can be implemented without the need of any buffering. TurboXPath [38] introduces an algorithm for answering XPath queries containing both arithmetic and structural predicates and which is neither directly based on finite automata nor on transducer networks. The dynamic data structure WA (*work array*), used to match the document nodes has certain similarities with our construction. Entries are added in the WA upon each start-tag event for each sub-pattern to which the children must conform, which roughly correspond to a *Down* transition of the LPA. Matches of the sub-patterns are detected upon end-tag events by AND-ing the fulfillment of the sub-patterns by the children, similarly to an *Up* transition. *Side* transitions are not needed as the pattern language does not impose any order on the children nodes. In this perspective the context information is optimally used, as in our case, by a combination of top-down and bottom-up transitions. Work by Bar-Yossef *et al.* [3], indicates that the space requirement for the TurboXPath approach is near the theoretical optimum for XPath queries.

5.7 Summary and Outlook

In this chapter, we have presented querying techniques for unranked trees based on (extensions of) tree grammars and tree automata and showed how these can be used to support declarative and efficient querying of hierarchically structured and semistructured data. In particular, we have showed how they can be put to work for solving practical problems as follows.

Expressive specification of *k*-ary queries We have introduced a simple yet powerful method based on forest grammars allowing the formulation of queries which identify tuples of k related nodes in the input document tree.

Efficient evaluation of queries We have shown here how unary and binary queries can be efficiently evaluated by providing an algorithm based on pushdown forest automata. We have mentioned how the algorithm for answering binary queries can be generalized for the evaluation of queries of arbitrary arity. The complexity of query evaluation grows exponentially with k. Nevertheless, we have suggested restrictions for k-ary grammar queries under which their evaluation is efficiently implementable. We have discussed the practical aspects of implementing the queries for XML processing.

Event-driven evaluation of grammar queries on XML streams We have presented an efficient algorithm which allows the evaluation of unary grammar queries in an event-based manner. This processing method is suitable for very large documents which cannot be built completely in memory. The algorithm is also useful in settings in which documents are received linearly on some communication channel and would ideally be processed while being received, rather than waiting until the whole document is locally available.

XML querying is an important task and is the foundation for more elaborate processing tasks, such as XML transformations. Some of the ideas presented here are not only useful for querying but directly support transformations. Binary queries turn out to be especially beneficial for rule-based transformations [6,9]. Stream-based querying is a first step towards supporting stream-based transformations [7]. Other applications of tree languages to XML processing are in the area of integrating XML support directly into programming languages, e.g. for type checking [43].

5.8 Proofs of Theorems

5.8.1 *Proof of Theorem 5.21*

We start by showing that the nodes collected in the attributes of a tree state at π are from the subtree located at π.

Lemma 5.33. *If $x \in p_\pi$, $\pi_1 \in x.l_1$ then $\pi_1 = \pi\pi'$.*

Proof. The proof is by induction on the height of $f[\pi]$.

If $f[\pi] = a\langle\varepsilon\rangle$ then $p_\pi = Up^{\leftarrow}(Down^{\leftarrow}(q_\pi, a), a)$. By the definition of $Down^{\leftarrow}$, $Up^{\leftarrow}$ and attributes it follows that $\pi_1 = \pi$.

Otherwise, by the definition of attributes we have that $\pi_1 = \pi$ or there is $y \in q_{\pi 0}$, $y = y_{0,j}$, $x \to a\langle r_j\rangle$ and $\pi_1 \in y.l_1$. From $\pi_1 \in y.l_1$ it follows by straightforward induction on $n = last_f(\pi)$ that there is $x_1 \in p_{\pi i}$ and $\pi_1 \in x_1.l_1$. By the induction hypothesis it follows that $\pi_1 = \pi i \pi'$. □

5.8.1.1 *Proof of (i_1)*

Let $\pi' = \pi i$ and $n = last_f(\pi')$.

Left-to-right: From $\pi_1 \in x.l_1$ it follows by Lemma 5.33 that $\pi_1 = \pi'\pi_1'$. In the following we do the proof by induction on the length of π_1'.

If $\pi_1' = \lambda$ then $\pi_1 = \pi'$ and by the definition of attributes it follows that $x = x^1$. Our conclusion follows now by Theorem 5.15.

If $\pi_1' = l\pi_1''$ then $l \leq n$. By Theorem 5.15 there is f_a s.t. $(f, f_a) \in \mathcal{D}eriv_{r_0}$ and $lab(f_a[\pi']) = x$. From $\pi_1 \in x.l_1$ and $\pi' \neq \pi_1$ it follows by the definition of attributes that there is $x \to a\langle r_h\rangle$, $y_{0,h} \in q_{\pi' 0}$ and $\pi_1 \in y_0.l_1$. By the definition of attributes it follows by straightforward induction on n that there is m, $0 < m \leq n$ and $x_1, \ldots, x_m$, $y_1, \ldots, y_m$ s.t. $(y_{k-1}, x_k, y_k) \in \delta$, $y_k \in q_{\pi' k} \cap \overrightarrow{q}_{\pi' k}$, $x_k \in p_{\pi' k}$ for $k = 1, \ldots, m$ and $\pi_1 \in x_m.l_1$. By Lemma 5.33 $m = l$. By the induction hypothesis it follows that there is f_c s.t. $(f, f_c) \in \mathcal{D}eriv_{r_0}$, $lab(f_c[\pi' l]) = x_l$ and $lab(f_c[\pi_1]) = x^1$.

From $y_l \in q_{\pi' l} \cap \overrightarrow{q}_{\pi' l}$ it follows from the definition of $Side^{\leftarrow}$ by straightforward induction on n that there are $x_l, \ldots, x_n$, $y_l, \ldots, y_n$ s.t. $(y_{k-1}, x_k, y_k) \in \delta$, $y_k \in q_{\pi' k} \cap \overrightarrow{q}_{\pi' k}$, $x_k \in p_{\pi' k}$ for $k = m+1, \ldots, n$. Also by the definitions of $Down^{\leftarrow}$ and $Up^{\leftarrow}$ $y_0 = y_{0,h}$ and $y_n \in F_p$. As NFA transitions are done only inside one NFA we have that $p = h$ and it follows that $x_1, \ldots, x_n \in [\![r_h]\!]$.

By Theorem 5.15 there is f_k s.t. $(f, f_k) \in \mathcal{D}eriv_{r_0}$, $lab(f_k[\pi' k]) = x_k$ for all k, and by Lemma 5.34, $(f[\pi' k], f_k[\pi' k]) \in \mathcal{D}eriv_{x_k}$. Thus $(f[\pi' 1] \ldots f[\pi' n], f_1[\pi' 1] \ldots f_n[\pi' n]) \in \mathcal{D}eriv_{r_h}$ and with $x_1 \ldots x_n \in [\![r_h]\!]$, $(f[\pi'], x\langle f_1[\pi' 1] \ldots f_n[\pi' n]\rangle \in \mathcal{D}eriv_x$. Let $t = x\langle f_1[\pi' 1] \ldots f_n[\pi' n]\rangle$ and let $f_b = f_a/^{\pi}{}' t$. By Lemma 5.35, $(f, f_b) \in \mathcal{D}eriv_{r_0}$, $lab(f_b[\pi']) = x$, $lab(f_b[\pi' l]) = x_l$.

Let $f_d = f_b/^{\pi' l} f_c[\pi' l]$. By Theorem 5.22 we now have that $(f, f_d) \in \mathcal{D}eriv_{r_0}$, $lab(f_d[\pi']) = x$ and $lab(f_d[\pi_1]) = x^1$.

Right-to-left: The proof is by induction on the length of π_1'.

If $\pi_1' = \lambda$ it follows that $x = x^1$ and by the definition of attributes $\pi_1 \in x.l_1$.

If $\pi_1' = l\pi_1''$ then $l \leq n$ and let $x_k = lab(f_1[\pi' k])$ for $k = 1, \ldots, n$. By Corollary 5.17, $x_k \in p_{\pi' k}$. By Lemma 5.34 $(f[\pi'], f_1[\pi']) \in \mathcal{D}eriv_x$ and by the definition of $\mathcal{D}eriv_x$ we have that there is $x \to lab(f[\pi'])\langle r_h\rangle$ and $x_1 \ldots x_n \in [\![r_h]\!]$. Thus there are $y_0, \ldots, y_n$ s.t. $(y_{k-1}, x_k, y_k) \in \delta_h$ for $k = 1, \ldots, n$, $y_0 = y_{0,h}$ and $y_n \in F_h$. Also, by hypothesis there are $y \in q_{\pi'} \cap \overrightarrow{q}_{\pi'}$ and y' s.t. $(y', x, y) \in \delta$. Using this, one can show by using the definition of *Down*, *Side*, and $Down^{\leftarrow}$, $Side^{\leftarrow}$ that for $k = 0, \ldots, n$, $y_k \in \overrightarrow{q}_{\pi' k}$ and $y_k \in q_{\pi' k}$, respectively.

By the induction hypothesis $\pi_1 \in x_l.l_1$. By straightforward induction on l, using the definition of $Side^{\leftarrow}$ and of the attributes, it follows that $\pi_1 \in y_0.l_1$. Now by the definition of $Up^{\leftarrow}$ and of the attributes it follows that $\pi_1 \in x.l_1$.

5.8.1.2 *Proof of (i_2)*

Let $n = last_f(\pi)$.

Left-to-right: Let $y_i = y$.

From $\pi_2 \in y.l_2$ it follows from the definition of $Side^{\leftarrow}$ and of attributes by straightforward induction on n that there are j, $i < j \leq n$, $y_{i+1}, \ldots, y_j$, $x_{i+1}, \ldots, x_j$, s.t. $(y_{k-1}, x_k, y_k) \in \delta_p$ for $k = i+1, \ldots, j$ with $y_k \in q_{\pi k} \cap \overrightarrow{q}_{\pi k}$ for all k and $\pi_2 \in x_j.l_2$. By (i_1) it follows that $\pi_2 = \pi j \pi_2'$ and there is f_a

s.t. $(f, f_a) \in \mathcal{D}eriv_{r_0}$, $lab(f_a[\pi j]) = x_j$ and $lab(f_a[\pi_2]) = x^2$.

From $y_i \in q_{\pi i} \cap \overrightarrow{q}_{\pi i}$ it follows from the definitions of *Side* and *Side*$^{\leftarrow}$ that there are $y_0, \dots, y_{i-1}, x_1, \dots, x_i$ s.t. $y_k \in q_{\pi k} \cap \overrightarrow{q}_{\pi k}$ for $k = 0, \dots, i-1$, $(y_{k-1}, x_k, y_k) \in \delta_h$ for $k = 1, \dots, i$ and $y_0 = y_{0,h}$ for some h. By the Berry-Sethi construction, since $(y', x, y_i) \in \delta$ and $(y_{i-1}, x_i, y_i) \in \delta$, it follows that $x = x_i$. Similarly, from $y_j \in q_{\pi j} \cap \overrightarrow{q}_{\pi j}$ it follows that there are $y_j, \dots, y_n$ s.t. $y_k \in q_{\pi k} \cap \overrightarrow{q}_{\pi k}$ for $k = j, \dots, n$, $(y_{k-1}, x_k, y_k) \in \delta_g$ for $k = j + 1, \dots n$ and $y_n \in F_g$ for some g. Because transitions in δ can be made only inside the same NFA we have that $p = g = h$. We further get that $x_1 \dots x_n \in [\![r_h]\!]$.

By Theorem 5.15 it follows that there is f_k s.t. $(f, f_k) \in \mathcal{D}eriv_{r_0}$, $lab(f_k[\pi k]) = x_k$ and by Lemma 5.34 $(f[\pi k], f_k[\pi k]) \in \mathcal{D}eriv_{x_k}$ for $k = 1, \dots, n$. Let the forest $f_b = f_1[\pi 1] \dots f_n[\pi n]$. It follows that $(f[\pi 1] \dots f[\pi n], f_b) \in \mathcal{D}eriv_{r_h}$. Let $f_c = f_b/^j\, f_a[\pi j]$. By Lemma 5.34 $(f[\pi j], f_a[\pi j]) \in \mathcal{D}eriv_{x_j}$ and by Lemma 5.36 we have that $(f[\pi 1] \dots f[\pi n], f_c) \in \mathcal{D}eriv_{r_h}$, $lab(f_c[i]) = lab(f_b[i]) = x_i = x$ and $lab(f_c[j\pi_2']) = lab(f_a[\pi_2]) = x^2$.

Now, if $\pi = \lambda$ then $h = 0$ and $f = f[\pi 1] \dots f[\pi n]$. As above $(f, f_c) \in \mathcal{D}eriv_{r_0}$ with the required properties.

If $\pi \neq \lambda$ then by the definition of *Down*$^{\leftarrow}$ there are $y'' \in q_\pi \cap \overrightarrow{q}_\pi$, $(y''', x', y'') \in \delta$, $x' \to a\langle r_h \rangle$. By Theorem 5.15 there is f_d s.t. $(f, f_d) \in \mathcal{D}eriv_{r_0}$ and $lab(f_d[\pi]) = x'$. Let $t = x'\langle f_c \rangle$. We have that $(f[\pi], t) \in \mathcal{D}eriv_{x'}$. Let $f_e = f_d/^\pi\, t$. By Lemma 5.35 we have that $(f, f_e) \in \mathcal{D}eriv_{r_0}$ with the required properties.

Right-to-left: Let $x_k = lab(f_2[\pi k])$ for $k = 1, \dots, n$. By (i_1) $\pi_2 \in x_j.l_2$.

We first show that $x_1 \dots x_n \in [\![r_h]\!]$ for some h. If $\pi = \lambda$ then by the definition of $\mathcal{D}eriv_{r_0}$ it follows that $x_1 \dots x_n \in [\![r_0]\!]$. If $\pi \neq \lambda$ let $lab(f_2[\pi]) = x'$. It follows by Theorem 5.15 that there is $(y''', x', y'') \in \delta$ and $y'' \in q_\pi \cap \overrightarrow{q}_\pi$. By Lemma 5.34 $(f[\pi], f_2[\pi]) \in \mathcal{D}eriv_{x'}$. By the definitions of $\mathcal{D}eriv_{x'}$ there is $x' \to a\langle r_h \rangle$ and $x_1 \dots x_n \in [\![r_h]\!]$.

There are thus $y_0, \dots, y_n$ s.t. $y_0 = y_{0,h}$, $y_n \in F_h$ and $(y_{k-1}, x_k, y_k) \in \delta_h$ for all k. From the definitions of transitions it follows that $y_k \in q_{\pi k} \cap \overrightarrow{q}_{\pi k}$.

By Corollary 5.17, $x_k \in p_{\pi k}$. From $\pi_2 \in x_j.l_2$ it follows by the definitions of attributes by straightforward induction on j that $\pi_2 \in y_i.l_2$. With $y = y_i$ we get the desired result.

5.8.2 *Proof of Theorem 5.22*

We start by showing that if a derivation f' of a forest f labels a node π with x, then the trees $f[\pi]$ and $f'[\pi]$ are in the derivation relation $\mathcal{D}eriv_x$.

Lemma 5.34. *If $(f, f') \in \mathcal{D}eriv_r$ and $lab(f'[\pi]) = x$ then $(f[\pi], f'[\pi]) \in \mathcal{D}eriv_x$.*

Proof. The proof is by induction on the length of π.

Let $\pi = i$ and $last_f(\lambda) = n$. Thus $f = f[1] \dots f[n]$ and $f' = f'[1] \dots f'[n]$. From the definition of $\mathcal{D}eriv_r$ it follows that there is some $x_1 \dots x_n \in [\![r]\!]$ with $(f[k], f'[k]) \in \mathcal{D}eriv_{x_k}$ for $k = 1, \dots, n$. In particular $(f[i], f'[i]) \in \mathcal{D}eriv_{x_i}$.

Now let $\pi = \pi_1 i$, $last_f(\pi_1) = n$ and let $lab(f[\pi_1]) = a$, $lab(f'[\pi_1]) = x'$. By the induction hypothesis, $(f[\pi_1], f'[\pi_1]) \in \mathcal{D}eriv_{x'}$. By the definition of $\mathcal{D}eriv_{x'}$ there is some $x' \to a\langle r_1 \rangle \in R$ with $(f[\pi_1 1] \dots f[\pi_1 n], f'[\pi_1 1] \dots f'[\pi_1 n]) \in \mathcal{D}eriv_{r_1}$. By the definition of $\mathcal{D}eriv_{r_1}$ there is some $x_1 \dots x_n \in [\![r_1]\!]$ with $(f[\pi_1 k], f'[\pi_1 k]) \in \mathcal{D}eriv_{x_k}$ for $k = 1, \dots, n$. In particular $(f[\pi_1 i], f'[\pi_1 i]) \in \mathcal{D}eriv_{x_i}$.

In either case, from $(f[\pi], f'[\pi]) \in \mathcal{D}eriv_{x_i}$ it follows by the definition of $\mathcal{D}eriv_{x_i}$ that $x_i = lab(f'[\pi]) = x$. □

In the following we show that if a derivation f' of a forest f labels a node π with x, and there is a derivation t' of the tree $f[\pi]$ from the same x, then we obtain another derivation of f' by grafting t' into f' at π.

Lemma 5.35. *Assume $(f, f') \in \mathcal{D}eriv_r$, $lab(f'[\pi]) = x$ and $(f[\pi], t') \in \mathcal{D}eriv_x$. Then $(f, f'/^{\pi}\, t') \in \mathcal{D}eriv_r$.*

Proof. The proof is by induction on the length of π.

If $\pi = i$ then let $f = t_1 \dots t_n$ and let $(t_1 \dots t_i \dots t_n, t'_1 \dots t'_i \dots t'_n) \in \mathcal{D}eriv_r$. By the definition of $\mathcal{D}eriv_r$ there is some $x_1 \dots x_n \in [\![r]\!]$ with $(t_k, t'_k) \in \mathcal{D}eriv_{x_k}$ for $k = 1, \dots, n$. Since $t'_i = f'[i] = x\langle \_ \rangle$ it follows that $x_i = x$. From $(f[i], f'[i]) \in \mathcal{D}eriv_{x_i}$ we have that $(t_1 \dots t_i \dots t_n, t'_1 \dots t' \dots t'_n) \in \mathcal{D}eriv_r$ which is $(f, f'/^{i}\, t') \in \mathcal{D}eriv_r$.

If $\pi = ij\pi_1$ we have that $(f[1] \dots f[i] \dots f[n], f'[1] \dots f'[i] \dots f'[n]) \in \mathcal{D}eriv_r$. By the definition of $\mathcal{D}eriv_r$ there is some $x_1 \dots x_n \in [\![r]\!]$ with $(f[k], f'[k]) \in \mathcal{D}eriv_{x_k}$ for $k = 1, \dots, n$. From $(f[i], f'[i]) \in \mathcal{D}eriv_{x_i}$ it follows that $f[i] = a\langle f_1 \rangle$, $f'[i] = x_i\langle f'_1 \rangle$ and there is $x_i \to a\langle r_1 \rangle \in R$ and $(f_1, f'_1) \in \mathcal{D}eriv_{r_1}$. As $f_1[j\pi_1] = f[ij\pi_1]$ and $f'_1[j\pi_1] = f'[ij\pi_1]$ we have that $(f_1[j\pi_1], t') \in \mathcal{D}eriv_x$ and $f'_1[j\pi_1] = x\langle \_ \rangle$. It follows by the induction hypothesis that $(f_1, f'_1/^{j\pi_1}\, t') \in \mathcal{D}eriv_{r_1}$. By the definition of $\mathcal{D}eriv_{x_i}$, $(a\langle f_1\rangle, x_i\langle f'_1/^{j\pi_1}\, t'\rangle) \in \mathcal{D}eriv_{x_i}$ which is $(f[i], x_i\langle f'_1/^{j\pi_1}\, t'\rangle) \in \mathcal{D}eriv_{x_i}$. Therefore, $(f[1]\ldots f[i]\ldots f[n], f'[1]\ldots x_i\langle f'_1/^{j\pi_1}\, t'\rangle \ldots f'[n]) \in \mathcal{D}eriv_r$ which is $(f, f'/^{ij\pi_1}\, t') \in \mathcal{D}eriv_r$. □

Now we show that the forest obtained by grafting t into f at π has the nodes below π labeled as in t and all other nodes as in f.

Lemma 5.36.

$$lab((f/^{\pi}\, t)[\pi_1]) = \begin{cases} lab(t[1\pi_2]), & \textit{if } \pi_1 = \pi\pi_2 \\ lab(f[\pi_1])\,, & \textit{otherwise} \end{cases}$$

Proof. First, observe the definition of the subtree located in a grafted forest:

$$(f/^{i\pi_1}\, t)[j\pi_2] = \begin{cases} f[j\pi_2] & , \text{ if } i \neq j \\ t[1\pi_2] & , \text{ if } i = j, \pi_1 = \lambda \\ a\langle f_1/^{\pi_1}\, t\rangle & , \text{ if } i = j, \pi_1 \neq \lambda, \pi_2 = \lambda, f[i] = a\langle f_1\rangle \\ (f_1/^{\pi_1}\, t)[\pi_2], & \text{if } i = j, \pi_1 \neq \lambda, \pi_2 \neq \lambda, f[i] = a\langle f_1\rangle \end{cases}$$

The proof is by induction on the length of π.

If $\pi = i$ then if $\pi_1 = i\pi_2$, $(f/^{i}\, t)[\pi_1] = t[1\pi_2]$ thus $lab((f/^{i}\, t)[\pi_1]) = lab(t[1\pi_2])$. If $\pi_1 = j\pi_2$, $j \neq i$ then $(f/^{i}\, t)[\pi_1] = f[\pi_1]$ thus $lab((f/^{i}\, t)[\pi_1]) = lab(f[\pi_1])$.

We consider now the case where $\pi = i\pi'$, $\pi' \neq \lambda$.

If $\pi_1 = i\pi_2$ then $(f/^{\pi}\, t)[\pi_1] = (f_1/^{\pi'}\, t)[\pi_2]$, where f[i]=$a\langle f_1\rangle$. If $\pi_1 = \pi\pi_3$, i.e. if $i\pi_2 = i\pi'\pi_3$, $\pi_2 = \pi'\pi_3$ then by the induction hypothesis $lab((f_1/^{\pi'}\, t)[\pi_2]) = lab(t[1\pi_3])$. Thus $lab(f/^{i\pi'}\, t)[i\pi_2]) = lab(t[1\pi_3])$ and therefore we obtain that $lab(f/^{\pi}\, t[\pi\pi_3]) = lab(t[1\pi_3])$ as required.

Otherwise, also by the induction hypothesis $lab((f_1/^{\pi'}\, t)[\pi_2]) = lab(f_1[\pi_2])$. Since $f_1[\pi_2] = f[i\pi_2] = f[\pi_1]$ it follows that $lab((f/^{\pi}\, t)[\pi_1]) = lab(f[\pi_1])$.

If $\pi_1 = j\pi_2$ and $j \neq i$ then $(f/^{\pi}\, t)[\pi_1] = f[\pi_1]$ thus $lab((f/^{\pi}\, t)[\pi_1]) = lab(f[\pi_1])$. □

Using the lemmas above we prove now Theorem 5.22.

Let $lab(f_1[\pi]) = lab(f_2[\pi]) = x$. By Lemma 5.34 we have that $(f[\pi], f_2[\pi]) \in \mathcal{D}eriv_x$. From Lemma 5.35 it follows that $(f, f_1/^{\pi}\, f_2[\pi]) \in \mathcal{D}eriv_r$. By Lemma 5.36:

$$lab((f_1/^{\pi}\, f_2[\pi])[\pi_1]) = \begin{cases} lab(f_2[\pi][1\pi_2]), & \text{if } \pi_1 = \pi\pi_2 \\ lab(f[\pi_1]) & , \text{ otherwise} \end{cases}$$

With $f_2[\pi][1\pi_2] = f_2[\pi\pi_2]$ we obtain now the result of our theorem.

5.8.3 *Proof of Theorem 5.32*

In the following we use the notation from Chapter 5.6 where Theorem 5.32 was stated.

Alternative Definition of Matches

In order to proof Theorem 5.32 a more refined definition of matches is needed in which the NFA states reached while checking the content models of elements are given explicitely. Let R be a set of forest grammar productions, r_0 be a regular expression over non-terminals and f an input forest. A *(non-deterministic, accepting) run* f_R over f for R and r_0, denoted $f_R \in \mathcal{R}uns_{r_0,f}$ is defined as follows:

$$y_0\langle f_1'\rangle \;\dots\; y_{n-1}\langle f_n'\rangle \; y_n\langle\rangle \in \mathcal{R}uns_{r_0, a_1\langle f_1\rangle \;\dots\; a_n\langle f_n\rangle} \text{ iff}$$
$$y_0 = y_{0,0}, y_n \in F_0, \text{ and}$$
$$(y_{i-1}, x_i, y_i) \in \delta_0, x_i \to a_i\langle r_i\rangle, f_i' \in \mathcal{R}uns_{r_i,f_i} \text{ for all } i = 1, \dots, n$$

$$y \in \mathcal{R}uns_{r_0,\varepsilon} \text{ iff } y = y_{0,0}, y \in F_0$$

An example run is given immediately below.

It is straightforward to see that a derivation f' with $(f, f') \in \mathcal{D}eriv_{r_0}$ (defined on page 219) exists iff a run $f_R \in \mathcal{R}uns_{r_0,f}$ exists.

Example 5.37. Let $G = (R, r_0)$ with R being the set of rules from Example 5.4 reproduced for convenience below:

$$\begin{array}{lll}
(1)\ x_\top \to a\langle x_\top^*\rangle & (4)\ x_1 \to a\langle x_\top^*(x_1|x_a)x_\top^*\rangle & (6)\ x_b \to b\langle x_\top^*\rangle \\
(2)\ x_\top \to b\langle x_\top^*\rangle & (5)\ x_a \to a\langle x_b x_c\rangle & (7)\ x_c \to c\langle x_\top^*\rangle \\
(3)\ x_\top \to c\langle x_\top^*\rangle & &
\end{array}$$

The NFAs for the regular expressions occurring in grammar G with the set are reproduced in Figure 5.22.

Consider the input tree depicted t reproduced for convenience in Figure 5.23 and one derivation of t' w.r.t. r_0 depicted in Figure 5.24. A run corresponding to t' is depicted in Figure 5.25 via dotted lines.

The derivation corresponding to a run can be obtained by taking the incoming transitions of the NFA states of the nodes which are not the first in their siblings sequence as one can see in Figure 5.25. Formally, the following expresses the relation between *derivations* and *runs*:

$$\begin{aligned}
&(f, f') \in \mathcal{D}eriv_r \\
&\quad \text{iff } \exists f_R \in \mathcal{R}uns_{r,f} \text{ with } L(f') = N(f_R) \\
&\qquad \text{and } lab(f'[\pi p]) = in(lab(f_R[\pi(p+1)])) \text{ for all } \pi p \in N(f').
\end{aligned}$$

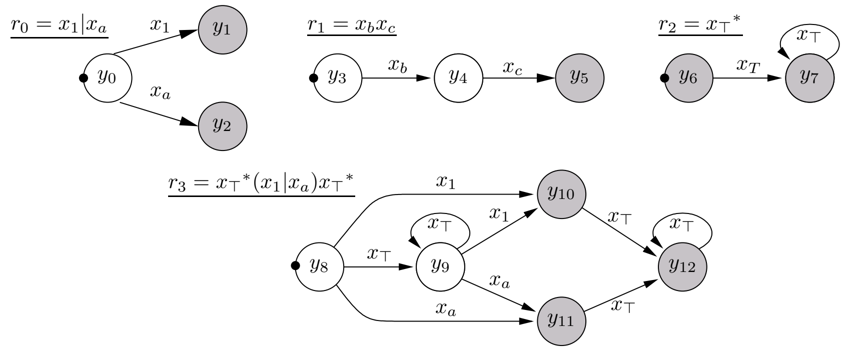


Fig. 5.22 NFAs obtained by Berry-Sethi construction for regular expressions in Example 5.37.

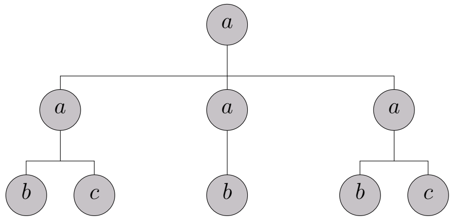


Fig. 5.23 Input tree t.

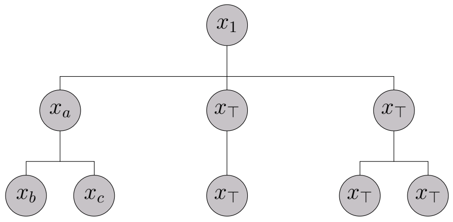


Fig. 5.24 Derivation t' of t w.r.t r_0.

Let f be an input forest and $Q = ((R, r_0), T)$ a grammar query. Matches of Q, which were originally defined in terms of *derivations*, can be equivalently defined in terms of runs as it follows:

$$\pi p \in \mathcal{M}_{Q,f} \text{ iff } \exists f_R \in \mathcal{R}uns_{r_0,f} \text{ s.t. } in(lab(f_R[\pi(p+1)])) \in T.$$

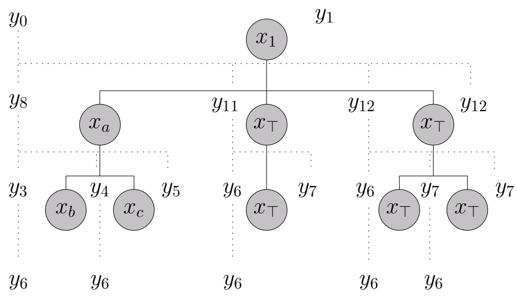


Fig. 5.25 Run corresponding to t'.

Notations

Before proceeding with the proof we further introduce a couple of useful notations. The set of *matches defined by runs with label y at location l* is defined as:

$$\begin{aligned}\pi p \in \mathcal{M}^{l,y}_{Q,f} \text{ iff } \exists f_R \in \mathcal{R}uns_{r_0,f} \text{ s.t. } & in(lab(f_R[\pi(p+1)])) \in T \\ \text{and } & lab(f_R[l]) = y\end{aligned}$$

The set of *l-right-ignoring matches defined by a run with label y at l* is defined as:

$$\pi \in ri\text{-}\mathcal{M}^{l,y}_{Q,f} \text{ iff } \pi \in \mathcal{M}^{l,y}_{Q,f_2} \forall f_2 \in \mathcal{R}ightCompl_{f,l}$$

A node π' is a *πi-upper-right ignoring match defined by a run with label y at πi* iff for any right-completion f_2 at the parent of πi there is a run defining π' as a match of Q in f_2 which labels πi with y, formally:

$$\pi' \in uri\text{-}\mathcal{M}^{\pi i,y}_{Q,f} \text{ iff } \pi' \in \mathcal{M}^{\pi i,y}_{Q,f_2} \forall f_2 \in \mathcal{R}ightCompl_{f,\pi}$$

Given a location πi and an NFA state y, a sequence of states is a *suffix run from y at πi* iff the last state in the sequence is a final state and the sequence of siblings to the right of πi allows to visit the sequence of states, formally:

$$\begin{aligned}& y_i, \ldots, y_n \in \mathcal{S}uf_{\pi i,y} \\ & \text{ iff } (y_{k-1}, x_k, y_k) \in \delta_j, f_1[\pi k] \in [\![R]\!]\, x_k \text{ with } x_k = in(y_k), \forall k \in i, \ldots, n \text{ and} \\ & \quad y_{i-1} = y, y_n \in F_j \text{ where } n = last_{f_1}(\pi)\end{aligned}$$

To denote the information on top of the stack at the some moment πi we write $\pi i.q$, $\pi i.m$ and $\pi i.ri$ in analogy to attributes of attribute grammars. Similarly to attribute grammars, these are computed by local rules as presented in Section 5.6.2.1.

Proof

Theorem 5.32 is a straightforward corollary of the following theorem:

Theorem 5.38. *The construction presented in Section 5.6.2.1 keeps the following invariant:*

$$\begin{aligned}&\pi' p \in \pi i.m(y), y \in \pi i.q \cap Y_j, \exists c \in \mathcal{S}uf_{\pi i,y} \textit{ and } r_j \in \pi i.ri \\ &\textit{iff } \pi' p < \pi i \textit{ and } \pi' p \in uri\text{-}\mathcal{M}^{\pi i,y}_{Q,f}\end{aligned} \tag{5.3}$$

Proof. We proof the two directions of Theorem 5.38 separately.

Left-to-right

We show that (5.3) holds at all locations in the input by induction using the lexicographic order on locations.

Base case Initially, at location 1, $1.m(y) = \varnothing, \forall y \in 1.q$, thus $\pi' p \in 1.m(y)$ is false, and the left-to-right direction trivially holds.

Induction step Supposing that (5.3) holds at all locations up to some location l we show that it also holds at the immediately next location.

Start-tag transition We first show that if (5.3) holds at $\pi i \in N(f)$, so does it at $\pi i1$.

Let $\pi' p \in \pi i1.m(y_0)$, $y_0 \in Y_j$ and suppose $\exists c \in \mathcal{S}uf_{\pi i1,y_0}$ and $r_j \in \pi i1.ri$. Since $\pi' p \in \pi i1.m(y_0)$, it follows by our construction (conform to (5.1) on page 285) that $\exists y \in \pi i.q$ with $(y, x, y') \in \delta_k$, $x \to a\langle r_j \rangle$, $rightIgn(y')$, $r_k \in \pi i.ri$ and either (i) $\pi' p \in \pi i.m(y)$ or (ii) $\pi' p = \pi i$ and $x \in T$.

In case (i) it follows from (5.3) at πi that $\pi' p < \pi i$ and $\pi' p \in uri\text{-}\mathcal{M}^{\pi i,y}_{Q,f}$. Thus, obviously $\pi' p < \pi i < \pi i1$ and it remains to show that $\pi' p \in uri\text{-}\mathcal{M}^{\pi i1,y_0}_{Q,f}$. This follows directly from $\pi' p \in uri\text{-}\mathcal{M}^{\pi i,y}_{Q,f}$, $c \in \mathcal{S}uf_{\pi i1,y_0}$ and $rightIgn(y')$ by grafting the run over the children of πi corresponding to c into the run corresponding to $uri\text{-}\mathcal{M}^{\pi i,y}_{Q,f}$.

In case (ii), $\pi' p = \pi i < \pi i1$. The proof will use in this case the following lemma (also used later on):

Lemma 5.39. *If there is a suffix run within a right ignoring content model, then, independently of what follows in the input after the enclosing element, there is a run over the input forest containing that suffix. Formally, if* $y \in \pi i.q \cap Y_k$, $r_k \in \pi i.ri$ *and* $\exists c \in \mathcal{S}uf_{\pi i,y}$ *then* $\forall f_2 \in \mathcal{R}ightCompl_{f,\pi}$ $\exists f_R \in \mathcal{R}uns_{r_0,f_2}$ *with* $c = lab(f_R[\pi i]), \ldots, lab(f_R[\pi\ last_{f_R}(\pi)])$.

Proof. The proof is by straightforward induction on the locations in the input forest. The assertion trivially holds at location 1. For the induction step, let $\pi i \in N(f)$. We show that if the assertion holds at the location πi, it also holds at (i) $\pi i1$ and (ii) at $\pi(i+1)$. In case (i) $\exists y \in \pi i.q$ with $(y, x, y') \in \delta_k$, $x \to a\langle r_j \rangle$, $rightIgn(y')$, $r_k \in \pi i.ri$. The required run is obtained by grafting the run over the children of πi corresponding to c into the run $y, y', \ldots$ corresponding to the induction hypothesis. In case (ii) the existence of the suffix run at $\pi(i+1)$ implies the existence of a run at πi and our conclusion follows by the induction hypothesis. $\square$

We continue now with the proof of Theorem 5.38.

Since $c \in \mathcal{S}uf_{\pi i1, y_0}$ it follows (straightforwardly by definition) that $f[\pi i] \in [\![R]\!]\, x$. Given that $(y, x, y') \in \delta_k$ and $rightIgn(y')$ it follows that there is thus a suffix run $c \in \mathcal{S}uf_{\pi i, y}$ with $c = y, y', \ldots$.

With $r_k \in \pi i.ri$ it follows by Lemma 5.39 that $\forall f_2 \in \mathcal{R}ightCompl_{f,\pi}$ $\exists f_R \in \mathcal{R}uns_{r_0, f_2}$ with $lab(f_R[\pi i]) = y'$. Since $rightIgn(y')$ it follows that $\exists f_R \in \mathcal{R}uns_{r_0, f_2}$ for any $f_2 \in \mathcal{R}ightCompl_{f,\pi i}$ and $lab(f_R[\pi i]) = y'$. With $c \in \mathcal{S}uf_{\pi i1, y_0}$ it follows that $\exists f'_R \in \mathcal{R}uns_{r_0, f_2}$ (obtained by grafting the run over the children of πi corresponding to c into f_R) with $lab(f'_R[\pi i]) = y'$ and $lab(f'_R[\pi i1]) = y_0$. Thus $\pi' p \in uri\text{-}\mathcal{M}_{Q,f}^{\pi i1, y_0}$.

End-tag transition We next show that if (5.3) holds at $l\ \forall l < \pi(i+1)$, so does it at $\pi(i+1)$.

Let $\pi' p \in \pi(i+1).m(y'')$, $y'' \in Y_k$ and suppose $\exists c \in \mathcal{S}uf_{\pi(i+1), y''}$ and $r_k \in \pi(i+1).ri$. Since $\pi' p \in \pi(i+1).m(y'')$, it follows by our construction (conform to (5.2) on page 286) that $\exists y \in \pi i.q$, $y' \in \pi i(n+1).q$ with $y' \in F_j$, $x \to a\langle r_j \rangle$, $(y, x, y'') \in \delta_k$ and either (i) $\pi' p \in \pi i.m(y)$, or (ii) $\pi' p \in \pi i(n+1).m(y')$, or (iii) $\pi' p = \pi i$ and $x \in T$.

In case (i) our conclusion follows directly from (5.3) at πi.

We continue with the cases (ii) and (iii). Given c and $r_k \in \pi(i+1).ri$ it follows by Lemma 5.39 that $\forall f_2 \in \mathcal{R}ightCompl_{f,\pi}$ $\exists f_R \in \mathcal{R}uns_{r_0, f_2}$ s.t. $lab(f_R[\pi(i+1)]) = y''$. Further we use the following lemma (also employed later on):

Lemma 5.40. *If* $y \in \pi n.q \cap \mathcal{F}_j$ *then* $\exists f_R \in \mathcal{R}uns_{r_j, f[\pi 1] \ldots f[\pi n]}$ *with* $lab(f_R[\pi(n+1)]) = y$.

Proof. The proof is straightforward by induction on the depth of $f[\pi]$. $\square$

In case (ii), $\pi' p$ was found either before or while visiting the content of πi, that is either $\pi' p \le \pi i$ or $\pi i < \pi' p < \pi(i+1)$, respectively. In the first case our conclusion follows directly from (5.3) at πi. In the second case $\pi' p < \pi(i+1)$ we further need the following lemma:

Lemma 5.41. *If $y \in \pi n.q \cap \mathcal{F}_j$, $\pi' p \in \pi n.m(y)$ and $\pi 1 \leq \pi' p \leq \pi n$ then $\exists f_R \in \mathcal{R}uns_{r_j, f[\pi 1] \ldots f[\pi n]}$ with $lab(f_R[\pi(n+1)]) = y$ and $in(lab(f_R[\pi'(p+1)])) \in T$ where $n = last_f(\pi)$.*

Proof. The proof is by induction on the depth of $f[\pi]$. By Lemma 5.40 $\exists f'_R \in \mathcal{R}uns_{r_j, f[\pi 1] \ldots f[\pi n]}$ with $lab(f'_R[\pi(n+1)]) = y$.

For depth 1 it directly follows that $\pi' p = \pi i$ for some $1 \leq i \leq n$ and $in(lab(f'_R[\pi'(p+1)])) \in T$. Therefore $f_R = f'_R$ is the sought after run. If the depth is more than 1, then either (A) $\pi' p = \pi i$ for some $1 \leq i \leq n$ and $in(lab(f_R[\pi'(p+1)])) \in T$ as above or (B) $\exists y' \in \pi i n'.q \cap \mathcal{F}_k$, $\pi' p \in \pi i n'.m(y)$ and $\pi i 1 \leq \pi' p \leq \pi i n'$ for some $1 \leq i \leq n$ and $n' = last_f(\pi i)$. In case (B) f_R in our conclusion can be constructed by grafting the run over the children of πi existent by the induction hypothesis into f'_R. $\square$

Our conclusion results now for the case (ii) $\pi i < \pi' p < \pi(i+1)$ by grafting the run corresponding to the children which defines the match (as implied by Lemma 5.41) into f_R.

In case (iii) $\pi' p = \pi i < \pi(i+1)$ and it remains to show that $\pi' p \in uri\text{-}\mathcal{M}^{\pi(i+1), y''}_{Q,f}$. We have by Lemma 5.40 that $\exists f'_R \in \mathcal{R}uns_{r_j, f[\pi i 1] \ldots f[\pi i n]}$ and $in(lab(f'_R[\pi'(p+1)])) \in T$. From f_R and f'_R it results (by grafting f'_R into f_R at π) that $\exists f''_R \in \mathcal{R}uns_{r_0, f_2}$ s.t. $in(lab(f''_R[\pi'(p+1)])) \in T$ and $lab(f''_R[\pi(i+1)]) = y''$, thus $\pi' p \in uri\text{-}\mathcal{M}^{\pi(i+1), y''}_{Q,f}$.

Right-to-left

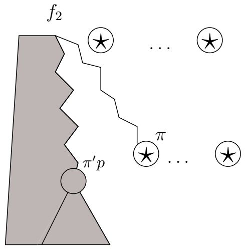


Fig. 5.26 Right completion of f at π.

Let $\pi' p < \pi i$ and $\pi' p \in uri\text{-}\mathcal{M}^{\pi i, y}_{Q,f}$. Let f_2 be a right-completion of f at π obtained by adding on every level from the root to π inclusively an arbitrary number of right siblings $\star\langle\rangle$, as depicted in Figure 5.26, where $\star$ is a symbol not occurring in any of the rules in the grammar. Since $\pi'p \in$ $\mathit{uri}\text{-}\mathcal{M}_{Q,f}^{\pi i,y}$ it follows that $\exists f_R \in \mathcal{R}\mathit{uns}_{G,f_2}$ s.t. $\mathit{in}(\mathit{lab}(f_R[\pi'(p+1)])) \in T$ and $\mathit{lab}(f_R[\pi i]) = y$.

Also, since $\star$ does not occur in any rule f_R must label all the ancestors of the πi node with right-ignoring states, i.e. $\mathit{rightIgn}(\mathit{lab}(f_R[\pi_1(k+1)]))\forall \pi_1 k \in \mathit{ancestors}_f(\pi i)$, where $\mathit{ancestors}_f : N(f) \mapsto N(f)$ is defined as follows:

$$\begin{array}{lcl} \mathit{ancestors}_f(i) & = & \varnothing \\ \mathit{ancestors}_f(\pi i) & = & \{\pi\} \cup \mathit{ancestors}_f(\pi) \end{array}$$

It follows that $\exists f'_R \in \mathcal{R}\mathit{uns}_{G,f}$ s.t. $\mathit{in}(\mathit{lab}(f'_R[\pi'(p+1)])) \in T$ and $\mathit{lab}(f'_R[\pi i]) = y$. Suppose that $y \in Y_j$. Since y is part of a run (f'_R), it obviously holds that $\exists c \in \mathcal{S}\mathit{uf}_{\pi i,y}$.

Also, since $\mathit{rightIgn}(\mathit{lab}(f_R[\pi_1(k+1)]))\forall \pi_1 k \in \mathit{ancestors}_f(\pi i)$, we obtain by using the NFA transitions in f'_R at the corresponding steps in our construction that all content models of the elements enclosing πi are right ignoring, thus $r_j \in \pi i.ri$.

Given that $\pi'p < \pi i$ it follows that there is an ancestor of $\pi'p$ which is either (i) a sibling of an ancestor a of πi or (ii) an ancestor a of πi. In any case it follows by using the NFA transitions in f'_R at the corresponding steps in our construction that $\pi'p$ is propagated down at location a until πi, thus $\pi'p \in \pi i.m(y)$.

We have proven thus that $\pi'p \in \pi i.m(y)$, $y \in \pi i.q \cap Y_j$, $\exists c \in \mathcal{S}\mathit{uf}_{\pi i,y}$ and $r_j \in \pi i.ri$.

This completes the proof of Theorem 5.38. $\square$

Theorem 5.32 follows now directly from Theorem 5.38.

References

1. Altinel, M. and Franklin, M. J. (2000). Efficient Filtering of XML Documents for Selective Dissemination of Information, in *Procedings of the 28th International Conference on Very Large Data Bases (VLDB 2000)*, pp. 53–64.
2. Avila-Campillo, I., Green, T. J., Gupta, A., Suciu, D. and Onizuka, M. (2002). XMLTK: An XML Toolkit for Scalable XML Stream Processing, in *Workshop on Programming Language Technologies for XML (PLAN-X)*, pLAN-X 2002.
3. Bar-Yossef, Z., Fontoura, M. and Josifovski, V. (2004). On the Memory Requirements of XPath Evaluation over XML Streams, in *Proceedings of*

the 20th Symposium on Principles of Database Systems (PODS 2004), pp. 391–441.

4. Becker, O. (2003). Transforming XML on the Fly, in *XML Europe 2003.*
5. Berger, S., Bry, F. and Schaffert, S. (2003). A Visual Language for Web Querying and Reasoning, in *Proceedings of Workshop on Principles and Practice of Semantic Web Reasoning, Mumbai, India (9th–13th December 2003), LNCS*, Vol. 2901, pp. 99–112.
6. Berlea, A. (2005). *Efficient XML Processing with Tree Automata*, Ph.D. thesis, Technical University of Munich, Munich.
7. Berlea, A. (2007). On-the-fly Tuple Selection for XQuery, in *Proceedings of the International Workshop on XQuery Implementation, Experience and Perspectives*, pp. 1–6.
8. Berlea, A. and Seidl, H. (2002). Binary Queries, in *Extreme Markup Languages 2002.*
9. Berlea, A. and Seidl, H. (2004). Binary Queries for Document Trees, *Nordic Journal of Computing* **11**, 1, pp. 41–71.
10. Berry, G. and Sethi, R. (1986). From Regular Expressions to Deterministic Automata, *Theoretical Computer Science Journal* **48**, pp. 117–126.
11. Bird, S., Chen, Y., Davidson, S., Lee, H. and Zheng, Y. (2005). Extending XPath to Support Linguistic Queries, in *Workshop on Programming Language Technologies for XML (PLAN-X) 2005*, pp. 35–46.
12. Bojanczyk, M. and Colcombet, T. (2005). Tree-Walking Automata Do Not Recognize All Regular Languages, in *Proceedings Of The ACM Symposium on Theory of Computing, STOC 2005*, pp. 234–243.
13. Brüggemann-Klein, A., Murata, M. and Wood, D. (2001). Regular Tree and Regular Hedge Languages over Non-Ranked Alphabets, Research report, HKUST Theoretical Computer Science Center.
14. Brüggemann-Klein, A. and Wood, D. (2000). Caterpillars: A Context Specification Technique, Research Report TCSC-2000-8, HKUST Theoretical Computer Science Center.
15. Ceri, S., Comai, S., Damiani, E., Fraternali, P., Paraboschi, S. and Tanca, L. (1999). XML-GL: A graphical language for querying and restructuring XML documents, in *Sistemi Evoluti per Basi di Dati*, pp. 151–165.
16. Chan, C.-Y., Felber, P., Garofalakis, M. and Rastogi, R. (2002). Efficient Filtering of XML Documents with XPath Expressions, in *Proceedings of the International Conference on Data Engineering (ICDE 2002).*
17. Chean, J., DeWitt, D. J., Tian, F. and Wang, Y. (2000). NiagaraCQ: a Scalable Continuous Query System for Internet Databases, in *Proceedings of the International Conference on Management of Data (SIGMOD 2000).*
18. Chitic, C. and Rosu, D. (2004). On Validation of XML Streams using Finite State Machines, in *WebDB '04: Proceedings of the 7th International Workshop on the Web and Databases* (ACM Press, New York, NY, USA), pp. 85–90.
19. Desai, A. (2001). Introduction to Sequential XPath, in *XML Conference 2001.*
20. Diao, Y., Fischer, P., Franklin, M. J. and To, R. (2002). YFilter: Efficient

and Scalable Filtering of XML Documents, in *Proceedings of the International Conference on Data Engineering (ICDE 2002).*

21. Diao, Y. and Franklin, M. (2003). YFilter: Query Processing for High-Volume XML Message Brokering, in *Proceedings of the 29th International Conference on Very Large Data Bases (VLDB 2003).*
22. DOM (1998). Document Object Model (DOM) Level 1 Specification, Version 1.0, `http://www.w3.org/TR/1998/REC-DOM-Level-1-19981001`.
23. Erwig, M. (2003). Xing: a Visual XML Query Language, *Journal of Visual Languages and Computing* **14**, 1, pp. 5–45.
24. Fegaras, L. (2004). The Joy of SAX, in *Proceedings of the First International Workshop on XQuery Implementation, Experience and Perspectives.*
25. Fernandez, M., Siméon, J. and Wadler, P. (1999). XML Query Languages: Experiences and Exemplars, Draft manuscript: `http://www.w3.org/1999/09/ql/docs/xquery.html`.
26. Fernández, M. and Suciu, D. (1998). Optimizing Regular Path Expressions Using Graph Schemas, in *Proceedings of the International Conference on Data Engineering (ICDE 1998).*
27. Frick, M., Grohe, M. and Koch, C. (2003). Query Evaluation on Compressed Trees, in *Proceedings of the 18th IEEE Symposium on Logic in Computer Science*, pp. 188–197.
28. Gécseg, F. and Steinby, M. (1997). Tree Languages, in G. Rozenberg and A. Salomaa (eds.), *Handbook of Formal Languages*, Vol. 3, chap. 1 (Springer, Heidelberg), pp. 1–68.
29. Gottlob, G., Koch, C. and Pichler, R. (2002). Efficient Algorithms for Processing XPath Queries, in *Proc. 28th Int. Conf. on Very Large Data Bases (VLDB 2002)* (Morgan Kaufmann, Hong Kong, China), pp. 95–106.
30. Gottlob, G., Koch, C. and Pichler, R. (2003). The Complexity of XPath Query Evaluation, in *Proceedings of the Eighteenth Symposium on Principles of Database Systems (PODS 2003).*
31. Gottlob, G., Koch, C. and Schulz, K. (2004). Conjunctive Queries over Trees, in *Proceedings of the 23rd ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS 2004).*
32. Green, T. J., Miklau, G., Onizuka, M. and Suciu, D. (2003). Processing XML Streams with Deterministic Automata, in *Proceedings of International Conference on Database Theory (ICDT 2003)*, pp. 173–189.
33. Grep (2005). Gnu grep, Software and Documentation.
34. Gupta, A. and Suciu, D. (2003). Stream Processing of XPath Queries with Predicates, in *Proceedings of the International Conference on Management of Data(SIGMOD 2003).*
35. Harold, E. R. (2001). *XML Bible* (John Wiley & Sons, Inc., New York, NY, USA), ISBN 0764547607.
36. Hosoya, H. and Pierce, B. C. (2000). XDuce: A Typed XML Processing Language, in *Proceedings Of The Third International Workshop on the Web and Databases (WebDB2000), Dallas, Texas*, pp. 111–116.
37. Hosoya, H. and Pierce, B. C. (2003). XDuce: A Statically Typed XML Processing Language, *ACM Trans. Inter. Tech.* **3**, 2, pp. 117–148.

38. Josifovski, V., Fontoura, M. and Barta, A. (2005). Querying XML Streams, *The VLDB Journal* **14**, 2, pp. 197–210.
39. Klarlund, N., Møller, A. and Schwartzbach., M. (2000). DSD: A Schema Language for XML, in *ACM SIGSOFT Workshop on Formal Methods in Software Practice*.
40. Koch, C. and Scherzinger, S. (2003). Attribute Grammars for Scalable Query Processing on XML Streams, in *Database Programming Languages (DBPL)*, pp. 233–256.
41. Koch, C., Scherzinger, S., Schweikardt, N. and Stegmaier, B. (2004). Schema-based Scheduling of Event-Processors and Buffer Minimization for Queries on Structured Data Streams, in *Proceedings of the 30th International Conference on Very Large Data Bases (VLDB 2004)*.
42. Ludäscher, B., Mukhopadhyay, P. and Papakonstantinou, Y. (2002). A Transducer-Based XML Query Processor, in *Proceedings of the 28th International Conference on Very Large Data Bases (VLDB 2002)*.
43. Maneth, S., Berlea, A., Perst, T. and Seidl, H. (2005). XML Type Checking with Macro Tree Transducers, in *Proceedings of the 20th Symposium on Principles of Database Systems (PODS 2005)*.
44. Marx, M. (2004). Conditional XPath, the First Order Complete XPath Dialect, in *PODS '04: Proceedings of the twenty-third ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems* (ACM, New York, NY, USA), ISBN 158113858X, pp. 13–22.
45. Marx, M. and de Rijke, M. (2005). Semantic characterizations of navigational xpath, *SIGMOD Rec.* **34**, 2, pp. 41–46.
46. Meuss, H. and Schulz, K. U. (2001). Complete Answer Aggregates for Tree-like Databases: a Novel Approach to Combine Querying and Navigation, *ACM Transactions on Information Systems* **19**, 2, pp. 161–215.
47. Meuss, H., Schulz, K. U., Weigel, F., Leonardi, S. and Bry, F. (2005). Visual Exploration and Retrieval of XML Document Collections with the Generic System X^2, *Journal on Digital Libraries, Special Issue on "Information Visualization Interfaces for Retrieval and Analysis"* **5**.
48. Munroe, K. D. and Papakonstantinou, Y. (2000). BBQ: A Visual Interface for Integrated Browsing and Querying of XML, in *VDB 5: Proceedings of the Fifth Working Conference on Visual Database Systems* (Kluwer, B.V., Deventer, The Netherlands, The Netherlands), ISBN 0-7923-7835-0, pp. 277–296.
49. Murata, M. (1997). Transformation of Documents and Schemas by Patterns and Contextual Conditions, in *Principles of Document Processing '96*, Vol. 1293 (Springer-Verlag), pp. 153–169.
50. Murata, M. (2001). Extended Path Expressions for XML, in *Proceedings of the 17th Symposium on Principles of Database Systems (PODS 2001)*.
51. Murata, M., Lee, D. and Mani., M. (2001). Taxonomy of XML Schema Languages Using Formal Language Theory, in *Extreme Markup Languages 2001, Montreal, Canada*.
52. Nakano, K. and Nishimura, S. (2001). Deriving Event-Based Document Transformers from Tree-Based Specifications, in M. van den Brand and

D. Parigot (eds.), *Electronic Notes in Theoretical Computer Science*, Vol. 44 (Elsevier Science Publishers).

53. Neumann, A. (2000). *Parsing and Querying XML Documents in SML*, Ph.D. thesis, University of Trier, Trier.
54. Neumann, A. and Berlea, A. (2010). fxgrep 4.6.4, `http://www2.informatik.tu-muenchen.de/~berlea/Fxgrep/`.
55. Neumann, A. and Seidl, H. (1998a). Locating Matches of Tree Patterns in Forests, in V. Arvind and R. Ramamujan (eds.), *Foundations of Software Technology and Theoretical Computer Science, (18th FST&TCS)*, *Lecture Notes in Computer Science*, Vol. 1530 (Springer, Heidelberg), pp. 134–145.
56. Neumann, A. and Seidl, H. (1998b). Locating Matches of Tree Patterns in Forests, Tech. Rep. 98-08, Mathematik/Informatik, Universität Trier.
57. Neven, F. (2005). Extensions of Attribute Grammars for Structured Document Queries, *Journal of Computer and System Sciences* **70**, 2, pp. 221–257.
58. Neven, F. and Bussche, J. V. D. (2002). Expressiveness of Structured Document Query Languages Based on Attribute Grammars, *Journal of the ACM* **49**, 1, pp. 56–100.
59. Neven, F. and Schwentick, T. (1999). Query Automata, in *Proceedings of the Eighteenth Symposium on Principles of Database Systems, May 31 - June 2, 1999, Philadelphia, Pennsylvania* (ACM Press), pp. 205–214.
60. Neven, F. and Schwentick, T. (2000). Expressive and Efficient Pattern Languages for Tree-structured Data, in *Proceedings of the Nineteenth Symposium on Principles of Database Systems 2000* (ACM Press), pp. 145–156.
61. Neven, F. and Schwentick, T. (2002). Query Automata over Finite Trees, *Theoretical Computer Science Journal* **275**, 1-2, pp. 633–674.
62. Neven, F. and Schwentick, T. (2003). Automata- and logic-based pattern languages for tree-structured data, in K.-D. S. L. Bertossi, G. Katona and B. Thalheim (eds.), *Semantics in Databases*, *Lecture Notes in Computer Science*, Vol. 2582 (Springer, Heidelberg), pp. 160–178.
63. Neven, F., Schwentick, T. and Vianu, V. (2001). Towards Regular Languages over Infinite Alphabets, in *MFCS '01: Proceedings of the 26th International Symposium on Mathematical Foundations of Computer Science* (Springer-Verlag, London, UK), ISBN 3-540-42496-2, pp. 560–572.
64. Niehren, J., Planque, L., Talbot, J.-M. and Tison, S. (2005). N-ary queries by tree automata, in *10th International Symposium on Database Programming Languages*, Vol. 3774, pp. 217–231.
65. OASIS (2001). RelaxNG Specification, `http://www.relaxng.org/`.
66. Olteanu, D., Furche, T. and Bry, F. (2004). Evaluating Complex Queries against XML Streams with Polynomial Combined Complexity, in *Proc. of 21st Annual British National Conference on Databases (BNCOD21)*.
67. Olteanu, D., Meuss, H., Furche, T. and Bry, F. (2002). XPath: Looking Forward, in *Proc. of Workshop on XML-Based Data Management (XMLDM) at EDBT 2002*.
68. Peng, F. and Chawathe, S. S. (2003). XPath Queries on Streaming Data, in *Proceedings of the International Conference on Management of Data(SIGMOD 2003)*.

69. Pito, R. (1996). Tgrep Documentation, `http://www.ldc.upenn.edu/ldc/online/treebank/`.
70. Rohde, D. L. T. (2004). Tgrep2 User Manual, `tedlab.mit.edu/~dr/Tgrep2/tgrep2.pdf`.
71. SAX (1998). SAX 1.0: The Simple API for XML, `http://www.megginson.com/SAX/index.html`.
72. Scherzinger, S. and Kemper, A. (2005). Syntax-directed Transformations of XML Streams, in *Workshop on Programming Language Technologies for XML (PLAN-X) 2005*.
73. Schwentick, T. (2000). On Diving into Trees, in *Proceedings of the 25-th Symposium on Mathematical Foundations of Computer Science 2000* (ACM Press), pp. 660–669.
74. Segoufin, L. and Vianu, V. (2002). Validating Streaming XML Documents, in *Symposium on Principles of Database Systems*, pp. 53–64.
75. Seidl, H., Schwentick, T. and Muscholl, A. (2003). Numerical Document Queries, in *Proceedings of the Eighteenth Symposium on Principles of Database Systems (PODS 2003)*, pp. 155–166.
76. Stegmaier, B., Kuntschke, R. and Kemper, A. (2004). StreamGlobe: Adaptive Query Processing and Optimization in Streaming P2P Environments, in *Proceedings of the International Workshop on Data Management for Sensor Networks*, pp. 88–97.
77. Thatcher, J. and Wright, J. (1968). Generalized Finite Automata with an Application to a Decision Problem of Second Order Logic, *Mathematical Systems Theory* **2**, pp. 57–82.
78. Unicode (1996). The Unicode Standard, Version 2.0, Addison Wesley Developers Press, Reading, Massachusetts.
79. van Best, J.-P. (1998). Tree-Walking Automata and Monadic Second Order Logic, Master's Thesis, TU Delft.
80. Weiand, K., Furche, T. and Bry, F. (2008). Quo vadis, web queries, in *Proceedings of International Workshop on Semantic Web Technologies, Belgrade, Serbia (29th–30th September 2008)*.
81. XML (1998). Extensible Markup Language (XML) 1.0, `http://www.w3.org/TR/REC-xml/`.
82. XMLSchema (2001). XML Schema Language, `http://www.w3.org/TR/xmlschema-0/`.
83. XPath (1999). XML Path Language (XPath) Version 1.0, `http://www.w3.org/TR/xpath`.
84. XPath2.0 (2005). XML Path Language (XPath) 2.0, `http://www.w3.org/TR/xpath20`.
85. XQuery (2005). XQuery 1.0: An XML Query Language, `http://www.w3.org/TR/xquery/`.
86. XSLT (1999). XSL Transformations (XSLT) Version 1.0, `http://www.w3.org/TR/xslt`.
87. XSLT2.0 (2003). XSL Transformations (XSLT) Version 2.0, `http://www.w3.org/TR/xslt20`.

This page is intentionally left blank

Chapter 6

Quotient Monoids and Concurrent Behaviours

Ryszard Janicki
Department of Computing and Software, McMaster University, Hamilton, Ontario, Canada L8S 4K1,
E-mail: `janicki@cas.mcmaster.ca`

Jetty Kleijn
LIACS, Leiden University, Leiden, 2300 RA The Netherlands,
E-mail: `kleijn@liacs.nl`

Maciej Koutny
School of Computing Science, Newcastle University, Newcastle upon Tyne NE1 7RU, United Kingdom,
E-mail: `maciej.koutny@ncl.ac.uk`

In this chapter, we discuss fundamental mathematical abstractions which can be used to capture and analyse operational semantics of concurrent systems. We focus our attention on the issues involved in the description of runs or behaviours of such systems. Assuming the discrete nature of system executions, in the most basic case, they can be represented as sequences of symbols, each symbol corresponding to the execution of a basic atomic action. By taking into account only the essential causal relationships between the executed actions one can then group together different runs which only differ by the ordering of causally unrelated actions. In the resulting model of Mazurkiewicz traces, each abstract execution (trace) is an equivalence class of the quotient monoid of sequential executions which can be represented by a (causal) partial order on the actions involved in these executions. Starting from this initial setting, the chapter then considers behaviours which are (step) sequences of sets of simultaneously executed actions, and takes into account other relationships between pairs of executed actions, such as weak causality. The resulting abstract behaviours can again be expressed in terms of suitable quotient monoids of step sequences or, equivalently, relational structures generalising causal partial orders. We then show how concrete system models coming from the Petri net domain can be treated using the above abstract notions of concurrent behaviour.

6.1 Introduction

Concurrent and distributed computing systems play an ever increasing role in the fast growing field of application of computer based technologies. At the same time, the complexity of such systems grows rapidly, making it very hard to guarantee the correctness of their ongoing operations, robustness, or resilience to security threats. A key issue is therefore to understand the behavioural characteristics of concurrent systems and, in particular, to provide appropriate mathematical abstractions for capturing their operational properties.

In the area of sequential computing, a successful formal capture of system behaviour can be obtained by specifying functional or relational dependencies between the generated inputs and outputs. Then it is also important, that for a system to be correct, it has to terminate for any given set of legal inputs. In the area of non-sequential computing, however, dynamic behaviours are not always adequately captured by functional input-output descriptions. Moreover, a concurrent system (such as a network router) may be considered as correct only if its operation never terminates. As a result, one may be interested in the modelling of ongoing evolutions of such systems at the interface with the environment, e.g., when communicating or reacting to external stimuli.

To model (ongoing) behaviour of systems one can simply use sequences of executed actions. For example, one can view a finite state machine as a generator of words (a language) over an alphabet of actions or events, yielding a powerful tool with numerous highly successful applications in almost every branch of Computer Science. Having said that, finite state machines (and other similar formalisms) are intrinsically sequential which means that they are far from being an ideal semantical framework for dealing with concurrent systems and their complex behaviours. For example, a sequential representation of system behaviour cannot be used to describe the result of action refinement for which the information about concurrency or independence may be of crucial importance. Problems like this one, were already well known in the 1970s when concurrent systems first became a focus of intensive scientific enquiry. What became clear was that sequential descriptions of system behaviours should be suitably augmented. A notable example is Milner's observational equivalence [35] which makes it possible to capture the exact moments in system executions when choices between alternative behaviours were decided, allowing one to analyse deadlock properties of concurrent designs. Another such example — and one which stands behind the theory presented in this chapter — are Mazurkiewicz traces [32] which provide as additional information, direct and explicit capture of causal dependencies between events.

The fundamental idea behind traces is that a single concurrent evolution can be observed by different sequential observers in different ways, and only the combination of all such partial observations provides a true representation of the underlying phenomenon. Still, different observations of the same run are not completely arbitrary, and share certain 'core' information which can be interpreted as the causality relationship between executed actions. Trace theory allows one to precisely identify all sequentialisations of a given concurrent behaviour. Formally, a trace is an equivalence class comprising all sequential observations of the same underlying concurrent run and to each trace a unique (causal) partial order of the executed actions can be associated. Moreover, given the alphabet of actions and the concurrency relationship between them, traces form a partially commutative quotient monoid with trace concatenation playing the role of the monoidal operation.

The original motivation of Mazurkiewicz was to use traces in order to analyse Petri nets [38, 40, 41], a widely used model of concurrent computation. Petri nets are an operational system model related to state machines and similar (sequential) behaviour generating devices. However, Petri nets are able to represent states in a distributed way, and model actions (state changes) in a purely local way involving only those parts of a distributed state which are directly affected. This allows one, in particular, to view their evolutions as partially ordered sets of executed actions. Since such partial orders can be interpreted as faithful recordings of causal relations between executed actions, there is a natural bridge between trace theory and the Petri net execution model.

Mazurkiewicz traces fit particularly well with the concurrent behaviours exhibited by the fundamental class of Elementary Net systems (EN-systems). There are however some aspects of concurrency that cannot adequately be modelled by partial orders alone (c.f. [14, 16]) and hence also not in terms of traces. Examples are the 'earlier than or simultaneous' (that is, 'not later than') and the 'earlier or later than (unordered), but not simultaneous' relationships [16], for which neither traces nor partial orders are expressive enough.

> Consider, for example, a priority system with three actions: a, b and c (with the priority of c higher than that of action b). Initially, a and b can be executed simultaneously, while c is blocked. However, upon its completion, action a makes action c enabled (i.e., c is causally dependent on a). As a result, there are two possible system executions involving all three actions, and bac (note that $\{a, b\}$ is a step in which actions a and b are executed simultaneously). If we were to look for a partial order underlying $\{a, b\}c$, we would fail as the sequence abc is not a valid system behaviour. To address this problem, we will consider structures richer than causal partial orders and, in this particular case, we will say that there is a 'weak causality' between b and a meaning that b earlier than or simultaneously with a, but not later than a.
>
> Another kind of relationship between actions can be observed when one considers two operations, d and f, updating a shared variable, and otherwise being completely independent. Then we have two potential executions, df and fd, meaning that they are not ordered, but a simultaneous execution $\{d, f\}$ is disallowed. In this case, we will consider them to be 'interleaved' and both df and fd will be corresponding to a single abstract concurrent history.

To overcome the limitations of traces, one may turn to generalisations. As such, the concepts of *comtrace* (combined trace) — based on causality and weak causality relationships — introduced in [17], and *g-comtrace* (generalised comtrace) — based on the weak causality and interleaving relationships — proposed in [21, 22], provide a suitable treatment for the above examples.

Comtraces and g-comtraces extend the original idea of traces in two directions. First, the evolutions of a concurrent system are represented by step sequences which are strings of finite *sets* representing simultaneously executed actions. Thus the underlying monoid is now generated by steps rather than individual actions. Second, the equivalence relation between step sequence observations is induced by systems of equations on step sequences rather than ordinary sequences. To express 'not later than', comtraces have, in addition to *simultaneity*, a *serialisability* relation which in general is not symmetric. It is applied to steps and can thus be used to define equations generating a quotient monoid. For general comtraces, the 'unordered, but not simultaneous' is expressed through a separate *interleaving* relation. The resulting quotient monoids are more expressive than the standard trace monoid yet, at the same time, they still enjoy algebraic properties similar to those exhibited in the standard case.

In this chapter, we bring together for the first time a whole range of issues and concepts involved in the modelling of complex concurrent behaviours through suitably defined causality structures and the corresponding treatment in the algebraic framework of quotient monoids. In general, the concern of (generalised) trace theory is how to add information to observations in order to convey the essence of causality between executed actions (i.e., the necessary ordering in the sense that cause must precede effect). We will discuss here Mazurkiewicz traces, comtraces, and generalised comtraces as language models of sophisticated concurrent behaviours as well as their relational counterparts (order structures), their corresponding (extended) Elementary Net models, and their mutual relationships.

Similarly to the relation between traces and partial orders, comtraces correspond to *stratified order structures* or (so-structures) [8, 15, 17, 18], and g-comtraces to *generalised stratified order structures* (gso-structures), introduced and studied in [11, 14]. (Note that both so-structures and gso-structures extend the standard causal partial orders if the underlying concurrent system does not exhibit features like priorities in the above example.) Stratified order structures have been successfully applied to model, e.g., inhibitor and priority systems and asynchronous races (see, e.g., [17, 24, 28]).

A main feature of our presentation which is is based, in particular, on [14] and [29], is the close correspondence of the more expressive quotient monoids to well motivated extensions of EN-systems, namely EN-systems with *inhibitor* arcs and EN-systems with *mutex* arcs which we will introduce in this chapter. In each case, one can lift the key properties established for Mazurkiewicz traces and EN-systems, providing a basis for new analysis techniques and verification tools, as well as extending and enhancing our *understanding* of concurrency-related phenomena like independence, unorderedness, and simultaneity. Crucial is that for these three elementary net models, the additional information to be represented in traces, comtraces, and generalised comtraces respectively, is indeed based on static (i.e., structural, graph theoretic) binary relationships between events (transitions), and does not depend on the current state of the net.

This chapter is organised in the following way. After a preliminary section on sets and languages, we introduce monoids and equational monoids which are later used to capture the essence of equivalent system executions. We then describe different kinds of (relational) order structures based on causality relationships between executed actions. The following section focuses on Mazurkiewicz traces and shows their relationship with causal partial orders. We then present a thorough discussion of comtraces, their algebraic properties, and show their correspondence to so-structures. A similar treatment is then applied to generalised comtraces and gso-structures. We finally consider Elementary Net systems which are generally regarded as the most fundamental class of Petri nets, and were indeed the model which inspired the introduction of traces. We investigate both sequential and non-sequential ways of executing them. The trace-based behaviour is obtained by taking sequential executions and combining them with the structural information about the dependencies between executed actions obtained from the graph structure of a net. That this approach is sound follows from the fact that the partial orders defined by traces coincide with the partial order semantics of nets represented by the non-sequential observations captured by operationally derived processes. This treatment is then repeated for two significant, and practically relevant, extensions of Elementary Net systems. The first extension consists in adding inhibitor arcs to the net, and the other in adding mutex arcs. In each case we demonstrate the necessary generalisations of the concept of action independence, leading to comtraces and generalised traces, respectively.

6.2 Preliminaries

In this section, we recall some well-known mathematical concepts and results that will be used throughout.

A *relational tuple* is a tuple $rel \overset{\mathrm{df}}{=} (X_1, \ldots, X_m, Q_1, \ldots, Q_n)$ where the X_i's are finite disjoint sets forming the *domain*, and the Q_i's are relations involving elements of the domain and perhaps some other elements (e.g., labels). In all cases considered later on, a relational tuple can be viewed as a graph of some sort and we will use the standard graphical conventions to represent its nodes (i.e., the elements of its domain), the various relationships between these nodes, and some particular characteristics of these nodes (e.g., labelling). We will often refer to the various components of a relational tuple using a subscript, e.g., X_{rel} or Q_{rel} for a relational tuple $rel \overset{\mathrm{df}}{=} (X, Q, \ell)$.

A particular issue linking together different relational tuples is the idea that what really matters is the *structures* they represent rather than the identities of the elements of their domains. A technical device which can be used to capture such a view is the following: two relational tuples, *rel* and *rel*′, are *isomorphic* if there is a bijection ψ from the domain of *rel* to the domain of *rel*′ such that if we replace throughout *rel* each element x in its domain by $\psi(x)$ then the result is *rel*′ (this is not strictly formal, but it should convey sufficient meaning to make the presentation clear). It is then standard to consider isomorphic relational tuples as indistinguishable.

Sets and Relations. $\mathbb{N}$ denotes the set of natural numbers including zero. The powerset of a set X is denoted by $\mathscr{P}(X)$, and the cardinality of a finite set X by $|X|$. Sets $X_1, \ldots X_n$ form a partition of a set X if they are non-empty disjoint subsets of X such that $X = X_1 \cup \ldots \cup X_n$.

A *labelling* ℓ for a set X is a function from X to a set of labels $\ell(X)$, and a *labelled set* is a pair (X, ℓ) where X is a set and ℓ is a labelling for X. The labelling is extended to finite sequences of elements of X by $\ell(x_1 \ldots x_n) \overset{\mathrm{df}}{=} \ell(x_1) \ldots \ell(x_n)$, and to finite sequences of subsets of X by $\ell(X_1 \ldots X_n) \overset{\mathrm{df}}{=} \ell(X_1) \ldots \ell(X_n)$.

Assumption 6.1. We *assume* throughout that all sets in this chapter are *labelled sets*, with the default labelling simply being the identity function. If the actual labelling is irrelevant for a particular definition or result, it may be omitted. Moreover, whenever it is stated that two domains are the same, we implicitly assume that their labellings are identical.

The composition $R \circ Q$ of two relations $R \subseteq X \times Y$ and $Q \subseteq Y \times Z$ comprises all pairs (x, z) in $X \times Z$ for which there is y in Y such that $(x, y) \in R$ and $(y, z) \in Q$.

Definition 6.2 (Relations). *Let R be a binary relation on a set X.*

- $R^{-1} \overset{\mathrm{df}}{=} \{(y, x) \mid (x, y) \in R\}$. *(reverse)*
- $R^0 = id_X \overset{\mathrm{df}}{=} \{(x, x) \mid x \in X\}$. *(identity relation)*
- $R^n \overset{\mathrm{df}}{=} R^{n-1} \circ R$. *(n-th power, $n \geq 1$).*
- $R^+ \overset{\mathrm{df}}{=} R^1 \cup R^2 \cup \ldots$. *(transitive closure)*
- $R^* \overset{\mathrm{df}}{=} R^0 \cup R^+$. *(reflexive transitive closure)*
- $R^{sym} \overset{\mathrm{df}}{=} R \cup R^{-1}$. *(symmetric closure)*

- *R is symmetric, reflexive, irreflexive, transitive if, respectively, $R = R^{-1}$, $id_X \subseteq R$, $id_X \cap R = \varnothing$, $R \circ R \subseteq R$.*
- *R is acyclic if R^+ is irreflexive.*

The restriction of a function $f : X \to Y$ to a subset Z of X is denoted by $f|_Z$, and the restriction of a relation $R \subseteq X \times Y$ to a subset Z of $X \times Y$ by $R|_Z$. The *domain* of R is given by $dom_R \overset{\mathrm{df}}{=} \{x \mid (x, y) \in R\}$ and its *codomain* by $codom_R \overset{\mathrm{df}}{=} \{y \mid (x, y) \in R\}$. We will often use the infix notation $x\,R\,y$ to denote that $(x, y) \in R$.

A binary relation is an *equivalence relation* if it is reflexive, symmetric, and transitive. If R is an equivalence relation on X, then $X/_R$ denotes the set of all equivalence classes of R, and $[\![x]\!]_R$ the equivalence class of R containing $x \in X$. (We may omit the subscript, if the relation is clear from the context.) Given an equivalence relation R on X and a function f defined for n-tuples of elements of X, it is often useful to lift f to n-tuples of equivalence classes of R by setting $f([\![x_1]\!], \ldots, [\![x_n]\!]) \overset{\mathrm{df}}{=} f(x_1, \ldots, x_n)$. Clearly, f is *well-defined* on $X/_R$ only if the value returned does not depend on the choice of the elements representing the equivalence classes of R.

Partial Orders. A relation is a *partial order* if it is *ir*reflexive and transitive.

Definition 6.3 (Partially ordered sets). *A (strictly)* partially ordered set *(or poset) $po \overset{\mathrm{df}}{=} (X, \prec)$ is a relational tuple consisting of a finite set X and a partial order $\prec$ on X. Two distinct elements x, y of X are* unordered*, $x \frown y$, if neither $x \prec y$ nor $y \prec x$. Moreover, $x \overset{\frown}{\prec} y$ if $x \prec y$ or $x \frown y$, and $x \simeq y$ if $x \frown y$ or $x = y$.*
po is total *(or linear) if all distinct elements of X are ordered, and* stratified *(or weak) if $\simeq$ is an equivalence relation.*

Figure 6.1 shows three example posets. Note that all total posets are also stratified.

A total poset *tpo* is a *linearisation* of a poset *po* with the same domain if $\prec_{po}$ is included in $\prec_{tpo}$. Similarly, a stratified poset *spo* is a *stratification* of a poset *po* with the same domain if $\prec_{po}$ is included in $\prec_{spo}$. We write *lin*(*po*) for the set of all linearisations of *po* and *strat*(*po*) for the set of all stratifications of *po*.

The *intersection* of a non-empty set of posets $\mathcal{PO}$ with the same domain X is given by $\bigcap \mathcal{PO} \overset{\mathrm{df}}{=} (X, \prec)$, where $\prec$ defined as $\bigcap\{\prec_{po} \mid po \in \mathcal{PO}\}$ is the

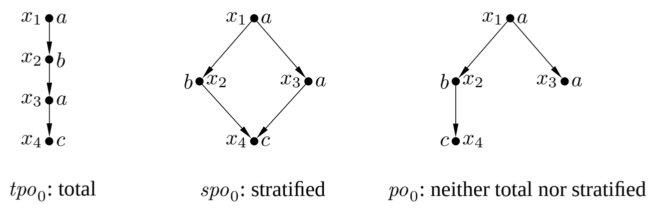


Fig. 6.1 Hasse diagrams of three posets showing also the labels (a, b and c) of their elements.

relation comprising all pairs (x, y) of elements of X such that $x \prec_{po} y$ for each poset po belonging to $\mathcal{PO}$.

Theorem 6.4. *$lin(po) \neq \varnothing$ and $po = \bigcap lin(po)$, for every poset po.*

The above result — Szpilrajn's Theorem [45] — states that every poset is uniquely determined by the intersection of all of its linearisations.[1] The same also holds for the set of its stratifications.

Proposition 6.5. *$strat(po) \neq \varnothing$ and $po = \bigcap strat(po)$, for every poset po.*

Proof. The first part and the $(\supseteq)$ inclusion follow from Thm. 6.4 and the observation that total posets are also stratified. The reverse inclusion follows from $\prec_{po} \subseteq \prec_{spo}$, for each $spo \in strat(po)$. □

The following result shows that each poset can be reconstructed from a single total extension if the incomparability relation is known.

Proposition 6.6. *$\prec_{tpo} \setminus \frown_{po} = (\prec_{tpo} \setminus \frown_{po})^+ = \prec_{po}$, for every poset po and a linearisation tpo of po.*

Proof. For all distinct $a, b \in dom_{po}$,

$$(a, b) \in \prec_{tpo} \setminus \frown_{po} \iff a \prec_{tpo} b \wedge (a \prec_{po} b \vee b \prec_{po} a) \iff a \prec_{po} b\ .$$

Hence $\prec_{tpo} \setminus \frown_{po} = \prec_{po}$. Moreover, since each partial order relation is transitive, we obtain that $\prec_{tpo} \setminus \frown_{po} = (\prec_{tpo} \setminus \frown_{po})^+$. □

Sequences and Step Sequences. Alphabets, sequences and step sequences are some of the main notions used for representing the behaviour of an evolving computing system:

[1]In the general case, when domains can be infinite, the proof requires the use of the Kuratowski-Zorn Lemma [7, 45].

- an *alphabet* E is a finite non-empty set of symbols (interpreted as events);
- a *sequence* (over E) is a finite string $a_1 \dots a_n$ of symbols (from E);
- a *step* (over E) is a non-empty subset of E; and
- a *step sequence* (over E) is a finite string $A_1 \dots A_n$ of steps (over E).

The empty (step) sequence, corresponding to the case $n = 0$, is denoted by λ.

As singleton sets can be identified with their only elements, sequences of symbols can be seen as special step sequences. We will take advantage of this by introducing some of the notions only for step sequences, and leaving their obvious specialisation for sequences implicit. Moreover, we will usually (but not always) drop the set brackets of singleton sets.

Definition 6.7 (Step sequences). *Let $u = A_1 \dots A_n$ and $v = B_1 \dots B_m$ be two step sequences.*

- $uv \overset{\text{df}}{=} A_1 \dots A_n B_1 \dots B_m$ *is the* concatenation *of u and v.*
- $len(u) \overset{\text{df}}{=} n$ *is the* length *of u.*
- $wgt(u) \overset{\text{df}}{=} |A_1| + \cdots + |A_n|$ *is the* weight *of u.*
- $alph(u)$ *comprises all symbols occurring within u.*
- $\#_a(u)$ *is the number of occurrences of a symbol a within u.*
- $occ(u)$ *is the set of* symbol occurrences *of u comprising all indexed symbols a^i with $a \in alph(u)$ and $1 \leq i \leq \#_a(u)$.*
- $pos_u(a^i) \overset{\text{df}}{=} \min\{j \mid \#_a(A_1 \dots A_j) = i\}$ *is the* position *of a^i within u.*
- $\ell(a^i) \overset{\text{df}}{=} a$ *is the default* label *of the symbol occurrence a^i.*

Consider $u = \{a,b\}b\{a,b,c\}\{c,d\}$ and $v = \{b,d\}\{b,c,d\}$. Then we have the following:

$$\begin{array}{llll}
alph(u) & = \{a,b,c,d\} & alph(v) & = \{b,c,d\} \\
len(u) & = 4 & len(v) & = 2 \\
wgt(u) & = 8 & wgt(v) & = 5 \\
\#_a(u) & = 2 & \#_b(u) & = 2 \\
pos_u(a^2) & = 3 & pos_v(d^1) & = 1 \\
\ell(a^2) & = a & \ell(d^1) & = d\,.
\end{array}$$

Moreover, we have $occ(u) = \{a^1, a^2, b^1, b^2, b^3, c^1, c^2, d^1\}$ as well as $occ(v) = \{b^1, b^2, c^1, d^1, d^2\}$.

Monoids. We will now outline a general approach aimed at introducing structure to the otherwise plain sets of sequences and step sequences by essentially grouping them into clusters of *equivalent* evolutions. The whole approach is underpinned by the notion of a monoid.

Definition 6.8 (Monoids). *A* monoid *is a triple* $\mathcal{M} \overset{\mathrm{df}}{=} (X, \circ, \mathbf{1})$ *where* X *is a (possibly infinite) set,* $\circ$ *is a binary operation on* X*, and* $\mathbf{1}$ *is an element of* X*, such that* $(a \circ b) \circ c = a \circ (b \circ c)$ *and* $a \circ \mathbf{1} = \mathbf{1} \circ a = a$*, for all* $a, b, c \in X$.
The monoid $(X, \circ, \mathbf{1})$ *for which there is a finite set* E *such that* $\mathbf{1} \in X \backslash E$ *and* $X \backslash \{\mathbf{1}\}$ *is the set of all elements that can be constructed from* E *using* $\circ$*, is called the monoid* generated *by* E *and* $\circ$.

We will be interested in two kinds of monoids, viz. monoids of sequences, and monoids of step sequences, over a given alphabet E.

If $\circ$ is the sequence concatenation operation, then $(E^*, \circ, \lambda)$ is the *free monoid of sequences* over E. In this case one can assume that the elements of E have no internal structure. As a result, the only readily available relationship between two elements of E is (in)equality, and the only operation we can apply to the sequences in E^* is concatenation. The situation becomes much more interesting if we are to consider step sequences.

Let $\mathbb{S}$ be a *step alphabet* given as a non-empty set of steps over an alphabet E. The monoid $(\mathbb{S}^*, \circ, \lambda)$ of step sequences, where $\circ$ is step sequence concatenation, is the *free monoid of step sequences* over $\mathbb{S}$. Since the elements of $\mathbb{S}$ are sets, one can — in addition to concatenation — use the standard set theoretic relationships and operators to manipulate them.

Definition 6.9 (Congruences and quotient monoids). *A* congruence *in a monoid* $\mathcal{M} = (X, \circ, \mathbf{1})$ *is an equivalence relation* $\sim$ *on* X *such that* $a \sim b$ *and* $c \sim d$ *implies* $a \circ c \sim b \circ d$*, for all* $a, b, c, d \in X$*. In such a case, the triple* $\mathcal{M}_\sim \overset{\mathrm{df}}{=} (X/_\sim, \hat{\circ}, [\![\mathbf{1}]\!])$ *with* $[\![a]\!] \hat{\circ} [\![b]\!] \overset{\mathrm{df}}{=} [\![a \circ b]\!]$*, for all* $a, b \in X$*, is the* quotient monoid *of* $\mathcal{M}$ *w.r.t. congruence* $\sim$.

Note that $\hat{\circ}$ is a well-defined operation, and that $\mathcal{M}_\sim$ is indeed a monoid. The mapping $\phi : X \to X/_\sim$ given by $\phi(a) \overset{\mathrm{df}}{=} [\![a]\!]$ is the *natural homomorphism* generated by the congruence $\sim$ (for more details see, e.g., [1]). The operation symbols $\circ$ and $\hat{\circ}$ are often omitted if this does not lead to confusion.

Equational Monoids. Quotient monoids defined by congruence relations provide a convenient way of introducing algebraic structure to sets of behaviours of evolving systems. In the approach presented in this chapter, a key role is played by quotient monoids defined by congruences induced by systems of equations on sequences and step sequences.

Let $\mathcal{M} = (X, \circ, \mathbf{1})$ be a monoid and let $EQ \overset{\text{df}}{=} \{\ x_1 = y_1\ \ldots\ x_n = y_n\ \}$, where $x_i, y_i \in X$ for $1 \leq i \leq n$, be a finite set of equations over $\mathcal{M}$. The *congruence defined by* EQ is the least congruence $\equiv$ in $\mathcal{M}$ such that $x_i \equiv y_i$, for every $i \leq n$. This congruence is also referred to as the EQ-congruence and denoted by $\equiv_{EQ}$, or simply $\equiv$ if no confusion can arise.
The quotient monoid $\mathcal{M}_{\equiv} \overset{\text{df}}{=} (X/_{\equiv}, \hat{\circ}, [\![\mathbf{1}]\!])$ is the *equational monoid* generated by EQ (for more details see, e.g., [21, 37]).

The next result demonstrates that EQ-congruence can be defined more directly using a binary relation $\approx_{EQ}$ on X (or simply $\approx$), comprising all pairs $(x, y) \in X \times X$ for which there are $u, w \in X$ and $1 \leq i \leq n$ such that the following hold: $x = u \circ x_i \circ w$ and $y = u \circ y_i \circ w$.

Proposition 6.10. *$\equiv_{EQ}$ is the reflexive, symmetric, and transitive closure of $\approx$.*

Proof. Clearly, $R \overset{\text{df}}{=} (\approx^{sym})^*$ is an equivalence relation. We now observe that if we have $t \approx^{sym} v$ then, for every $w \in X$, $t \circ w \approx^{sym} v \circ w$ and $w \circ t \approx^{sym} w \circ v$. Hence $t\ R\ v$ implies $t \circ w\ R\ v \circ w$ and $w \circ t\ R\ w \circ v$, for every $w \in X$. As a result, $t\ R\ v$ and $t'\ R\ v'$ together imply that $t \circ t'\ R\ v \circ t'\ R\ v \circ v'$, and so R is a congruence in $\mathcal{M}$.

Let $\sim$ be a congruence in $\mathcal{M}$ such that $x_i \sim y_i$, for every $i \leq n$. Clearly, $x \approx^{sym} y$ implies $x \sim y$. Hence, $x\ R\ y$ implies $x \sim y$. Thus, R also satisfies the minimality requirement and, as a result, is equal to $\equiv_{EQ}$. $\square$

We will now present three kinds of equational monoids which will be used in the rest of this chapter.

Partially Commutative Monoids. For a monoid $\mathcal{M} = (X, \circ, \mathbf{1})$ generated by E and $\circ$. and equation set $EQ \overset{\text{df}}{=} \{\ a_1 \circ b_1 = b_1 \circ a_1\ \ldots\ a_n \circ b_n = b_n \circ a_n\ \}$, where $a_1, \ldots, a_n, b_1, \ldots, b_n \in E$, the equational monoid $\mathcal{M}_{\equiv}$ is *partially commutative* [2]. In particular, when $\mathcal{M}$ is the free monoid of sequences over alphabet E and $\circ$ is concatenation, the elements of $\mathcal{M}_{\equiv}$ are called *Mazurkiewicz traces* [32] or simply *traces* and the equations $a_i \circ b_i = b_i \circ a_i$ are interpreted as signifying that events a_i and b_i are independent or concurrent.

Consider $E_0 \overset{\text{df}}{=} \{a, b, c\}$ and $EQ_0 \overset{\text{df}}{=} \{\ bc = cb\ \}$. Then, for example, we have $abcbca \equiv accbba$ as $abcbca \approx$

> $acbbca \approx acbcba \approx accbba$. Take the following three members of $E_0^*/_\equiv$: $\mathbf{y} \stackrel{\mathrm{df}}{=} [\![abc]\!] = \{abc, acb\}$, $\mathbf{x} \stackrel{\mathrm{df}}{=} [\![abcbca]\!] = \{abcbca, abccba, acbbca, acbcba, abbcca, accbba\}$ and $\mathbf{z} \stackrel{\mathrm{df}}{=} [\![bca]\!] = \{bca, cba\}$. Then $\mathbf{x} = \mathbf{y}\hat{\circ}\mathbf{z}$ as $abcbca = abc \circ bca$.

Absorbing Monoids. Let $(\mathbb{S}^*, \circ, \lambda)$ be a free monoid of step sequences with $\mathbb{S}$ being *subset closed*,[2] and let $\equiv$ be the *EQ*-congruence generated by a set of equations $EQ \stackrel{\mathrm{df}}{=} \{\ C_1 = A_1 \circ B_1\ \ldots\ C_n = A_n \circ B_n\ \}$, where each C_i belongs to $\mathbb{S}$ and A_i, B_i form a partition of C_i. Then the quotient monoid $(\mathbb{S}^*/_\equiv, \hat{\circ}, [\![\lambda]\!])$ is an *absorbing monoid of step sequences* [21].

> Consider $E_1 \stackrel{\mathrm{df}}{=} \{a, b, c\}$ and $\mathbb{S}_1 \stackrel{\mathrm{df}}{=} \{\{a, b, c\}, \{a, b\}, \{b, c\}, \{a, c\}, a, b, c\}$, as well as the following set of two equations:
>
> $$EQ_1 \stackrel{\mathrm{df}}{=} \{\ \{a, b, c\} = \{a, b\}c \quad \{a, b, c\} = a\{b, c\}\ \}\ .$$
>
> Then we have $\{a, b\}ca\{b, c\} \equiv a\{b, c\}\{a, b\}c$ since
>
> $$\begin{array}{rcl} \{a, b\}ca\{b, c\} & \approx & \{a, b, c\}a\{b, c\} \approx \{a, b, c\}\{a, b, c\} \\ & \approx & a\{b, c\}\{a, b, c\} \approx a\{b, c\}\{a, b\}c\ . \end{array}$$
>
> Take the following two members of $\mathbb{S}^*/_\equiv$:
>
> $$\mathbf{x} \stackrel{\mathrm{df}}{=} [\![\{a, b, c\}]\!] = \{\{a, b, c\}, \{a, b\}c, a\{b, c\}\}$$
>
> $$\mathbf{y} \stackrel{\mathrm{df}}{=} [\![\{a, b\}ca\{b, c\}]\!] = \left\{\begin{array}{ll} \{a, b, c\}\{a, b, c\} & \{a, b, c\}\{a, b\}c \\ \{a, b, c\}a\{b, c\} & \{a, b\}c\{a, b, c\} \\ \{a, b\}c\{a, b\}c & \{a, b\}ca\{b, c\} \\ a\{b, c\}\{a, b, c\} & a\{b, c\}\{a, b\}c \\ a\{b, c\}a\{b, c\} & \end{array}\right\}$$
>
> Then $\mathbf{y} = \mathbf{x}\hat{\circ}\mathbf{x}$ since we have $\{a, b\}ca\{b, c\} \equiv \{a, b, c\} \circ \{a, b, c\}$.

Partially Commutative Absorbing Monoids. Consider again a step sequence monoid $(\mathbb{S}^*, \circ, \lambda)$ with $\mathbb{S}$ being subset closed. Let $\equiv$ be the *EQ*-congruence generated by the following set of equations:

$$EQ \stackrel{\mathrm{df}}{=} \left\{\begin{array}{c} C_1 = A_1 \circ B_1\ \ldots\ C_n = A_n \circ B_n \\ D_1 \circ F_1 = F_1 \circ D_1\ \ldots\ D_m \circ F_m = F_m \circ D_m \end{array}\right\}$$

where the equations in the upper row are as in the previous case and, for each $j \leq m$, D_j and F_j are disjoint steps in $\mathbb{S}$ such that $D_j \cup F_j \notin \mathbb{S}$. Then the quotient monoid $(\mathbb{S}^*/_\equiv, \hat{\circ}, [\![\lambda]\!])$ is a *partially commutative absorbing monoid of step sequences.*

[2]That is, all non-empty subsets of a step $A \in \mathbb{S}$ belong to $\mathbb{S}$.

Consider $E_2 \overset{\text{df}}{=} \{a,b,c\}$ and $\mathbb{S}_2 \overset{\text{df}}{=} \big\{\{a,b\},\{b,c\},a,b,c\big\}$, as well as the set of equations $EQ \overset{\text{df}}{=} \big\{\ \{b,c\} = bc \quad \{a,b\} = ab \quad ac = ca\ \big\}$. Then $bacb \equiv \{b,c\}\{a,b\}$ since $bacb \approx bcab \approx \{b,c\}\{a,b\}$.

6.3 Partial Orders and Order Structures

In this section, we associate the monoids of the previous section to relational structures with the aim of arriving at a formal representation of the causal ordering underlying their elements. First we investigate sequences and step sequences themselves.

Clearly, sequences convey a linear (total) ordering of their elements. Conversely, the elements of a total poset $tpo = (X, \prec)$ can be listed as a (unique) sequence $x_1 \dots x_n$ such that $x_i \prec x_j$ *iff* $i < j$. The sequence *generated* by *tpo* is then defined as $seq(tpo) \overset{\text{df}}{=} \ell(x_1 \dots x_n)$ where ℓ is the labelling associated with X. Total orders are isomorphic *iff* the sequences they generate are the same. As observed above, every sequence u corresponds to a total poset *tpo* such that $seq(tpo) = u$. Since all such total posets are isomorphic it does not really matter which one is chosen and so it is convenient to single out one, based on the symbol occurrences of the sequence u. This total poset is called the *canonical* total poset of u and is defined as $cantpo(u) \overset{\text{df}}{=} (occ(u), \prec)$ where $\alpha \prec \beta$ if $pos_u(\alpha) < pos_u(\beta)$. Distinct sequences have distinct canonical total posets, and the sequence generated by the canonical total poset of a sequence is that sequence itself.

For step sequences, the situation is more complicated. Such sequences are not linear with respect to their symbol occurrences and so identifying a canonical partial order reflecting its structure requires some preparation. We begin by associating symbol occurrences to the steps in a step sequence.

For a step sequence $u = A_1 \dots A_k$, its *enumerated step sequence* is given by $\widehat{u} \overset{\text{df}}{=} \widehat{A}_1 \dots \widehat{A}_k$ where, for each $i \leq k$, $\widehat{A}_i \overset{\text{df}}{=} \big\{a^{\#_a(A_1 \dots A_i)} \mid a \in A_i\big\}$. Clearly, $\ell(\widehat{u}) = u$ which means that it is always possible to reconstruct the original step sequence from its enumerated version. Next, we define two relations $\prec_u$ and $\simeq_u$ on the symbol occurrences of u such that, for all $\alpha, \beta \in occ(u)$:

$$\begin{aligned} \alpha \prec_u \beta &\overset{df}{\iff} pos_u(\alpha) < pos_u(\beta) \\ \alpha \simeq_u \beta &\overset{df}{\iff} pos_u(\alpha) = pos_u(\beta)\ . \end{aligned}$$

Since $\simeq_u$ captures the 'unordered w.r.t. $\prec_u$ or equal' relationship, it follows

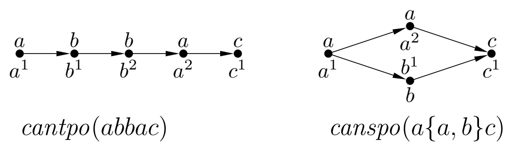


Fig. 6.2 Hasse diagram of the canonical posets of a sequence and step sequence. Both identities of nodes (symbol occurrences) and their labels are shown.

that $\mathit{canspo}(u) \overset{\mathrm{df}}{=} (\mathit{occ}(u), \prec_u)$ is a stratified poset, called the *canonical* stratified poset of u.

Proposition 6.11. *For every step sequence u, $\simeq_u$ is an equivalence relation.*

The total poset tpo_0 of Figure 6.1 is represented by the sequence *abac*. Figure 6.2 shows the canonical total poset of the sequence *abbac*. For the step sequence $u \overset{\mathrm{df}}{=} a\{a,b\}c$, the symbol occurrences are $\mathit{occ}(u) = \{a^1, a^2, b^1, c^1\}$, and its enumerated step sequence is $\widehat{u} = a^1\{a^2, b^1\}c^1$. Figure 6.2 shows the canonical total poset of u. Moreover, we have $\prec_u = \big\{(a^1,a^2),(a^1,b^1),(a^1,c^1),(a^2,c^1),(b^1,c^1)\big\}$ as well as $\simeq_u = \mathit{id}_{\mathit{occ}(u)} \cup \big\{(a^2,b^1),(b^1,a^2)\big\}$.

Conversely, a stratified poset *spo* is *represented* by the step sequence $\mathit{stepseq}(\mathit{spo}) \overset{\mathrm{df}}{=} B_1 \dots B_k$ where $B_1, \dots, B_k$ are the equivalence classes of $\simeq_{\mathit{spo}}$ and $\prec_{\mathit{spo}} = \bigcup_{i<j\leq k} B_i \times B_j$. The soundness of this notion follows immediately from the next result.

Proposition 6.12. *If spo is a stratified poset and A, B are two distinct equivalence classes of $\simeq_{\mathit{spo}}$, then either $A \times B$ or $B \times A$ is included in $\prec_{\mathit{spo}}$.*

Proof. We have $A \times B \subseteq \prec_{\mathit{spo}} \cup \prec_{\mathit{spo}}^{-1}$ since A and B are two distinct equivalence classes of $\simeq_{\mathit{spo}}$. Suppose that $a, b \in A$ and $c, d \in B$ are such that $a \prec c$ and $d \prec b$. Then, $a \prec d$ implies $a \prec b$ contradicting $a \frown b$, and $d \prec a$ implies $d \prec c$ contradicting $d \frown c$. Hence $A \times B$ either is included in $\prec_{\mathit{spo}}$ or in $\prec_{\mathit{spo}}^{-1}$. $\square$

The above result implies that the set of equivalence classes of the relation $\simeq_{\mathit{spo}}$ is totally ordered in a rather natural way. Propositions 6.11 and 6.12 are fundamental for the understanding of *equivalence* between stratified posets and step sequences. Since sequences are special cases of step sequences and total posets are special cases of stratified posets, the above results can be applied also to sequences and total posets. Then, for each sequence u, $\mathit{canspo}(u) = \mathit{cantpo}(u)$. Moreover, for every total poset tpo, we have $\mathit{seq}(\mathit{tpo}) = \ell(\mathit{stepseq}(\mathit{tpo}))$.

Canonical Extensions of Posets. It turns out that each poset has a unique stratified extension that can be interpreted as the 'greedy', i.e., maximally concurrent, execution consistent with the causality relation represented by this poset.

Proposition 6.13. *For every poset po, there is a unique stratification spo with $\mathit{stepseq}(spo) = B_1 \dots B_k$ such that, for all $i \geq 2$ and $b \in B_i$, there is $a \in B_{i-1}$ satisfying $a \prec_{po} b$.*

Proof. Let $S(po)$ be the set of all stratifications of po for which the property from the formulation of this proposition holds. We will show that $|S(po)| = 1$, by induction on the size of X_{po}. In the base case, $X_{po} = \varnothing$, we have $S(po) = \{po\}$ and so the property holds.

In the induction step, let B be the set of all minimal elements of po (i.e., $a \in B$ if there is no b such that $b \prec_{po} a$), and let po' be po restricted to $X_{po} \backslash B$. Then:

$$\begin{array}{lcl} B_1 \dots B_k \in \mathit{stepseq}(S(po)) & \Longrightarrow & B_1 = B \ \wedge \ B_2 \dots B_k \in \mathit{stepseq}(S(po')) \\ B_1 \dots B_k \in \mathit{stepseq}(S(po')) & \Longrightarrow & BB_1 \dots B_k \in \mathit{stepseq}(S(po)) \ . \end{array}$$

Both implications follow from the definitions of $S(po)$ and of the minimal elements of a poset. Then the first implication can be used to show that $|S(po)| \leq 1$, and the second that $|S(po)| > 0$. Note that the induction hypothesis can be applied to $S(po')$ since $B \neq \varnothing$ as X_{po} is finite and non-empty. $\square$

The poset from Prop. 6.13 will be denoted by $\mathit{canstratposet}(po)$ and referred to as the *canonical stratified extension* of po. It will play a key role in establishing a connection between equational monoids and their relational counterparts.

Defining a unique total extension of a poset is more difficult. We use a somewhat complicated mechanism to achieve the desired result which resorts to an additional total order in cases when the canonical stratified extension cannot order two elements.

Formally, given a poset po and a total poset tpo with the same domain as po, the *canonical total extension of po w.r.t. tpo* is the linearisation tpo'

of *po*, denoted by $cantotalposet_{tpo}(po)$, such that for all $a, b \in X_{po}$,

$$a \prec_{tpo'} b \iff a \prec_{canstratposet(po)} b \vee (a \frown_{canstratposet(po)} b \wedge a \prec_{tpo} b) .$$

In other words, elements belonging to the same equivalence class of $\frown_{spo}$ are ordered according to *tpo*, and otherwise the ordering is inherited from *spo*.

For the poset po_0 of Figure 6.1, $canstratposet(po_0) = spo_0$. Moreover, if we take *tpo* with $x_2 \prec_{tpo} x_3$, then we have the following: $cantotalposet_{tpo}(po_0) = cantotalposet_{tpo}(spo_0) = tpo_0$.

Relational Invariants of Sets of Posets. We will now consider relational representations of sets of posets. Throughout this subsection we use Δ to denote a set of posets with the same domain X and belonging to a class of posets $\mathbb{O}$. In particular, we will consider the class $\mathbb{TO}$ of the total posets and the class $\mathbb{SO}$ of the stratified posets.[3]

Definition 6.14 (Relational invariants). *A* relational invariant *over* Δ *is any relation*

$$inv_\Delta \overset{\text{df}}{=} \big\{(x, y) \in X \times X \mid x \neq y \ \wedge\ \forall z \in \Delta :\ inv\big\}$$

where inv is a well-formed propositional formula built from the standard logic connectives and three basic formulas: $x \prec_z y$, $y \prec_z x$ *and* $x \frown_z y$.

Each relational invariant describes a fundamental relationship between two elements of X which is common to all the posets included in Δ. For example, $(x \prec_z y)_\Delta$ comprises all pairs (a, b) of elements of X such that a precedes b in every poset z belonging to Δ. We will be particularly interested in four relational invariants:

$$\begin{array}{ll} \rightleftharpoons_\Delta \overset{\text{df}}{=} (x \prec_z y \ \vee\ y \prec_z x)_\Delta & \sqsubset_\Delta \overset{\text{df}}{=} (x \prec_z y \ \vee\ x \frown_z y)_\Delta \\ \prec_\Delta \overset{\text{df}}{=} (x \prec_z y)_\Delta & \bowtie_\Delta \overset{\text{df}}{=} (x \frown_z y)_\Delta \ . \end{array}$$

In general, knowing explicitly all invariant relationships is not necessary, for example, $(y \prec_z x)_\Delta$ is the reverse of $(x \prec_z y)_\Delta$. We call a subset of relational invariants an *invariant representation* of Δ if this subset uniquely identifies all the remaining relational invariants.

Proposition 6.15. *Each relational invariant is equal to one of the following:*

$$\varnothing \qquad \rightleftharpoons_\Delta \qquad \sqsubset_\Delta \qquad \sqsubset_\Delta^{-1} \qquad \prec_\Delta \qquad \prec_\Delta^{-1} \qquad \bowtie_\Delta \qquad X \times X \backslash id_X$$

Moreover, $\rightleftharpoons_\Delta$ *and* $\sqsubset_\Delta$ *form an invariant representation of* Δ.

[3]A discussion of other classes of posets can be found in [14].

Proof. The first part follows directly from the definitions, and the second from an observation that $\prec_\Delta$ and $\bowtie_\Delta$ are respectively equal to $\rightleftharpoons_\Delta \cap \sqsubset_\Delta$ and $\sqsubset_\Delta \cap \sqsubset_\Delta^{-1}$. $\square$

Hence, in order to get a complete information about the relational invariants, one can always take $\rightleftharpoons_\Delta$ and $\sqsubset_\Delta$, or simply $\prec_\Delta$ if $\mathbb{O} = \mathbb{TO}$ as in this case $\rightleftharpoons_\Delta$ and $\sqsubset_\Delta$ are respectively equal to $\prec_\Delta \cup \prec_\Delta^{-1}$ and $\prec_\Delta$.

The *invariant closure* $\Delta_{\mathbb{O}}^{icl}$ of Δ in $\mathbb{O}$ comprises all posets $po \in \mathbb{O}$ with domain X such that $inv_\Delta = inv_{\Delta\cup\{po\}}$, for every formula inv as in Defn. 6.14. We also say that Δ is *closed* in $\mathbb{O}$ if $\Delta = \Delta_{\mathbb{O}}^{icl}$ (note that $\Delta \subseteq \Delta_{\mathbb{O}}^{icl}$ always holds). Intuitively, a poset po belongs to $\Delta_{\mathbb{O}}^{icl}$ if it obeys all relationships captured by the relational invariants of Δ.

Proposition 6.16. *A poset $po \in \mathbb{O}$ with domain X belongs to $\Delta_{\mathbb{O}}^{icl}$* iff

$$\begin{array}{lcl} x \rightleftharpoons_\Delta y & \Longrightarrow & x \prec_{po} y \vee y \prec_{po} x \\ x \sqsubset_\Delta y & \Longrightarrow & x \prec_{po} y \vee x \frown_{po} y \end{array}$$

for all $x, y \in X$. A total poset tpo with domain X belongs to $\Delta_{\mathbb{TO}}^{icl}$ iff

$$x \prec_\Delta y \Longrightarrow x \prec_{tpo} y$$

for all $x, y \in X$.

Proof. Follows from the second part of Prop. 6.15, and the observation that $\prec_\Delta$ is an invariant representation of Δ if $\mathbb{O} = \mathbb{TO}$. $\square$

Clearly, $(X, \prec_\Delta)$ is a poset. Hence, if Δ is a set of total posets which is closed in $\mathbb{TO}$, then $\Delta_{\mathbb{TO}}^{icl} = lin(X, \prec_\Delta)$ and so Δ is completely represented by $(X, \prec_\Delta)$. In this context, Thm. 6.4 (Szpilrajn's Theorem) implies that, for every poset po, the set $lin(po)$ is closed in the class of total posets.

In concurrency theory, when system runs are modelled by sequences, a closed set of total posets is interpreted as a grouping of all equivalent concurrent runs (executions) of some concurrent *history* (or behaviour). Then, e.g., $\Delta = \{abc, cba\}$ does *not* correspond to a concurrent history. Indeed, since the intersection of the orders induced by abc and cba is empty, we have $\prec_\Delta = \varnothing$. As a consequence, there is no causal relationship between a, b, and c. This means that, e.g., bca is also a possible run, contradicting $bca \notin \Delta$. However, $\Delta_{\mathbb{TO}}^{icl} = \big\{abc, bac, acb, bca, cab, aba\big\}$ can be considered as a concurrent history.

If Δ is to be interpreted as a concurrent history then, depending on the assumed model of concurrency, some additional constraints may have to be added. For example, the following 'diagonal rule' — or 'diamond property' — by which simultaneity is the same as the possibility to occur in any order:

$$\begin{aligned}\forall x, y \in X : &(\exists po \in \Delta : \ x \frown_{po} y) \\ &\Longleftrightarrow \\ &(\exists po \in \Delta : \ x \prec_{po} y) \ \wedge \ (\exists po \in \Delta : \ y \prec_{po} x) \ .\end{aligned} \qquad (\pi_8)$$

Constraints like π_8 — called *paradigms* in [14, 16] — are essentially suppositions or statements about the intended treatment of simultaneity.

Proposition 6.17. *If Δ belongs to $\mathbb{SO}$ and satisfies π_8, then a stratified poset spo with domain X belongs to $\Delta^{icl}_{\mathbb{SO}}$* iff

$$x \prec_\Delta y \implies x \prec_{spo} y$$

for all $x, y \in X$. Moreover, $\Delta^{icl}_{\mathbb{SO}} = strat(X, \prec_\Delta)$.

Proof. The first part follows since in this case $\rightleftharpoons_\Delta$ and $\sqsubset_\Delta$ are respectively equal to $\prec^{sym}_\Delta$ and $\prec_\Delta$. The second part follows from the first part and $\mathbb{O} = \mathbb{SO}$. □

Hence a set of stratified posets Δ, closed in $\mathbb{SO}$ and satisfying π_8, is completely represented by the poset $(X, \prec_\Delta)$. However this is not, in general, true if Δ does not satisfy π_8. In this context, Prop. 6.5 implies that, for every poset *po*, the set $strat(X, \prec_{po})$ is closed in $\mathbb{SO}$.

To express the relations 'not later than' and 'unordered, but not necessarily simultaneous', one has to weaken paradigm π_8. The 'not later than' relation can be modelled by dropping the requirement that simultaneity should imply unorderedness leading to the following paradigm:

$$\begin{aligned}\forall x, y \in X : &(\exists po \in \Delta : \ x \prec_{po} y) \ \wedge \ (\exists po \in \Delta : \ y \prec_{po} x) \\ &\Longrightarrow \\ &(\exists po \in \Delta : \ x \frown_{po} y) \ .\end{aligned} \qquad (\pi_3)$$

Proposition 6.18. *If Δ satisfies π_3 then $\prec_\Delta$ and $\sqsubset_\Delta$ form an invariant representation of Δ.*

Proof. Follows from the fact that π_3 implies that $\rightleftharpoons_\Delta$ is equal to $\prec^{sym}_\Delta$. □

The symmetric counterpart of π_3:

$$\begin{aligned}\forall x, y \in X : &(\exists po \in \Delta : \ x \frown_{po} y) \\ &\Longrightarrow \\ &(\exists po \in \Delta : \ x \prec_{po} y) \ \wedge \ (\exists po \in \Delta : \ y \prec_{po} x)\end{aligned} \qquad (\pi_6)$$

does not simplify theory too much. If Δ satisfies π_6 but not π_3, we need both $\rightleftharpoons_\Delta$ and $\sqsubset_\Delta$ to form in the general case an invariant representation of Δ [14].

Stratified Order Structures. In this subsection we add a second ordering relation to partial orders that can be used to capture the concept of the 'not later than' relationship between events of a concurrent history.

Definition 6.19 (Stratified order structure). *A* stratified order structure *(or so-structure) is a relational structure $S \overset{\text{df}}{=} (X, \prec, \sqsubset)$ where $\prec$ (*causality*) and $\sqsubset$ (*weak causality*) are binary relations on X such that, for all $a, b, c \in X$:*

S1: $a \not\sqsubset a$ *S3:* $a \sqsubset b \sqsubset c \wedge a \neq c \implies a \sqsubset c$

S2: $a \prec b \implies a \sqsubset b$ *S4:* $a \sqsubset b \prec c \vee a \prec b \sqsubset c \implies a \prec c$.

The axioms (S1)–(S4) imply that $\prec$ is a partial order relation, and that $a \prec b$ implies $b \not\sqsubset a$. The relation $\prec$ represents the 'earlier than' relationship on the domain of S, and the relation $\sqsubset$ the 'not later than' relationship. The four axioms model the mutual relationship between 'earlier than' and 'not later than' relations, when system runs are modelled by step sequences (stratified posets).

So-structures were independently introduced in [8] and [15] (where the axioms were slightly different from the above, though equivalent). Their comprehensive theory has been presented in [18], and they have been successfully used, e.g., to model inhibitor and priority systems, asynchronous races and synthesis problems (see, e.g., [28, 24]).

The adjective 'stratified' is motivated by the following result [16].

Proposition 6.20. *Let Δ be a non-empty set of stratified posets with the same domain X. Then $S_\Delta \overset{\text{df}}{=} (X, \prec_\Delta, \sqsubset_\Delta)$ is an so-structure. Moreover, for every stratified poset spo, $S_{spo} \overset{\text{df}}{=} (X_{spo}, \prec_{spo}, \sqsubset_{spo})$ with $\sqsubset_{spo} \overset{\text{df}}{=} \prec_{spo} \cup \frown_{spo}$, is an so-structure.*

Conversely, so-structures can be extended to stratified posets.

Definition 6.21 (Stratified poset extension). *Let $S = (X, \prec, \sqsubset)$ be an so-structure. A stratified poset spo is a* stratified poset extension *of S if it has the same domain and, for all $a, b \in X$:*

$$a \prec b \quad \implies \quad a \prec_{spo} b \quad \text{and} \quad a \sqsubset b \quad \implies \quad a \sqsubset_{spo} b\ .$$

We denote this by $spo \in ext(S)$.

According to Szpilrajn's Theorem, posets can be reconstructed by intersecting their linearisations. A similar result holds for so-structures and their stratified poset extensions [18].

Theorem 6.22. *If S is an so-structure then $ext(S) \neq \varnothing$ and*

$$S = \big(X_S, \bigcap_{spo \in ext(S)} \prec_{spo}, \bigcap_{spo \in ext(S)} \sqsubset_{spo}\big) .$$

The set of stratified poset extensions of an so-structure always satisfies paradigm π_3 (see [16]). This is a useful characteristic which simplifies proofs but, at the same time, somewhat restricts potential applications.

Theorem 6.23. *If a and b are two elements of an so-structure S then:*

$$\left.\begin{array}{l} \exists spo \in ext(S) :\ a \prec_{spo} b \\ \exists spo \in ext(S) :\ b \prec_{spo} a \end{array}\right\} \implies \exists spo \in ext(S) :\ a \frown_{spo} b .$$

Whenever Δ is closed in $\mathbb{SO}$, the class of stratified posets, and Δ satisfies π_3, then $\Delta = ext(X_\Delta, \prec_\Delta, \sqsubset_\Delta)$.

Corollary 6.24. *If Δ belongs to $\mathbb{SO}$ and Δ satisfies π_3, then*

$$\Delta^{icl}_{\mathbb{SO}} = ext(X_\Delta, \prec_\Delta, \sqsubset_\Delta) .$$

Hence a set of stratified posets Δ, closed in $\mathbb{SO}$, and satisfying π_3 is completely represented by the so-structure $S_\Delta \overset{\mathrm{df}}{=} (X_\Delta, \prec_\Delta, \sqsubset_\Delta)$. Moreover, by Thm. 6.22, we have that for every stratified poset structure S, the set $ext(S)$ is closed in the class of stratified posets. However, this no longer holds if Δ fails to satisfy π_3.

Generalised Stratified Order Structures. Stratified order structures can adequately model concurrent histories when π_3 is satisfied. However, when system runs are defined as stratified posets and π_3 is not satisfied, one needs more general so-structures, as introduced in [11] and then analysed in [14].

Definition 6.25 (Gso-structure). *A* gso-structure *is a relational tuple $G \overset{\mathrm{df}}{=} (X, \rightleftharpoons, \sqsubset)$ such that the relation $\rightleftharpoons$ is symmetric and irreflexive, $\sqsubset$ is irreflexive, and $S_G \overset{\mathrm{df}}{=} (X, \rightleftharpoons \cap \sqsubset, \sqsubset)$ is an so-structure (*induced *by G).*

The *commutativity* relationship $\rightleftharpoons$ represents the 'earlier than or later than, but never simultaneous' relationship, while $\sqsubset$ again represents the 'not later than' relationship.

Definition 6.26 (Stratified poset extensions). *Let $G = (X, \rightleftharpoons, \sqsubset)$ be a gso-structure. A stratified poset spo is a* stratified poset extension *of G*

if it has the same domain and, for all $a, b \in X_G$:

$$a \rightleftharpoons b \implies a \prec_{spo}^{sym} b \quad \textit{and} \quad a \sqsubset b \implies a \sqsubset_{spo} b\,.$$

We denote this by $spo \in ext(G)$.

We then have the following [16].

Proposition 6.27. *For every non-empty set Δ of stratified posets with the same domain X, $G_\Delta \overset{\mathrm{df}}{=} (X, \rightleftharpoons_\Delta, \sqsubset_\Delta)$ is a gso-structure. Moreover, for every stratified poset spo, $G_{spo} \overset{\mathrm{df}}{=} (X, \prec_{spo}^{sym}, \sqsubset_{spo})$ is a gso-structure.*

Each gso-structure can be uniquely reconstructed from its stratified poset extensions [11, 14].

Theorem 6.28. *If G is a gso-structure then $ext(G) \neq \varnothing$ and*

$$G = \Big(X_G, \bigcap_{spo \in ext(G)} \prec_{spo}^{sym}, \bigcap_{spo \in ext(G)} \sqsubset_{spo}\Big)\,.$$

It turns out that gso-structures do *not* have an equivalent of Thm. 6.23 which tends to make proofs more difficult, but they can model the most general concurrent behaviours in the case that observations are modelled by stratified posets (step sequences) [14].

In a concurrency framework, the results of this subsection could be interpreted as follows. If Δ is closed in $\mathbb{SO}$, the class of stratified posets, then we have that $\Delta = ext(X_\Delta, \rightleftharpoons_\Delta, \sqsubset_\Delta)$ and so Δ is completely represented by the generalised stratified order structure $G_\Delta \overset{\mathrm{df}}{=} (X, \rightleftharpoons_\Delta, \sqsubset_\Delta)$. Moreover, by Thm. 6.28, we have that, for every generalised stratified order structure G, the set $ext(G)$ is closed in the class of stratified posets. In this case no constraint on the structure of Δ is assumed (paradigm π_1 in the terminology of [16]).

6.4 Mazurkiewicz Traces

Sequences represent a purely sequential view of executed actions. As such, no further information is provided about the intrinsic dependencies among their actions and the resulting necessary ordering of their occurrences. The introduction of traces starts from the definition of a concurrency alphabet, which simply states which pairs of symbols represent independent actions (i.e., not interfering with each other) and should be treated as concurrent.

Monoids of Mazurkiewicz traces (or traces) are equational, partially commutative, monoids of sequences. The theory of trace monoids has been developed and applied within diverse areas, such as combinatorics [2] and, in particular, concurrency theory [4, 32]. Applications of traces in concurrency theory are motivated mainly by the fact that traces are the sequence counterpart of posets, and thus have the ability to model causality semantics.

Let E be an alphabet, $\mathcal{M} \overset{\text{df}}{=} (E^*, \circ, \lambda)$ be the free monoid of sequences generated by E, and *ind* be an irreflexive and symmetric binary relation on E, called *independence* relation. Then the pair $\Gamma \overset{\text{df}}{=} (E, \mathit{ind})$ will be called a *trace alphabet.* For such a trace alphabet Γ, we define the set of equations:

$$EQ_{\mathit{ind}} \overset{\text{df}}{=} \{\ ab = ba\ \mid (a,b) \in \mathit{ind}\ \}$$

and we refer to $\equiv_{\mathit{ind}}$, the congruence defined by EQ_{ind}, as a *trace congruence.* The partially commutative monoid $\mathcal{M}_{\equiv_{\mathit{ind}}}$ is also called a *trace monoid.* A simple trace monoid was already given for E_0 and EQ_0.

We will now discuss the concept of a canonical form for the traces from $\mathcal{M}_{\equiv_{\mathit{ind}}}$. First, a sequence $x = a_1 \dots a_k \in E^*$ is *fully commutative* if $(a_i, a_j) \in \mathit{ind}$ for all $i \neq j$. Similar to the use of a total poset to identify a canonical total extension of a poset, we assume also that we have a total poset *lexpo* on E extended to a total (lexicographical) ordering of E^*.

A sequence $x \in E^*$ is in (Foata) *canonical form* (w.r.t. *ind* and *lexpo*), if $x = \lambda$ or $x = x_1 \dots x_n$, with each $x_i \in E^* \backslash \{\lambda\}$, is such that:

- each x_i is fully commutative and minimal w.r.t. *lexpo* among all sequences $u \in E^*$ satisfying $\mathit{occ}(x_i) = \mathit{occ}(u)$; and
- for all $i < n$ and $a \in E$ occurring in x_{i+1}, there is $b \in E$ occurring in x_i such that $(a,b) \notin \mathit{ind}$.

If x is in canonical form, then x is the (Foata) *canonical representation* of $[\![x]\!]$. The following result [2, 20] justifies the terminology.

Theorem 6.29. *Every trace contains exactly one sequence in canonical form.*

Relationship with Partial Orders. We will now show how a trace $[\![x]\!]$ can be represented by a poset. First, since $\mathit{occ}(x) = \mathit{occ}(y)$, for every $y \in [\![x]\!]$, we can define $\mathit{occ}([\![x]\!]) \overset{\text{df}}{=} \mathit{occ}(x)$. Furthermore, we let

$$\mathit{cantpo}([\![x]\!]) \overset{\text{df}}{=} \{\mathit{cantpo}(y) \mid y \in [\![x]\!]\}$$

consist of all canonical total posets associated with the elements of the trace. Now, let ρ_y be the relation on the symbol occurrences of $y \in [\![x]\!]$ such that

$$(a^i, b^j) \in \rho_y \iff a^i \prec_{\mathit{cantpo}(y)} b^j\ \wedge\ (a,b) \notin \mathit{ind}\ .$$

Such a relation is always acyclic, and so $seqpo(y) \overset{\text{df}}{=} (occ(y), \rho_y^+)$ is a poset derived from the ordering of dependent symbol occurrences in y. Then we define:

$$trpo([\![x]\!]) \overset{\text{df}}{=} \bigcap\{seqpo(y) \mid y \in [\![x]\!]\}$$

and call it the poset *generated* by the trace $[\![x]\!]$.

For the trace $[\![abcbca]\!]$ over E_0 and EQ_0, the Hasse diagram of the generated poset $trpo([\![abcbca]\!])$ with all the labels shown looks as in Figure 6.3.

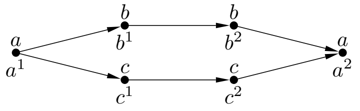


Fig. 6.3 Poset generated by a trace.

The proofs of the next three theorems are simplified versions of similar results presented later on for *comtraces*. Therefore, we will here only provide pointers to these results and their proofs.

The first theorem states that all elements in a trace have the same ordering of dependent symbol occurrences. Consequently, the poset generated by a trace is simply the poset of any single representative of the trace. Moreover, the linearisations of any such poset give exactly all elements of the trace.

Theorem 6.30. *Let $[\![x]\!]$ be a trace and $y \in [\![x]\!]$.*

(1) $\rho_x = \rho_y$.
(2) $trpo([\![x]\!]) = seqpo(y)$.
(3) $lin(trpo([\![x]\!])) = cantpo([\![x]\!])$.

Proof. Part (1) follows from Lem. 6.49 and its proof, part (2) follows from Thm. 6.53(3) and its proof, and part (3) follows from Thm. 6.53(2) and its proof. □

Conversely, when unorderedness is viewed as independence, the words generated by the linearisations of a poset form a trace.

Theorem 6.31. *For each poset po with the identity function labelling its domain, the set of sequences $\mathbf{t}_{po} \overset{\text{df}}{=} \{seq(tpo) \mid tpo \in lin(po)\}$ is a trace over the trace alphabet $\Gamma = (X_{po}, \simeq_{po})$.*

Proof. From Thm. 6.57 and its proof. □

Finally, it can be shown that the (Foata) canonical representation of a trace corresponds to the canonical total extension of the poset generated by that trace. For a total poset *lexpo* with domain E, we extend *lexpo* to the set of symbol occurrences $occ(x)$ of a word $x \in E^*$ as follows: $a^i \prec_{lexpo} b^j$ if $a \prec_{lexpo} b$; and $a^i \prec_{lexpo} a^j$ if $i < j$.

Theorem 6.32. *For every trace* $\mathbf{t}$ *and its canonical representation* x, $cantpo(x) = cantotalposet_{lexpo}(trpo(\mathbf{t}))$.

Proof. From Prop. 6.51 and its proof. □

Together with Thm. 6.30 this implies that given a total order on its alphabet, the (Foata) canonical representation of a trace can be read off directly from any single representative.

It is worth noting at this point, that by the above results, investigating a single element of a trace is often sufficient. This then leads to more efficient techniques for analysis and manipulation of traces. Calculating, e.g., the poset $trpo(\llbracket x \rrbracket)$ generated by trace $\llbracket x \rrbracket$ directly from the definition may in the worst case involve exponentially many elements of $\llbracket x \rrbracket$, whereas the complexity of calculating $\prec_y$, for $y \in \llbracket x \rrbracket$, is dominated by just the complexity $O(len(y)^3)$ of calculating the transitive closure of ρ_y.

6.5 Comtraces

Traces and partial orders represent independence through an absence of ordering (including possible simultaneity) between individual events. Consequently, these models are not expressive enough to capture the possibility of one event occurring 'not later than' another one, meaning that the first event may occur 'earlier than or simultaneous with' the second event. To overcome this limitation, comtraces are defined in terms of step sequences with each step representing simultaneity of the events in that step. In addition, a serialisability relation (in general not symmetric) describes if and how simultaneous pairs of events may be sequentialised, i.e., occur one after the other. Thus the underlying monoid is generated by steps and the simultaneity and serialisability relations are applied to steps to define equations generating the quotient monoid of comtraces.

Throughout this section, let E be an alphabet and $ser \subseteq sim \subset E \times E$ be two relations respectively called *serialisability* and *simultaneity* such that the relation *sim* is irreflexive and symmetric. Then $\Theta \overset{\text{df}}{=} (E, \mathit{sim}, \mathit{ser})$ is a *comtrace alphabet*. Intuitively, if $(a,b) \in \mathit{sim}$ then, whenever a and b both occur, they occur simultaneously, whereas $(a,b) \in \mathit{ser}$ means that in addition a may occur before b (with both executions being equivalent). The set of all (potential) steps over Θ is then defined as the following subset-closed step alphabet:

$$\mathbb{S} \overset{\text{df}}{=} \big\{A \mid \varnothing \neq A \subseteq E \ \wedge \ \forall a \neq b \in A : \ (a,b) \in \mathit{sim}\big\} \ .$$

The *comtrace congruence* over Θ is the EQ-congruence $\equiv$ defined by the equations

$$EQ \overset{\text{df}}{=} \big\{ \ A = BC \ \mid A = B \cup C \in \mathbb{S} \ \wedge \ B \times C \subseteq \mathit{ser} \ \big\} \ .$$

Since *ser* is irreflexive, for each equation $A = BC$ in EQ, we have $B \cap C = \varnothing$.

Definition 6.33 (Comtraces). $(\mathbb{S}^*/_{\equiv}, \hat{\circ}, [\![\lambda]\!])$ *is the (equational) monoid of* comtraces *over* Θ.

Note that $(\mathbb{S}^*/_{\equiv}, \hat{\circ}, [\![\lambda]\!])$ is an absorbing monoid. By Prop. 6.10, the comtrace congruence relation can be re-defined in a non-equational form, as follows.

Corollary 6.34. *Let* $\approx$ *be the relation comprising all pairs* (u,v) *of step sequences in* $\mathbb{S}^*$ *such that* $u = wAz$ *and* $v = wBCz$*, where* w, z *are sequences in* $\mathbb{S}^*$ *and* A*,* B*,* C *are steps in* $\mathbb{S}$ *satisfying* $B \cup C = A$ *and* $B \times C \subseteq \mathit{ser}$*. Then* $\equiv$ *is equal to* $(\approx^{sym})^*$*.*

Let $E \overset{\text{df}}{=} \{a,b,c\}$ where a, b and c are actions respectively representing three assignments: $x \leftarrow x+y$, $x \leftarrow y+2$ and $y \leftarrow y+1$. If simultaneous reading is allowed, then b and c can be performed simultaneously, and the simultaneous execution of b and c gives the same outcome as executing b followed by c. We can therefore set $\mathit{sim} \overset{\text{df}}{=} \{(b,c),(c,b)\}$ and $\mathit{ser} \overset{\text{df}}{=} \big\{(b,c)\big\}$ and then we obtain that $\mathbb{S} = \{a,b,c,\{b,c\}\}$ and $EQ = \{ \ \{b,c\} = bc \ \}$. As a result we have that, for example, $[\![a\{b,c\}]\!] = \{a\{b,c\}, abc\}$ is a comtrace which does not include acb.

Even though traces are quotient monoids of sequences and comtraces are quotient monoids of step sequences with steps being used in the definition of the quotient congruence, traces can still be regarded as a special case of comtraces. In principle, each trace commutativity equation, $ab = ba$, corresponds to two comtrace equations, $\{a,b\} = ab$ and $\{a,b\} = ba$ which can be captured in the following way.

Let $\Gamma = (E, ind)$ be a trace alphabet and, for each sequence $x = a_1 \dots a_n \in E^*$, let $x^{\langle\rangle} \overset{\mathrm{df}}{=} \{a_1\} \dots \{a_n\}$ be the corresponding sequence of singleton sets. We observe that when serialisability coincides with simultaneity, every comtrace can be represented by a singleton sequence and, moreover in that case trace congruence and comtrace congruence restricted to singleton sequences are the same.

Lemma 6.35. *Let ser = sim.*

(1) For each $\mathbf{t} \in \mathbb{S}^*/_{\equiv}$ *there is* $x \in E^*$ *such that* $\mathbf{t} = [\![x^{\langle\rangle}]\!]_{\equiv}$.
(2) If ser = ind then $x \equiv_{ind} y$ iff $x^{\langle\rangle} \equiv y^{\langle\rangle}$, *for all* $x, y \in E^*$.

Proof. (1) Let $\mathbf{t} = [\![A_1 \dots A_m]\!]$ where, for all $i \leq m$, $A_i = \{a^i_1, \dots, a^i_{n_i}\}$. Then $\mathbf{t} = [\![A_1]\!] \dots [\![A_m]\!]$ and, moreover, by $ser = sim$, we have for $i \leq m$: $[\![A_i]\!] = [\![\{a^i_1\}]\!] \cdots [\![\{a^i_{n_i}\}]\!]$.

(2) It suffices to show that $x \approx_{ind} y$ iff $x^{\langle\rangle} \approx^{sym} y^{\langle\rangle}$, where $\approx$ is as in Cor. 6.34. To see this, we observe that we have the following, for some $w, z \in E^*$ and $a, b \in E$:

$$\begin{aligned} x \approx_{ind} y &\iff x = wabz \ \wedge\ y = wbaz \ \wedge\ (a,b) \in ind \\ &\iff x^{\langle\rangle} = w^{\langle\rangle}\{a\}\{b\}bz^{\langle\rangle} \ \wedge\ y^{\langle\rangle} = w^{\langle\rangle}\{b\}\{a\}z^{\langle\rangle} \\ &\qquad\qquad\qquad\qquad \wedge\ \{a\} \times \{b\} \subseteq ser \ \wedge\ \{b\} \times \{a\} \subseteq ser \\ &\iff x^{\langle\rangle} \approx w^{\langle\rangle}\{a,b\}z^{\langle\rangle} \ \wedge\ w^{\langle\rangle}\{a,b\}z^{\langle\rangle} \approx^{-1} y^{\langle\rangle} \ . \end{aligned}$$

In the above, the first equivalence follows from the definition of $\approx_{ind}$, the second from $ser = ind$ and the symmetry of ind, and the third from the definition of $\approx$. $\square$

Let $\mathbf{t}$ be a trace over Γ and $\mathbf{v}$ be a comtrace over Θ. From Lem. 6.35 it follows that the singleton sequences in a comtrace correspond exactly to one trace if $sim = ser = ind$. Hence traces can be seen as the sequential core of comtraces. We say that $\mathbf{t}$ and $\mathbf{v}$ are *equivalent* if $sim = ser = ind$ and there is $x \in E^*$ such that $\mathbf{t} = [\![x]\!]_{\equiv_{ind}}$ and $\mathbf{v} = [\![x^{\langle\rangle}]\!]_{\equiv}$. We denote this by $\mathbf{t} \overset{\mathsf{T}\leftrightsquigarrow\mathsf{C}}{\equiv} \mathbf{v}$.

Proposition 6.36. *Let sim = ser = ind,* $\mathbf{t}, \mathbf{r}$ *be traces, and* $\mathbf{v}, \mathbf{w}$ *be comtraces.*

$$\begin{aligned} \mathbf{t} \overset{\mathsf{T}\leftrightsquigarrow\mathsf{C}}{\equiv} \mathbf{v} \wedge \mathbf{t} \overset{\mathsf{T}\leftrightsquigarrow\mathsf{C}}{\equiv} \mathbf{w} \quad &\Longrightarrow \quad \mathbf{v} = \mathbf{w} \\ \mathbf{t} \overset{\mathsf{T}\leftrightsquigarrow\mathsf{C}}{\equiv} \mathbf{v} \wedge \mathbf{r} \overset{\mathsf{T}\leftrightsquigarrow\mathsf{C}}{\equiv} \mathbf{v} \quad &\Longrightarrow \quad \mathbf{t} = \mathbf{r} \ . \end{aligned}$$

Also from the partial order point of view, traces can be regarded as a special case of comtraces. As we will show later on, in Thm. 6.53, equivalent traces and comtraces generate identical posets.

> Consider a comtrace alphabet $\Theta = (E, sim, ser)$ such that we have the following: $E = \{a,b,c,d,e\}$, $sim = \{(a,b),(b,a),(a,c),(c,a),(a,d),(d,a)\}$ and $ser = \{(a,b),(b,a),(a,c)\}$. Then $\mathbb{S} = \{\{a,b\},\{a,c\},\{a,d\},a,b,c,d,e\}$. Moreover, $\mathbf{z} = \mathbf{x}\hat{\circ}\mathbf{y}$ for the comtraces $\mathbf{x} = \{\{a,b\}ca, abca, baca, b\{a,c\}a\}$, $\mathbf{y} = \{e\{a,d\}\{a,c\}, e\{a,d\}ac\}$ and
>
> $$\mathbf{z} = \left\{ \begin{array}{lll} \{a,b\}cae\{a,d\}\{a,c\} & abcae\{a,d\}\{a,c\} & bacae\{a,d\}\{a,c\} \\ b\{a,c\}ae\{a,d\}\{a,c\} & \{a,b\}cae\{a,d\}ac & abcae\{a,d\}ac \\ bacae\{a,d\}ac & b\{a,c\}ae\{a,d\}ac & \end{array} \right\}.$$

Algebraic Properties. Algebraic properties of trace congruence operations such as *left/right cancellation* and *projection* are well understood [33]. They are intuitive and powerful tools with many applications [4]. The basic obstacle for lifting these properties to comtraces is the necessary switching from sequences to step sequences. Furthermore unlike the independence relation for traces, the serialisability relation *ser* is in general not commutative. So comtrace congruence does not have a *mirror rule* which states that if two sequences are congruent, then their *reverses* or *mirror images* are also congruent. Hence in trace theory, *right cancellation* alone is sufficient to extract congruent *subsequences* from congruent sequences, since *left cancellation* follows from right-cancellation in the mirror images. For comtraces however we need a separate notion of left-cancellation.

Let $a \in E$. The operators $\div_R a$ and $\div_L a$ of, respectively, *right cancellation* and *left cancellation* in step sequences, are defined by $\lambda \div_R a = \lambda \div_L a \overset{\mathrm{df}}{=} \lambda$ and

$$wA \div_R a \overset{\mathrm{df}}{=} \begin{cases} (w \div_R a)A & \text{if } a \notin A \\ w & \text{if } A = \{a\} \\ w(A \backslash \{a\}) & \text{otherwise} \end{cases} \qquad Aw \div_L a \overset{\mathrm{df}}{=} \begin{cases} A(w \div_L a) & \text{if } a \notin A \\ w & \text{if } A = \{a\} \\ (A \backslash \{a\})w & \text{otherwise} \end{cases}$$

where $A \in \mathbb{S}$ and $w \in \mathbb{S}^*$. Since $(w \div_R a) \div_R b = (w \div_R b) \div_R a$, we can extend $\div_R$ so that it is parameterised by step sequences. More precisely, we define:

$$\begin{aligned} w \div_R \{a_1, \ldots, a_n\} &\overset{\mathrm{df}}{=} \big(\ \ldots \big((w \div_R a_1) \div_R a_2\big) \ \ \ldots \big) \div_R a_n \\ w \div_R A_1 \ldots A_k &\overset{\mathrm{df}}{=} \big(\ \ldots \big((w \div_R A_1) \div_R A_2\big) \ \ldots \big) \div_R A_k \\ w \div_R \lambda &\overset{\mathrm{df}}{=} w \end{aligned}$$

for all steps $\{a_1,\ldots,a_n\}, A_1,\ldots,A_k \in \mathbb{S}$ and step sequences $w \in \mathbb{S}^*$. We also extended $\div_L$ in a similar way.

Moreover, for every subset D of E, the *projection* function $\pi_D : \mathbb{S}^* \to \mathbb{S}^*$ is such that $\pi_D(\lambda) \overset{\text{df}}{=} \lambda$ and, for all $A \in \mathbb{S}$ and $w \in \mathbb{S}^*$:

$$\pi_D(wA) \overset{\text{df}}{=} \begin{cases} \pi_D(w) & \text{if } A \cap D = \varnothing \\ \pi_D(w)(A \cap D) & \text{otherwise .} \end{cases}$$

The result below shows that the algebraic properties of comtraces are similar to the algebraic properties of traces [33].

Proposition 6.37. *Let $u, v, w, s, t \in \mathbb{S}^*$, $a \in E$, and $D \subseteq E$.*

(1) $u \equiv v \implies wgt(u) = wgt(v)$. (step sequence weight equality)
(2) $u \equiv v \implies \#_a(u) = \#_a(v)$. (event preservation)
(3) $u \equiv v \implies u \div_R w \equiv v \div_R w$. (right cancellation)
(4) $u \equiv v \implies u \div_L w \equiv v \div_L w$. (left cancellation)
(5) $u \equiv v \iff sut \equiv svt$. (step subsequence cancellation)
(6) $u \equiv v \implies \pi_D(u) \equiv \pi_D(v)$. (projection rule)

Proof. For all parts except (5), it suffices to show that $u \approx v$ implies the right hand side of the relevant formula. Recall that $u \approx v$ means that $u = xAz$ and $v = xBCz$, for some $x, z \in \mathbb{S}^*$ and $A, B, C \in \mathbb{S}$ satisfying $A = B \cup C$, $B \cap C = \varnothing$ and $B \times C \subseteq ser$. Since this immediately implies that weight and events are preserved by $\approx$, we only discuss parts (3)–(6).

(3) Let $a \in E$. We prove $u \div_R a \equiv v \div_R a$ whenever $u \approx v$ as above.
We distinguish four cases.

Case 1: $a \in alph(z)$. Then $u \div_R a = xAy \approx xBCy = v \div_R a$, where $y = z \div_R a$.

Case 2: $a \in C \backslash alph(z)$. Then $u \div_R a = x(A \backslash \{a\})z \approx xB(C \backslash \{a\})z = v \div_R a$.

Case 3: $a \in B \backslash alph(Cz)$. Then $u \div_R a = x(A \backslash \{a\})z \approx x(B \backslash \{a\})Cz = v \div_R a$.

Case 4: $a \notin alph(Az)$. Then $u \div_R a = yAz \approx yBCz = v \div_R a$, where $y = x \div_R a$.

(4) The proof is similar to that of part (3).

(5) The ($\implies$) implication follows from the fact that $\equiv$ is a congruence. To show the ($\implies$) implication, we observe that, by $sut \equiv svt$ and parts (3) and (4):

$$u = (sut \div_R t) \div_L s \equiv (svt \div_R t) \div_L s = v \ .$$

(6) We observe that

$$\pi_D(u) = \pi_D(x)\pi_D(A)\pi_D(z) \approx \pi_D(x)\pi_D(B)\pi_D(C)\pi_D(z) = \pi_D(v) \ .$$

which follows from $\pi_D(A) = \pi_D(B) \cup \pi_D(C)$ and $\pi_D(B) \times \pi_D(C) \subseteq \mathit{ser}$. □

An immediate consequence of the event preservation is event occurrence preservation which means that $u \equiv v$ implies $\mathit{occ}(u) = \mathit{occ}(v)$. Note furthermore that comtrace congruence preserves the length of sequences, but not of step sequences.

A Canonical Form for Comtraces. The canonical form of a comtrace that will be discussed in this subsection, has a clear interpretation. Similar to the canonical stratified extension of posets (Prop. 6.13), it essentially describes a greedy, maximally concurrent, execution of the events occurring in the comtrace conforming to the simultaneity and serialisability relations. Actually, it is a straightforward application of the approach used in [20] for an alternative vector representation of traces in [19, 43]. Greedy execution turned out to be a useful technique also for the transformation of vector control languages into rational relations in [26].

Definition 6.38 (Greedy maximally concurrent form). *A step sequence $u = A_1 \dots A_k \in \mathbb{S}^*$ is in* greedy maximally concurrent form *(or GMC-form) if, for each $i \leq n$, whenever $Av \equiv A_i \dots A_k$ for some $A \in \mathbb{S}$ and $v \in \mathbb{S}^*$, then $|A| \leq |A_i|$.*

Proposition 6.39. *Each comtrace comprises a step sequence in GMC-form.*

Proof. Let $\mathbf{t}$ be a comtrace and $v \in \mathbf{t}$. We derive a sequence of steps $u = A_1 \dots A_k$, as follows:

```
initialise $i \leftarrow 0$ and $v_0 \leftarrow v$
while $v_i \neq \lambda$ do
    $i \leftarrow i + 1$
    find $A_i$ such that $|A| \leq |A_i|$, for each $Aw \equiv v_{i-1}$
    $v_i \leftarrow v_{i-1} \div_L A_i$
endwhile
```

The algorithm always terminates as $\mathit{wgt}(v_{i+1}) < \mathit{wgt}(v_i)$, for all i. Moreover, u is in GMC-form and $u \in \mathbf{t}$. □

The algorithm in the above proof is indeed greedy in the sense that it adds as many symbols as early as possible to each step. Moreover, the GMC-form is equivalent to the unique Foata-type canonical form of comtraces proposed in [17]. To prove this claim, we introduce a relation between steps generalising the independence between events exploited in the Foata canonical form by observing that $(a, c) \in ser$ means that the sequence ac can be replaced by the set $\{a, c\}$.

Definition 6.40 (Forward dependency). *A relation* $\mathbb{FD}$ *of* forward dependency *on steps comprises all pairs* $(A, B) \in \mathbb{S} \times \mathbb{S}$ *for which there is a step* $C \subseteq B$ *such that* $A \times C \subseteq ser$ *and* $C \times (B \backslash C) \subseteq ser$*.*

Note that in the above definition C is a step and hence is non-empty, and $C = B$ is allowed. The next result explains the term 'forward dependency'. If $(A, B) \notin \mathbb{FD}$ then there are no elements in B that can be moved forward from B in AB to A without losing the equivalence with AB.

Lemma 6.41. *A pair of steps* (A, B) *belongs to* $\mathbb{FD}$ iff $A \cup B \equiv AB$ *or, for some step* $C \subset B$, $(A \cup C)(B \backslash C) \equiv AB$*.*

Proof. $(\Longrightarrow)$ Let $C \subseteq B$ be as in Defn. 6.40. If $C = B$ then $A \cup B \approx AB$, and if $C \neq B$ then $(A \cup C)(B \backslash C) \approx AC(B \backslash C) \approx AB$.

$(\Longleftarrow)$ If $A \cup B \equiv AB$ then $A \cup B \in \mathbb{S}$ and, by Prop. 6.37(2), $A \cap B = \varnothing$. Let $a \in A$ and $b \in B$. By Prop. 6.37(6), $\{a, b\} = \pi_{\{a,b\}}(A \cup B) \equiv \pi_{\{a,b\}}(AB) = \{a\}\{b\}$ which means that $(a, b) \in ser$. Hence $A \times B \subseteq ser$, and so $(A, B) \in \mathbb{FD}$.

Suppose now that $C \subset B$ and $(A \cup C)(B \backslash C) \equiv AB$. Then $A \cup C \in \mathbb{S}$ and, by Prop. 6.37(2), $A \cap C = \varnothing$. Let $a \in A$ and $c \in C$. By Prop. 6.37(6), we have that: $\{a, c\} = \pi_{\{a,c\}}((A \cup C)(B \backslash C)) \equiv \pi_{\{a,c\}}(AB) = \{a\}\{c\}$ which means $(a, c) \in ser$. Hence $A \times C \subseteq ser$. Let $b \in B \backslash C$ and $c \in C$. Again, by Prop. 6.37(6), we have $\{c\}\{b\} = \pi_{\{b,c\}}(A \cup C)(B \backslash C) \equiv \pi_{\{b,c\}}(AB) = \{b, c\}$ which means that $(c, b) \in ser$, and so $C \times (B \backslash C) \subseteq ser$. Hence $(A, B) \in \mathbb{FD}$. $\square$

The canonical step sequence representing a comtrace introduced in [17] can now be defined.

Definition 6.42 (Comtrace canonical step sequence). *A step sequence* $u = A_1 \dots A_k$ *is* canonical *if* $(A_i, A_{i+1}) \notin \mathbb{FD}$*, for all* $i < k$*.*

The canonical step sequences for comtraces refine and generalise the Foata canonical form and can be seen as greedy according to the next result.

Lemma 6.43. $A_1 = \bigcup\{C \mid Cv \in [\![u]\!]\}$ *for a comtrace canonical step sequence* $u = A_1 \dots A_k$.

Proof. Let $A = \bigcup\{C \mid Cv \in [\![u]\!]\}$. Since $u \in [\![u]\!]$, $A_1 \subseteq A$. We need to prove that $A \subseteq A_1$. Clearly, $A = A_1$ if $k = 1$, so assume $k > 1$. Suppose that $a \in A \backslash A_1$, $a \in A_j$, $1 < j \leq k$ and $a \notin A_i$ for $i < j$. Since $a \in A$, there is $v = Bx \in [\![u]\!]$ such that $a \in B$. Note that $A_{j-1}A_j$ is also canonical and

$$u' = A_{j-1}A_j = (u \div_R (A_{j+1} \dots A_k)) \div_L (A_1 \dots A_{j-2}) \ .$$

Let $v' \overset{\text{df}}{=} (v \div_R (A_{j+1} \dots A_k)) \div_L (A_1 \dots A_{j-2})$. We have $v' = B'x'$ where $a \in B'$. By Prop. 6.37(3,4) $u' \equiv v'$. Since $u' = A_{j-1}A_j$ is canonical, we can consider two cases.

Case 1: $(c, a) \notin ser$, for some $c \in A_{j-1}$. Then we have:

$$\pi_{\{a,c\}}(u') = ca \text{ (if } c \notin A_j) \quad \text{or} \quad \pi_{\{a,c\}}(u') = c\{a, c\} \text{ (if } c \in A_j) \ .$$

In the former case, $\pi_{\{a,c\}}(v')$ equals either $\{a, c\}$ (if $c \in B'$) or ac (if $c \notin B'$), and so in both cases $\pi_{\{a,c\}}(u') \not\equiv \pi_{\{a,c\}}(v')$, contradicting Prop. 6.37(6). In the latter case, $\pi_{\{a,c\}}(v')$ equals either $\{a, c\}c$ (if $c \in B'$) or acc (if $c \notin B'$), and so in both cases $\pi_{\{a,c\}}(u') \not\equiv \pi_{\{a,c\}}(v')$, contradicting Prop. 6.37(6).

Case 2: $(a, b) \notin ser$, for some $b \in A_j$. Then we take $d \in A_{j-1}$ and have the following:

$$\pi_{\{a,b,d\}}(u') = d\{a, b\} \text{ (if } d \notin A_j) \quad \text{or} \quad \pi_{\{a,b,d\}}(u') = d\{a, b, d\} \text{ (if } d \in A_j) \ .$$

In the former case, $\pi_{\{a,b,d\}}(v')$ is one of the following step sequences:

$$\{a, b, d\} \quad \{a, b\}d \quad \{a, d\}b \quad abd \quad adb$$

and so in each case $\pi_{\{a,b,d\}}(u') \not\equiv \pi_{\{a,b,d\}}(v')$, contradicting Prop. 6.37(6). In the latter case, $\pi_{\{a,b,d\}}(v')$ is one of the following step sequences:

$$\{a, b, d\}d \quad \{a, b\}dd \quad \{a, d\}\{b, d\} \quad \{a, d\}bd \quad \{a, d\}db \quad abdd \quad adbd \quad addb$$

and so in each case $\pi_{\{a,b,d\}}(u') \not\equiv \pi_{\{a,b,d\}}(v')$, contradicting Prop. 6.37(6). $\square$

Now we can show that the above canonical form and GMC-form are equivalent.

Theorem 6.44. *A step sequence u is in GMC-form* iff *it is canonical.*

Proof. ($\Longleftarrow$) Suppose that $u = A_1 \dots A_k$ is canonical. By Lem. 6.43, we have that for each $B_1 y_1 \equiv A_1 \dots A_k$, $|B_1| \leq |A_1|$. Since each $A_i \dots A_k$ is canonical, we have by Lem. 6.43, that for each $B_2 y_2 \equiv A_2 \dots A_k$, $|B_2| \leq |A_2|$. Proceeding in this way k times we obtain that $u = A_1 \dots A_k$ is in GMC-form.

($\Longrightarrow$) Suppose that $u = A_1 \dots A_k$ is not canonical, and j is the smallest number such that $(A_j, A_{j+1}) \in \mathbb{FD}$. Hence $A_1 \dots A_{j-1}$ is canonical, and, by the ($\Longleftarrow$) implication of this theorem, in GMC-form. By Lem. 6.41, either there is a non empty $C \subset A_{j+1}$ such that $(A_j \cup C)(A_{j+1} \backslash C) \equiv A_j A_{j+1}$, or $A_j \cup A_{j+1} \equiv A_j A_{j+1}$. In the former case, we have $|A_j \cup C| > |A_j|$ as $C \neq \varnothing$, and, in the latter case, $A_j \dots A_k$ is not in GMC-form as $|A_j \cup A_{j+1}| > |A_j|$, which means that u is not in GMC-form either. □

Just as each trace can be represented by a unique sequence in canonical form, each comtrace has a single canonical step sequence.

Theorem 6.45. *For each step sequence v there is a unique canonical step sequence u such that $v \equiv u$.*

Proof. The existence of u follows from Prop. 6.39 and Thm. 6.44. We only need to show its uniqueness. Suppose that $u = A_1 \dots A_k$ and $v = B_1 \dots B_m$ are both canonical step sequences and $u \equiv v$. By induction on $k = |u|$, we will show that $u = v$. By Lem. 6.43, we have that $B_1 = A_1$. If $k = 1$ we are done. Otherwise, let

$$u' = A_2 \dots A_k \quad \text{and} \quad w' = B_2 \dots B_m$$

and u', w' be both canonical step sequences of $[\![u']\!]$. Since $|u'| < |u|$, by the induction hypothesis, we obtain $A_i = B_i$ for $i = 2, \dots, k$ and $k = m$. □

Finally, we establish that no equivalent step sequence is shorter than the canonical one (confirming its greediness).

Proposition 6.46. *If u is a canonical step sequence and $u \equiv v$, then $len(u) \leq len(v)$.*

Proof. The proof proceeds by induction on $len(v)$. In the base case, $len(v) = 1$, the result follows from the definition of $\mathbb{FD}$ by which $u = v$ must hold. In the induction step, we assume that the result holds for all v such that $len(v) \leq r - 1$ where $r \geq 2$. Let us consider $v = B_1 B_2 \dots B_r$, and let $u = A_1 A_2 \dots A_k$ be a canonical step sequence such that $v \equiv u$. Moreover, $v_1 \overset{\mathrm{df}}{=} v \div_L A_1 = C_1 \dots C_s$. By Prop. 6.37(4) $v_1 \equiv u \div_L A_1 = A_2 \dots A_k$, and $A_2 \dots A_k$ is clearly canonical. Hence, by the induction hypothesis, $k-1 = len(A_2 \dots A_k) \le s$. By Lem. 6.43, we obtain $B_1 \subseteq A_1$ which means that $v_1 = v \div_L A_1 = B_2 \dots B_r \div_L (A_1 \backslash B_1) = C_1 \dots C_s$ which in turn means that $s \le r-1$. Hence $k-1 \le s \le r-1$, and so $k \le r$. □

Relationship with Stratified Order Structures. We now aim to show that the relationship between comtraces and so-structures is basically the same as that between traces and posets; more precisely, that each comtrace is represented by a unique so-structure, and each so-structure defines a single comtrace.

We start with the definition of the closure construction crucial for the application of so-structures to the modelling of concurrent systems [17, 28].

Definition 6.47 (Diamond closure of relational structures). *Given a relational structure* $S \overset{\text{df}}{=} (X, Q, R)$*, the* $\Diamond$-closure *of* S *is:*

$$S^{\Diamond} \overset{\text{df}}{=} \big(X, \prec_{QR}, \sqsubset_{QR}\big)$$

where $\prec_{QR} \overset{\text{df}}{=} (Q \cup R)^* \circ Q \circ (Q \cup R)^*$ *and* $\sqsubset_{QR} \overset{\text{df}}{=} (Q \cup R)^* \backslash id_X$.

Diamond closure can be seen as a generalisation of the transitive closure of an acyclic relation to obtain a partial order. The idea is that 'reasonable' relations Q and R with common domain X, form a basis for a relational structure $(X, Q, R)^{\Diamond}$ satisfying the axioms (S1)–(S4) in the definition of an so-structure. As an example, note that whenever $Q = R$ and Q is irreflexive and acyclic, then diamond closure describes a poset: $(X, Q, R)^{\Diamond} = (X, Q^+, Q^+)$. The following result shows that the properties of $\Diamond$-closure are rather similar to those of transitive closure.

Theorem 6.48. *Let* $S = (X, Q, R)$ *be a relational structure.*

(1) If R *is irreflexive, then* $Q \subseteq \prec_{QR}$ *and* $R \subseteq \sqsubset_{QR}$.
(2) $\big(S^{\Diamond}\big)^{\Diamond} = S^{\Diamond}$.
(3) $S^{\Diamond}$ *is an so-structure iff* $\prec_{QR}$ *is irreflexive.*
(4) If S *is an so-structure, then* $S = S^{\Diamond}$.

Proof. (1) $Q \subseteq \prec_{QR}$, and if R is irreflexive, then $R \subseteq R \backslash id_X \subseteq \sqsubset_{QR}$.

(2) We first show that $(\prec_{QR} \cup \sqsubset_{QR})^* = (Q \cup R)^*$. The $(\subseteq)$ inclusion follows from the definitions of $\prec_{QR}$ and $\sqsubset_{QR}$ by which $\prec_{QR} \cup \sqsubset_{QR} \subseteq (Q \cup R)^*$ while the converse follows from $(Q \cup R)^* = (\sqsubset_{QR})^* \subseteq (\prec_{QR} \cup \sqsubset_{QR})^*$. We

can now show $\left(S^{\Diamond}\right)^{\Diamond} = S^{\Diamond}$. Note that

$$\begin{array}{rcl} \prec_{\prec_{QR},\sqsubset_{QR}} & = & (\prec_{QR} \cup \sqsubset_{QR})^* \circ \prec_{QR} \circ (\prec_{QR} \cup \sqsubset_{QR})^* \\ & = & (Q \cup R)^* \circ \prec_{QR} \circ (Q \cup R)^* \quad = \quad \prec_{QR} \ . \end{array}$$

Similarly, $\sqsubset_{\prec_{QR},\sqsubset_{QR}} = (\prec_{QR} \cup \sqsubset_{QR})^* \backslash id_X = (Q \cup R)^* \backslash id_X = \sqsubset_{QR}$.

(3) The ($\Longrightarrow$) implication follows from the axioms (S1) and (S2) for so-structures. To show the ($\Longleftarrow$) implication, we proceed as follows. (S1) clearly holds and (S2) follows from $\prec_{QR}$ being irreflexive. Then (S3) and (S4) follow from
$(\sqsubset_{QR} \circ \sqsubset_{QR}) \backslash id_X \subseteq \sqsubset_{QR}$, $\sqsubset_{QR} \circ \prec_{QR} \subseteq \prec_{QR}$ and $\prec_{QR} \circ \sqsubset_{QR} \subseteq \prec_{QR}$.

(4) We observe that $\prec_{QR} = (Q \cup R)^* \circ Q \circ (Q \cup R)^* =_{S2} R^* \circ Q \circ R^* =_{S4} Q$ and $\sqsubset_{QR} = (Q \cup R)^* \backslash id_X =_{S2} R^* \backslash id_X =_{S3} R \backslash id_X = R$. □

By Thm. 6.48(2), $\Diamond$ is indeed a closure operator. Furthermore, $S^{\Diamond}$ is an so-structure iff $\prec_{QR} = (\prec_{QR})^+$ is acyclic, and hence a partial order (Q specifies causality — or 'earlier than' — relationship between certain events). Note that $\sqsubset_{QR}$ is irreflexive, but not necessarily acyclic (R describes weak causality — or 'earlier than or simultaneous' — relationship between certain events).

Using the $\Diamond$-closure operator, we prove how, for a given comtrace alphabet, every comtrace is completely determined by any single one of its step sequences.

Let u be a step sequence in $\mathbb{S}^*$. We define two binary relations Q^u and R^u on $occ(u)$, as follows:

$$\begin{array}{lcl} (\alpha,\beta) \in Q^u & \overset{df}{\Longleftrightarrow} & \alpha \prec_u \beta \ \wedge \ (l(\alpha), l(\beta)) \notin ser \\ (\alpha,\beta) \in R^u & \overset{df}{\Longleftrightarrow} & \alpha \precsim_u \beta \ \wedge \ (l(\beta), l(\alpha)) \notin ser \end{array}$$

where $(occ(u), \prec_u)$ is the canonical stratified poset $canspo(u)$ of u. In other words, Q^u provides the information on the necessary strict ordering (w.r.t. Θ) of pairs of symbol occurrences in u, while R^u tells us for certain pairs of occurrences which one should be not later than the other one. As the next lemma shows, these two relations are respected by all step sequences belonging to the same comtrace. (Recall that $u \equiv v$ implies that $occ(u) = occ(v)$).

Lemma 6.49. *If $u \equiv v$ then $Q^u = Q^v$ and $R^u = R^v$, for all $u, v \in \mathbb{S}^*$.*

Proof. It suffices to show that $u \approx v$ implies $Q^u = Q^v$ and $R^u = R^v$. So we let $u = wAz$ and $v = wBCz$ where $w, z \in \mathbb{S}^*$ and $A, B, C \in \mathbb{S}$

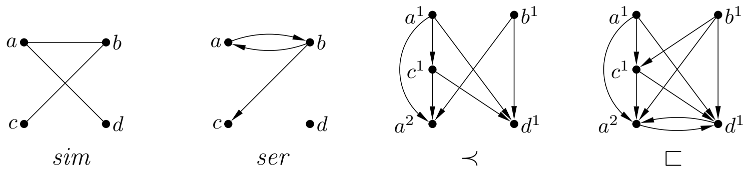


Fig. 6.4 Relations defining a comtrace alphabet and an so-structure.

satisfy $B \cup C = A$ and $B \times C \subseteq ser$. Consequently, the only difference between u and v is the ordering of symbol occurrences in A (unordered) in comparison with their ordering in BC (one before the other) which belong to ser. Hence $Q^u = Q^v$ and $R^u = R^v$. □

Since $u \equiv v$ implies $occ(u) = occ(v)$, we can define the set of occurrences of the comtrace $[\![u]\!]$ as $occ([\![u]\!]) \overset{\mathrm{df}}{=} occ(u)$. Together with the above, this allows us to associate with each comtrace a (well-defined) so-structure determined by any one of its step sequences and Θ.

Definition 6.50 (So-structure defined by step sequence and comtrace alphabet). Let u be a step sequence in $\mathbb{S}^*$. Then $S^{\langle u \rangle} \overset{\mathrm{df}}{=} \big(occ([\![u]\!]), Q^u, R^u\big)^{\diamondsuit}$ is the *so-structure defined* by u and Θ.

Note that, by Thm. 6.48(3), $S^{\langle u \rangle}$ is indeed an so-structure since $\prec_{Q^u R^u} = (Q^u \cup R^u)^* \circ Q^u \circ (Q^u \cup R^u)^*$ is irreflexive. Furthermore, this so-structure is different from the stratified order structure $S_u = (occ(u), \prec_u, \sqsubset_u)$ with $\sqsubset_u = \frown\!\!\!\!\prec_u$, as defined in Prop. 6.20. There is however, also from Prop. 6.20, the so-structure $S_{[\![u]\!]}$ defined by (the stratified posets representing the step sequences in) comtrace $[\![u]\!]$:

$$S_{[\![u]\!]} \overset{\mathrm{df}}{=} \Big(occ([\![u]\!]), \bigcap_{x \in [\![u]\!]} \prec_x, \bigcap_{x \in [\![u]\!]} \frown\!\!\!\!\prec_x\Big)$$

This so-structure will be shown to be identical to $S^{\langle u \rangle}$ and, moreover, the stratified extensions of these two so-structures turn out to correspond exactly to the step sequences in $[\![u]\!]$.

Figure 6.4 shows a comtrace alphabet $\Theta = (\{a,b,c,d\}, sim, ser)$ and the two relations of so-structure defined by the step sequence $u = \{a,b\}c\{a,d\}$ and Θ. Note that we have $[\![u]\!] = \big\{\{a,b\}c\{a,d\}, abc\{a,d\}, a\{b,c\}\{a,d\}, bac\{a,d\}\big\}$ as well as that $\prec$ is equal to $Q^u \cup \{(b^1, a^2)\}$ and $\sqsubset$ is equal to R^u.

By Prop. 6.13 and Lem. 6.49, we have for each comtrace $[\![u]\!]$ a well-defined poset $(occ(u), \prec_{Q^uR^u})$ representing the causality in the comtrace. We will show first that the stratified poset extension of $(occ(u), \prec_{Q^uR^u})$ corresponds to the unique canonical step sequence in $[\![u]\!]$. Hence, the canonical representative of a comtrace can be read off directly from any of its representatives.

Proposition 6.51. $(occ(u), \prec_u) = canstratposet(occ(u), \prec_{Q^uR^u})$*, for every canonical step sequence* u*.*

Proof. Let $u = A_1 \dots A_n$ and let $\widehat{u} \stackrel{\text{df}}{=} \widehat{A}_1 \dots \widehat{A}_k$ be its enumerated step sequence. By Prop. 6.13, it suffices to show that for every $i \geq 2$ and every $\beta \in \widehat{A}_i$, there is $\alpha \in \widehat{A}_{i-1}$ such that $\alpha \prec_{Q^uR^u} \beta$. Suppose that this does not hold. Then

$$B = \{\beta \in \widehat{A}_i \mid \forall \alpha \in \widehat{A}_{i-1} : \neg \alpha \prec_{Q^uR^u} \beta\} \neq \varnothing$$

for some $i \geq 2$. Since $\prec_{Q^uR^u} = (Q^u \cup R^u)^* \circ Q^u \circ (Q^u \cup R^u)^*$ it follows from the definitions of Q^u and R^u, that $\widehat{A}_{i-1} \times l(B) \subseteq ser$.

Suppose there are $\alpha \in B$ and $\beta \in \widehat{A}_i \backslash B$ such that $(l(\alpha), l(\beta)) \notin ser$. Then $\beta \sqsubset_{Q^uR^u} \alpha$, and, by the definition of B, $\gamma \prec_{Q^uR^u} \beta$ for some $\gamma \in \widehat{A}_{i-1}$. Thus we have $A_{i-1} \times l(B) \subseteq ser$ and $l(B) \times (A_i \backslash l(B)) \subseteq ser$ contradicting u being canonical. $\square$

Secondly, we have as an important auxiliary result that step sequences representing a stratified poset extension of an so-structure defined by a step sequence (and comtrace alphabet), actually define the same so-structure.

Lemma 6.52. $S^{\langle v \rangle} = S^{\langle u \rangle}$*, for all* u, v *satisfying* $(occ(v), \prec_v) \in ext(S^{\langle u \rangle})$*.*

Proof. It follows from $(occ(v), \prec_v) \in ext(S^{\langle u \rangle})$, that we have $occ(v) = occ(u)$ and $\prec_{Q^uR^u} \subseteq \prec_v$. Hence

$$\begin{aligned} \alpha Q^u \beta &\implies (\alpha \prec_{Q^uR^u} \beta \ \wedge \ (l(\alpha), l(\beta)) \notin ser) \\ &\implies (\alpha \prec_v \beta \ \wedge \ (l(\alpha), l(\beta)) \notin ser) \iff \alpha Q^v \beta \end{aligned}$$

and so $Q^v \subseteq Q^u$. Similarly, $R^u \subseteq R^v$ follows from:

$$\begin{aligned} \alpha R^u \beta &\implies (\alpha \sqsubset_{Q^uR^u} \beta \ \wedge \ (l(\beta), l(\alpha)) \notin ser) \\ &\implies (\alpha \precsim_v \beta \ \wedge \ (l(\beta), l(\alpha)) \notin ser) \iff \alpha R^v \beta\,. \end{aligned}$$

To prove the converse inclusions, first suppose that $\alpha Q^v \beta$ and $\neg(\alpha Q^u \beta)$. Since $\alpha Q^v \beta$ implies $(l(\alpha), l(\beta)) \notin ser$, we have that $\neg(\alpha Q^u \beta)$ means $\neg(\alpha \prec_u \beta)$. Hence $\beta \precsim_u \alpha$ and, consequently, $\beta R^u \alpha$. But $\alpha Q^v \beta$ and $\beta R^u \alpha$

imply $\alpha \prec_v \beta$ and $\beta \sqsubset_{Q^u R^u} \alpha$, contradicting $(occ(v), \prec_v) \in ext(S^{\langle u \rangle})$.
Next, suppose that $\alpha R^v \beta$ and $\neg(\alpha R^u \beta)$. Since $\alpha R^v \beta$ implies $(l(\beta), l(\alpha)) \notin ser$, we have that $\neg(\alpha R^u \beta)$ means $\neg(\alpha \overset{\frown}{<}_u \beta)$. Hence $\beta \prec_u \alpha$ and, consequently, $\beta Q^u \alpha$. But $\alpha R^v \beta$ and $\beta Q^u \alpha$ imply $\alpha \overset{\frown}{<}_v \beta$ and $\beta \prec_{Q^u R^u} \alpha$, contradicting $(occ(v), \prec_v) \in ext(S^{\langle u \rangle})$ again.
Hence $S^{\langle u \rangle} = S^{\langle v \rangle}$. □

Now we are ready to prove that each comtrace defines a single so-structure.

Theorem 6.53. *Let* $u, v \in \mathbb{S}^*$.

(1) $u \equiv v \iff S^{\langle u \rangle} = S^{\langle v \rangle}$,
(2) $ext\big(S^{\langle u \rangle}\big) = \big\{(occ(x), \prec_x) \mid x \in [\![u]\!]\big\}$,
(3) $S^{\langle u \rangle} = S_{[\![u]\!]}$.

Proof. (1) From Lem. 6.49 we immediately have that $u \equiv v$ implies $S^{\langle u \rangle} = S^{\langle v \rangle}$. Assume $S^{\langle u \rangle} = S^{\langle v \rangle}$. By Thm. 6.45, there are canonical step sequences u', v' such that $u \equiv u'$ and $v \equiv v'$. By Lem. 6.49, $S^{\langle u' \rangle} = S^{\langle u \rangle}$ and $S^{\langle v' \rangle} = S^{\langle v \rangle}$, and so $S^{\langle u' \rangle} = S^{\langle v' \rangle}$. With Prop. 6.51 this yields $\prec_{u'} = \prec_{v'}$. Thus $u' = v'$, so $u \equiv v$.

(2) By definition, $Q^u \subseteq \prec_u$ and $R^u \subseteq \overset{\frown}{<}_u$ which implies $\prec_{Q^u R^u} \subseteq \prec_u$ and $\sqsubset_{Q^u R^u} \subseteq \overset{\frown}{<}_u$, i.e., $(occ(u), \prec_u) \in ext(S^{\langle u \rangle})$, and so $\big\{(occ(x), \prec_x) \mid x \in [\![u]\!]\big\}$ is included in $ext\big(S^{\langle u \rangle}\big)$.
Let v be a step sequence such that $(occ(v), \prec_v) \in ext(S^{\langle u \rangle})$. By Lem. 6.52, $S^{\langle u \rangle} = S^{\langle v \rangle}$. This and the assertion (1) above yields $u \equiv v$, and so $ext\big(S^{\langle u \rangle}\big)$ is included in $\big\{(occ(x), \prec_x) \mid x \in [\![u]\!]\big\}$.

(3) By the definition of $S_{[\![u]\!]}$, $\big\{(occ(x), \prec_x) \mid x \in [\![u]\!]\big\} \subseteq ext\big(S_{[\![u]\!]}\big)$. By Lem. 6.49 $\prec_{Q^u R^u} \subseteq \prec_x$ and $\sqsubset_{Q^u R^u} \subseteq \overset{\frown}{<}_x$, for all $x \in [\![u]\!]$, which implies $S^{\langle u \rangle} \subseteq S_{[\![u]\!]}$. Hence $ext(S_{[\![u]\!]}) \subseteq ext(S^{\langle u \rangle})$. From part (2) we now have that $\big\{(occ(x), \prec_x) \mid x \in [\![u]\!]\big\} \subseteq ext\big(S_{[\![u]\!]}\big) \subseteq ext\big(S^{\langle u \rangle}\big) = \big\{(occ(x), \prec_x) \mid x \in [\![u]\!]\big\}$ which means that $ext(S_{[\![u]\!]}) = ext(S^{\langle u \rangle})$. Together with Thm. 6.22 this yields $S^{\langle u \rangle} = S_{[\![u]\!]}$. □

By Thm. 6.53, the so-structures $S^{\langle u \rangle}$ and $S_{[\![u]\!]}$ are identical and their set of stratified extensions is exactly the comtrace $[\![u]\!]$ with step sequences interpreted as stratified posets. From an algorithmic point of view, the definition of $S^{\langle u \rangle}$ is the more interesting one, since building the relations $\prec_u$ and $\sqsubset_u$ and getting their $\diamondsuit$-closure, which in turn can be reduced to computing transitive closures, can be done efficiently. In contrast, a direct use of the definition of $S_{[\![u]\!]}$ requires precomputing of potentially exponentially many elements of the comtrace $[\![u]\!]$.

Theorem 6.53 characterises the so-structures derived from a comtrace. We will now discuss how to derive a comtrace from an so-structure.

Definition 6.54 (Simultaneity and serialisability in so-structures). The simultaneity and serialisability relations *induced* by the so-structure $S = (X, \prec, \sqsubset)$ are respectively given below, for all distinct $a, b \in X$:

$$\begin{aligned}(a,b) \in \mathit{sim}_S &\overset{df}{\Longleftrightarrow} a \frown_{(X,\prec)} b\\ (a,b) \in \mathit{ser}_S &\overset{df}{\Longleftrightarrow} a \frown_{(X,\prec)} b \ \wedge\ \neg(b \sqsubset a)\ .\end{aligned}$$

That the terminology in the definition is justified, follows from the next result.

Proposition 6.55. *For all distinct $a, b \in X$:*

$$\begin{aligned}(a,b) \in \mathit{sim}_S &\Longleftrightarrow \exists \mathit{spo} \in \mathit{ext}(S) :\ a \frown_{\mathit{spo}} b\\ (a,b) \in \mathit{ser}_S &\Longleftrightarrow \exists \mathit{spo}, \mathit{spo}' \in \mathit{ext}(S) :\ a \frown_{\mathit{spo}} b \ \wedge\ a \prec_{\mathit{spo}'} b\\ (a,b) \notin \mathit{ser}_S &\Longleftrightarrow a \prec b \ \vee\ b \sqsubset a\ .\end{aligned}$$

Proof. The first part is a consequence of Theorems 6.22 and 6.23 as we have:

$$\begin{aligned}(a,b) \in \mathit{sim}_S &\Longleftrightarrow \neg(a \prec b) \wedge \neg(b \prec a)\\ &\Longleftrightarrow \neg(\forall \mathit{spo} \in \mathit{ext}(S) :\ a \prec_{\mathit{spo}} b) \wedge \neg(\forall \mathit{spo} \in \mathit{ext}(S) :\ b \prec_{\mathit{spo}} a)\\ &\Longleftrightarrow \big((\exists \mathit{spo} \in \mathit{ext}(S) :\ a \prec_{\mathit{spo}} b) \wedge (\exists \mathit{spo} \in \mathit{ext}(S) :\ b \prec_{\mathit{spo}} a)\big)\\ &\qquad\qquad \vee (\exists \mathit{spo} \in \mathit{ext}(S) :\ a \frown_{\mathit{spo}} b)\\ &\Longleftrightarrow \exists \mathit{spo} \in \mathit{ext}(S) :\ a \frown_{\mathit{spo}} b\ .\end{aligned}$$

The second part follows from the first part and Thm. 6.22. The third part follows from Defn. 6.54. □

Consequently, when the stratified posets in $\mathit{ext}(S)$ are interpreted as observations of concurrent histories (see Section 6.3, and [14, 16]), then $(a,b) \in \mathit{sim}_S$ means that there is an observation in $\mathit{ext}(S)$ where a and b are executed simultaneously and $(a,b) \in \mathit{ser}_S$ means there are two *equivalent* observations (i.e., both belonging to $\mathit{ext}(S)$) where according to one a and b are executed simultaneously, while the other states that b follows a.

Since each $\mathit{spo} \in \mathit{ext}(S)$ can be interpreted as a step sequence, we can define a relational structure from spo similarly to the definition of $S^{\langle u\rangle}$

from a step sequence u (and Θ), but this time in a much simpler way. More precisely, for each $spo \in ext(S)$, we define

$$S^{\langle spo\rangle} \overset{\text{df}}{=} \left(X\ ,\prec_{spo}\backslash ser_S\ ,\frown_{spo}\backslash ser_S^{-1}\right)\ .$$

This $S^{\langle spo\rangle}$ is an so-structure with the relations $\prec_{spo}\backslash ser_S$ and $\frown_{spo}\backslash ser_S^{-1}$ playing roles similar to Q^u and R^u except that $\diamondsuit$-closure is *not* needed. More precisely,

Proposition 6.56. *For every $spo \in ext(S)$,*

$$\begin{array}{lclcl}\prec_{spo}\backslash ser_S & = & \prec & = & \prec\backslash ser_S\\ \frown_{spo}\backslash ser_S^{-1} & = & \sqsubset & = & \sqsubset\backslash ser_S^{-1}\end{array}$$

and so $S^{\langle spo\rangle} = (X, \prec, \sqsubset)$.

Proof. We show the first equality using Prop. 6.55(3) and Thm. 6.22:

$$\begin{array}{rcl} a \prec_{spo} b \wedge (a,b) \notin ser_S & \iff & a \prec_{spo} b \wedge (a \prec b \vee b \sqsubset a)\\ & \iff & (a \prec_{spo} b \wedge a \prec b) \vee (a \prec_{spo} b \wedge b \sqsubset a)\\ & \iff & a \prec b \vee \mathit{false}\ .\end{array}$$

For the second equality, we have $\prec\backslash ser_S = (\prec_{spo}\backslash ser)\backslash ser_S = \prec_{spo}\backslash ser_S = \prec$. The third equality again follows from Prop. 6.55(3) and Thm. 6.22:

$$\begin{array}{rcl} a \frown_{spo} b \wedge (b,a) \notin ser_S & \iff & a \frown_{spo} b \wedge (b \prec a \vee a \sqsubset b)\\ & \iff & (a \frown_{spo} b \wedge b \prec a) \vee (a \frown_{spo} b \wedge a \sqsubset b)\\ & \iff & \mathit{false} \vee a \sqsubset b\ .\end{array}$$

The last equality follows immediately. □

Consequently, for all so-structures S we have that $S = S^{\langle spo\rangle}$ for every $spo \in ext(S)$. Proposition 6.56 can be interpreted as a generalisation of Prop. 6.6 by which a poset can be reconstructed from any one of its total extensions provided its incomparability relation is known.

We conclude by proving that for all so-structures S, the set $ext(S)$, when interpreted as a set of step sequences, is a comtrace. Moreover, the so-structure defined by this comtrace is S itself. First, for every so-structure $S = (X, \prec, \sqsubset)$:

$$\Theta_S \overset{\text{df}}{=} (X, sim_S, ser_S) \quad\text{and}\quad \mathfrak{C}(S) \overset{\text{df}}{=} \{stepseq(spo) \mid spo \in ext(S)\}\ .$$

Clearly, Θ_S is a comtrace alphabet which is assumed below. We will prove now that the set $\mathfrak{C}(S)$ comprising the step sequences representing the stratified poset extensions of S, forms a comtrace.

Theorem 6.57. *Let $S = (X, \prec, \sqsubset)$ be an so-structure, $spo \in ext(S)$ and $u = stepseq(spo)$. Then $S_{[\![u]\!]} = S^{\langle u\rangle} = S^{\langle spo\rangle} = S$ and $\mathfrak{C}(S) = [\![u]\!]$.*

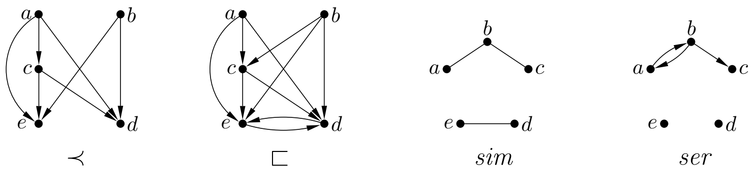


Fig. 6.5 Relations defining an so-structure and its comtrace alphabet.

Proof. First we need to check that u is a step sequence over Θ_S. Assume that $u = A_1 \dots A_n$ with each A_i a non-empty subset of X. Then, $a \frown_{spo} b$ whenever $a, b \in A_i$ and $a \neq b$. Thus we have $(a,b) \in sim_S$ by Prop. 6.55, which implies that A_i is indeed a step over Θ_S.
Hence we can construct $S^{\langle u \rangle}$ as in Defn. 6.50 and we have:

$$\begin{aligned}
S_{[\![u]\!]} &= S^{\langle u \rangle} && \text{by Thm. 6.53(3)}\\
&= \left(X, \prec_{spo} \setminus ser_S, \sqsubset_{spo} \setminus ser_S^{-1}\right)^{\diamondsuit} && \text{by Def. 6.50}\\
&= (X, \prec, \sqsubset)^{\diamondsuit} && \text{by Prop. 6.56}\\
&= (X, \prec, \sqsubset) = S && \text{by Thm. 6.48(4)}\\
&= S^{\langle spo \rangle} && \text{by Prop. 6.56}
\end{aligned}$$

From the above and Thm. 6.53(1,2), it now follows that $ext(S) = ext(S^{\langle u \rangle}) = \{(X, \prec_x) \mid x \in [\![u]\!]\}$. But this implies that, by the definition of $\mathfrak{C}(S)$, $[\![u]\!] = \{stepseq(X, \prec_x) \mid x \equiv u\} = \{stepseq(X, \prec_x) \mid (X, \prec_x) \in ext(S)\} = \mathfrak{C}(S)$. $\square$

By Thm. 6.57 we can call $\mathfrak{C}(S)$ the *comtrace generated by S*. The fact that $S_{[\![u]\!]} = S$ means that the so-structure defined by the comtrace $\mathfrak{C}(S)$ is exactly S, so comtraces and so-structures can be interpreted as equivalent or *tantamount* (cf. [14]) concurrent models.

Figure 6.5 shows an so-structure (through its $\prec$ and $\sqsubset$) with its induced simultaneity and serialisability relations *sim* and *ser*. It generates the comtrace: $[\![\{a,b\}c\{d,e\}]\!] = \big\{\{a,b\}c\{d,e\}, abc\{d,e\}, a\{b,c\}\{d,e\}, bac\{d,e\}\big\}$.

To conclude this section, we would like to add the following observation. Given the correspondence between step sequences and stratified posets, comtraces as equivalence classes of step sequences, can be viewed as sets of 'equivalent' stratified posets. From the theory presented in Section 6.3 one would expect this set to satisfy paradigm π_3 which expresses that unorderedness implies simultaneity (but not necessarily vice versa). As we have shown, the stratified poset extensions of the so-structure generated by a comtrace represent all step sequences forming the original comtrace and by Thm. 6.23 (see [16]) these do indeed together satisfy π_3.

6.6 Generalised Comtraces

There are realistic concurrent behaviours that cannot be modelled by comtraces.

> Let $E = \{a, b, c\}$ where a, b and c are three atomic operations respectively representing assignments:
>
> $$x \leftarrow x + 1 \qquad x \leftarrow x + 2 \qquad y \leftarrow y + 1$$
>
> It is reasonable to consider a, b, and c as 'independent' as any order of execution of two or more of them yields exactly the same result [14, 16]. Moreover, if simultaneous reading is allowed, then the steps $\{a, c\}$ and $\{b, c\}$ are allowed to occur. However, simultaneous execution of a, b, and c *is not* since simultaneous writing on the same variable is not permitted!
> The set of all equivalent executions (or runs) involving a single occurrence of each operation,
>
> $$\mathbf{x} = \Big\{abc, acb, bac, bca, cab, cba, \{a, c\}b, \{b, c\}a, b\{a, c\}, a\{b, c\}\Big\} ,$$
>
> is a valid concurrent history or behaviour [14, 16]. However $\mathbf{x}$ is *not* a comtrace. The problem is that we have $ab \equiv ba$ but $\{a, b\}$ *is not* a valid step.

We therefore consider an extension of comtraces to model the idea of 'unordered, but not simultaneous' relationship as discussed in the above example. This leads to the concept of *generalised comtraces* (or g-comtraces), again in the framework of equational monoids of step sequences.

We start with the notion of a *g-comtrace alphabet*

$$\Psi \overset{\text{df}}{=} (E, \mathit{sim}, \mathit{ser}, \mathit{inl}) ,$$

where E is an alphabet and *ser*, *sim* and *inl* are three relations on E, respectively called *serialisability*, *simultaneity* and *interleaving*, such that *sim* and *inl* are irreflexive and symmetric, $\mathit{ser} \subseteq \mathit{sim}$, and $\mathit{sim} \cap \mathit{inl} = \varnothing$. The interpretation of the relations *sim* and *ser* is as before, and $(a, b) \in \mathit{inl}$ means that a and b cannot occur simultaneously, but if they occur one after the other, the resulting orders are equivalent. As for comtraces, we define

$\mathbb{S}$ — the set of all (potential) steps over Ψ — as the set of all cliques of the relation *sim*.

Definition 6.58 (Generalised comtrace monoid). The *g-comtrace congruence* $\equiv_{ser,inl}$ over Ψ is the EQ-congruence generated by the set of equations $EQ = EQ_{ser} \cup EQ_{inl}$ where:

$$\begin{aligned} EQ_{ser} &\overset{\text{df}}{=} \{A = BC \mid A = B \cup C \in \mathbb{S} \ \wedge \ B \times C \subseteq ser\} \\ EQ_{inl} &\overset{\text{df}}{=} \{BA = AB \mid A \in \mathbb{S} \ \wedge \ B \in \mathbb{S} \ \wedge \ A \times B \subseteq inl\} \ . \end{aligned}$$

Then $(\mathbb{S}^*/_{\equiv_{ser,inl}}, \hat{\circ}, [\![\lambda]\!])$ is the *monoid of g-comtraces* over Ψ.

Note that since *ser* and *inl* are irreflexive relations, $(A = BC) \in EQ_{ser}$ implies $B \cap C = \varnothing$, and $(AB = BA) \in EQ_{inl}$ implies $A \cap B = \varnothing$. Moreover, since $inl \cap sim = \varnothing$, we have that $(AB = BA) \in EQ_{inl}$ implies $A \cup B \notin \mathbb{S}$.

By Prop. 6.10, the g-comtrace congruence relations can also be defined explicitly in a non-equational form.

Proposition 6.59. *Let $\approx_{ser,inl}$ be the relation comprising all pairs (t,u) of step sequences in $\mathbb{S}^*$ such that one of the following holds, for some steps sequences $w, z \in \mathbb{S}^*$ and steps $A, B, C \in \mathbb{S}$:*

- *$t = wAz$ and $u = wBCz$ where $B \cup C = A$ and $B \times C \subseteq ser$.*
- *$t = wABz$ and $u = wBAz$ where $A \times B \subseteq inl$.*

Then $\equiv_{ser,inl}$ is equal to $(\approx^{sym}_{ser,inl})^*$.

We will omit the subscript *ser*, *inl* from $\equiv_{ser,inl}$ and $\approx_{ser,inl}$, whenever this does not lead to ambiguity.

That generalised comtraces are indeed an extension of comtraces can be seen from the fact that whenever the relation *inl* is empty, Defn. 6.58 coincides with Defn. 6.33 of comtrace monoids. Hence comtraces can be regarded as a *special case* of generalised comtraces.

> The set of step sequences $\mathbf{x}$ from the example above is a g-comtrace with $E = \{a,b,c\}$, $ser = sim = \{(a,c),(c,a),(b,c),(c,b)\}$, $inl = \{(a,b),(b,a)\}$ and $\mathbb{S} = \{\{a,c\},\{b,c\},a,b,c\}$. As a result, we have that $\mathbf{x} = [\![\{a,c\}b]\!]$.

It is worth noting that there is an important difference between the equation $ab = ba$ for traces, and the equation $ab = ba$ for g-comtrace monoids. For traces, the equation $ab = ba$, when translated into step sequences, corresponds to two equations $\{a,b\} = ab$ and $\{a,b\} = ba$, which implies $ab \equiv \{a,b\} \equiv ba$. For g-comtrace monoids, the equation $ab = ba$

implies that $\{a,b\}$ *is not a step*, i.e., neither the equation $\{a,b\} = ab$ nor the equation $\{a,b\} = ba$ belongs to the set of equations. In other words, for traces the equation $ab = ba$ means 'independence', in the sense that executing a and b in any order is the same (equivalent) and thus will yield the same result. For g-comtrace monoids, the equation $ab = ba$ means that executing a and b in any order is the same (and yields the same result), but executing a and b in any order is *not* equivalent to executing them simultaneously.

As the next proposition states, g-comtraces have virtually the same algebraic properties as comtraces.

Proposition 6.60. *Let $u, v, w, s, t \in \mathbb{S}^*$, $a \in E$, and $D \subseteq E$.*

(1) $u \equiv v \implies wgt(u) = wgt(v)$. (step sequence weight equality)
(2) $u \equiv v \implies \#_a(u) = \#_a(v)$. (event preservation)
(3) $u \equiv v \implies u \div_R w \equiv v \div_R w$. (right cancellation)
(4) $u \equiv v \implies u \div_L w \equiv v \div_L a$. (left cancellation)
(5) $u \equiv v \iff sut \equiv svt$. (step subsequence cancellation)
(6) $u \equiv v \implies \pi_D(u) \equiv \pi_D(v)$. (projection rule)

Proof. For all parts except (5), it suffices to show that $u \approx v$ implies the right hand side of the formula. Notice that when $u \approx v$, the results for the case $u = xAy \approx xBCy = v$ follows from Prop. 6.37. So one only needs to consider explicitly $u = xABy$ and $v = xBAy$, where $A \times B \subseteq inl$ and $A \cap B = \varnothing$. The proofs are similar to those for Prop. 6.37. ☐

Unlike traces and comtraces, g-comtraces do not have a canonical form with a natural interpretation. The Greedy Maximally Concurrent approach (see Defn. 6.38) works for g-comtraces as well, but does not lead to a unique GMC-form and often does not even resemble what could intuitively be interpreted as 'maximally concurrent' behaviour. For more discussion of this issue, the reader is referred to [22].

Relationship with Generalised Stratified Order Structures. The relationship between g-comtraces and gso-structures is in principle the same as the relationship between comtraces and so-structures discussed in the previous section. Each g-comtrace uniquely determines a gso-structure and each gso-structure can be represented by a g-comtrace. However the proofs and even the formulations of those results are much more complex than in the case of the relationship between comtraces and so-structures. The difficulties stem mainly from the following observations:

- The definition of gso-structures is implicit, it involves using the induced so-structures (see Defn. 6.25), which makes practically all definitions much more complex (especially the counterpart of $\diamondsuit$-closure), and the use of Thm. 6.28 more difficult than the use of Thm. 6.22.
- An internal property like the one expressed by Thm. 6.23, which says that $ext(S)$ conforms to paradigm π_3 of [16], does not hold for gso-structures.
- There is no 'natural' canonical form for g-comtraces with a well understood interpretation.
- The relation *inl* introduces irregularities and substantially increases the number of cases that need to be considered in many proofs.

Therefore in this subsection we will show only the simplest proofs, only the most important results and the definitions that are really needed. In particular, we will not discuss the counterpart of $\diamondsuit$-closure for gso-structures, even though such a concept does exist. For details and proofs, the reader is referred to [22].

Let $\Psi = (E, sim, ser, inl)$ be a g-comtrace alphabet and let $u \in \mathbb{S}^*$ be a step sequence. Note that if $u \equiv x$ for a step sequence x, then $occ(u) = occ(x)$. Thus, for every g-comtrace $[\![u]\!]$, we can define $occ([\![u]\!]) \overset{\text{df}}{=} occ(u)$.

Definition 6.61 (Gso-structure defined by a g-comtrace). Let u be a step sequence in $\mathbb{S}^*$. Then the *gso-structure* induced by the g-comtrace $[\![u]\!]$ is given by $G_{[\![u]\!]} \overset{\text{df}}{=} \big(occ([\![u]\!]), \bigcap_{x\in[\![u]\!]} \prec_x^{sym}, \bigcap_{x\in[\![u]\!]} \sqsubset_x\big)$.

By the following theorem, $G_{[\![u]\!]}$ is a well defined gso-structure. Moreover, as a counterpart of Thm. 6.53, it states that the set of stratified poset extensions of $G_{[\![u]\!]}$ represents exactly the elements of the g-comtrace $[\![u]\!]$.

Theorem 6.62. *Let $u, v \in \mathbb{S}^*$.*

(1) $G_{[\![u]\!]}$ is a gso-structure.
(2) $u \equiv v \iff G_{[\![u]\!]} = G_{[\![v]\!]}$.
(3) $ext\big(G_{[\![u]\!]}\big) = \big\{(occ(x), \prec_x) \mid x \in [\![u]\!]\big\}$.

Figure 6.6 shows a g-comtrace alphabet $\Psi = (\{a, b, c, d\}, sim, ser, inl)$ and the relations of the gso-structure $G_{[\![\{a,b\}c\{a,d\}]\!]} = (X, \rightleftharpoons, \sqsubset)$ defined by the g-comtrace:

$$[\![\{a,b\}c\{a,d\}]\!] = \left\{ \begin{array}{lll} \{a,b\}c\{a,d\} & abc\{a,d\} & a\{b,c\}\{a,d\} \\ bac\{a,d\} & bca\{a,d\} & \{b,c\}a\{a,d\} \end{array} \right\}.$$

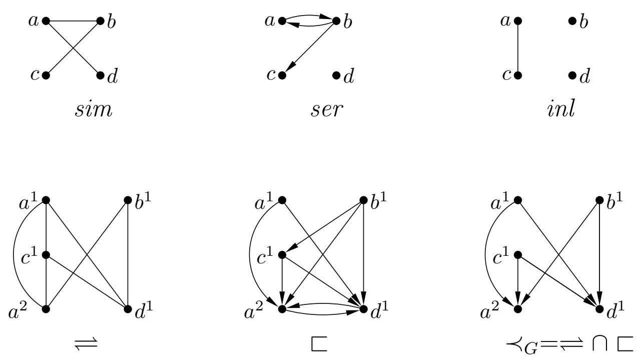


Fig. 6.6 Relations defining a g-comtrace alphabet, relations of a gso-structure G and $\prec_G$.

We will now show that every gso-structure can be represented by a g-comtrace. We start with the definition of the causality relations derivable from gso-structures.

Definition 6.63 (Causality relations in gso-structures). The simultaneity, serialisability and interleaving relations *induced* by the gso-structure $G = (X, \rightleftharpoons, \sqsubset)$ are defined below, for all distinct $a, b \in X$:

$$\begin{aligned}
(a,b) \in sim_G &\overset{df}{\iff} \neg(a \rightleftharpoons b)\\
(a,b) \in ser_G &\overset{df}{\iff} \neg(a \rightleftharpoons b) \wedge \neg(b \sqsubset a)\\
(a,b) \in inl_G &\overset{df}{\iff} a \rightleftharpoons b \wedge \neg(a \sqsubset b \vee b \sqsubset a)\ .
\end{aligned}$$

The above is a generalisation of Defn. 6.54 as shown in the following proposition. Following Defn. 6.25, we use $\prec_G$ to denote $\rightleftharpoons \cap \sqsubset$.

Proposition 6.64. *For all $a, b \in X$, we have:*

$$\begin{aligned}
(a,b) \in sim_{S_G} &\iff a \frown_{(X,\prec_G)} b\\
(a,b) \in ser_{S_G} &\iff a \frown_{(X,\prec_G)} b \wedge \neg(b \sqsubset a)
\end{aligned}$$

The terminology used in Defn. 6.63 is justified by the following result which shows the connection between the three relations there and the stratified extensions of the original gso-structure.

Proposition 6.65. *For all $a, b \in X$, we have:*

$$\begin{aligned}
&(a,b) \in \mathit{sim}_G \iff \exists \mathit{spo} \in \mathit{ext}(G):\ a \frown_{\mathit{spo}} b \\
&(a,b) \in \mathit{ser}_G \iff (a,b) \in \mathit{sim}_G \wedge (\exists \mathit{spo} \in \mathit{ext}(G):\ a \prec_{\mathit{spo}} b) \\
&(a,b) \in \mathit{inl}_G \iff (a,b) \notin \mathit{sim}_G \wedge (\exists \mathit{spo} \in \mathit{ext}(G):\ a \prec_{\mathit{spo}} b) \wedge \\
&\qquad\qquad\qquad\qquad\qquad\qquad (\exists \mathit{spo} \in \mathit{ext}(G):\ b \prec_{\mathit{spo}} a) \\
&(\exists \mathit{spo} \in \mathit{ext}(G):\ a \prec_{\mathit{spo}} b) \wedge (a,b) \notin \mathit{ser}_G \\
&\qquad \iff (\forall \mathit{spo} \in \mathit{ext}(G):\ a \prec^{\mathit{sym}}_{\mathit{spo}} b) \quad \iff \quad a \rightleftharpoons b\ .
\end{aligned}$$

Proof. The first three parts follow from Thm. 6.28, and the last one from the first two parts and Thm. 6.28. □

We will now define a relational structure $G^{\langle \mathit{spo} \rangle}$ based on a single stratified poset *spo* similar to the definition of the so-structure $S^{\langle \mathit{spo} \rangle}$. Let $G = (X, \rightleftharpoons, \sqsubset)$ be a gso-structure and $\mathit{spo} \in \mathit{ext}(G)$. We define:

$$G^{\{\mathit{spo}\}} \stackrel{\mathrm{df}}{=} \big(X, (\prec_{\mathit{spo}} \backslash \mathit{ser}_G)^{\mathit{sym}} \cup \mathit{inl}_G, \frown_{\mathit{spo}} \backslash (\mathit{ser}_G^{-1} \cup \mathit{inl}_G)\big)\ .$$

As the following proposition shows, $G^{\langle \mathit{spo} \rangle}$ and $G = (X, \rightleftharpoons, \sqsubset)$ are identical gso-structures, which is similar to the relationship between so-structures $S^{\langle \mathit{spo} \rangle}$ and S. It is interesting to observe that we do not need commutative closure to build G from a stratified poset $\mathit{spo} \in \mathit{ext}(G)$.

Proposition 6.66. *For every* $\mathit{spo} \in \mathit{ext}(G)$, $G^{\langle \mathit{spo} \rangle} = (X, \rightleftharpoons, \sqsubset)$.

Proof. First we show that $(\prec_{\mathit{spo}} \backslash \mathit{ser}_G)^{\mathit{sym}} \cup \mathit{inl}_G$ is equal to $\rightleftharpoons$. Indeed, for all $a, b \in X$, we have:

$$\begin{aligned}
&(a,b) \in (\prec_{\mathit{spo}} \backslash \mathit{ser}_G)^{\mathit{sym}} \cup \mathit{inl}_G \\
&\quad \iff (a \prec_{\mathit{spo}} b \wedge (a,b) \notin \mathit{ser}_G) \vee \\
&\qquad\quad (b \prec_{\mathit{spo}} a \wedge (b,a) \notin \mathit{ser}_G) \vee (a,b) \in \mathit{inl}_G \\
&\quad \iff a \rightleftharpoons b \vee (a,b) \in \mathit{inl}_G && \text{by Prop. 6.65(4)} \\
&\quad \iff a \rightleftharpoons b \vee \big(a \rightleftharpoons b \wedge \neg(b \sqsubset a \vee a \sqsubset b)\big) && \text{by Def. 6.63(3)} \\
&\quad \iff a \rightleftharpoons b\ .
\end{aligned}$$

Next we show that $\frown_{\mathit{spo}} \backslash (\mathit{ser}_G^{-1} \cup \mathit{inl}_G)$ is equal to $\sqsubset$. Note that $\mathit{inl}_G \cap \mathit{ser}_G = \varnothing$, and then, for all $a, b \in X$, we have:

$$\begin{aligned}
&(a,b) \in \frown_{\mathit{spo}} \backslash (\mathit{ser}_G^{-1} \cup \mathit{inl}_G) \\
&\quad \iff a \frown_{\mathit{spo}} b \wedge \big(a \rightleftharpoons b \vee a \sqsubset b\big) \\
&\qquad\quad \wedge \big(\neg(a \rightleftharpoons b) \vee a \sqsubset b \vee b \sqsubset a\big) && \text{by Def. 6.63} \\
&\quad \iff a \frown_{\mathit{spo}} b \wedge \big(a \sqsubset b \vee (a \rightleftharpoons b \wedge b \sqsubset a)\big) \\
&\quad \iff (a \frown_{\mathit{spo}} b \wedge a \sqsubset b) \vee (a \frown_{\mathit{spo}} b \wedge b \prec_G a) \\
&\quad \iff a \sqsubset b \vee \mathit{false} && \text{by Thm. 6.28}
\end{aligned}$$

Hence $G^{\langle \mathit{spo} \rangle} = (X, \rightleftharpoons, \sqsubset)$. □

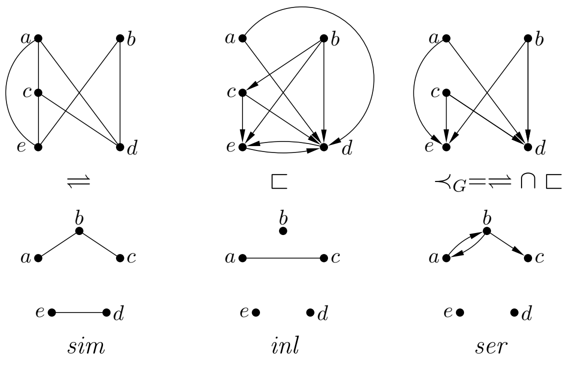


Fig. 6.7 A gso-structure $G = (X, \rightleftharpoons, \sqsubset)$, where $X = \{a, b, c, d, e\}$ and the relations $\prec_G$, sim_G, ser_G and inl_G.

Before stating the theorem that fully characterises g-comtraces defined by gso-structures, we define, for every gso-structure $G = (X, \rightleftharpoons, \sqsubset)$,

$$\Psi_G \overset{\text{df}}{=} (X, sim_G, ser_G, inl_G) \quad \text{and} \quad \mathfrak{gC}(G) \overset{\text{df}}{=} \{stepseq(spo) \mid spo \in ext(G)\} \ .$$

Note that $ser_G \subseteq sim_G$, the relations *sim* and *inl* are symmetric, $sim_G \cap inl_G = \varnothing$, and all three relations are irreflexive, so Ψ_G is a *g-comtrace alphabet*. Hence we can define the relations $\approx_{ser,inl}$ and $\equiv_{ser,inl}$ with respect to the g-comtrace alphabet Ψ_G. We will call $\mathfrak{gC}(G)$ the g-comtrace *generated* by the gso-structure G which is justified by the following result.

Theorem 6.67. *For all gso-structures G and stratified posets $spo \in ext(G)$, $G_{[\![stepseq(spo)]\!]} = G^{\langle spo \rangle} = G$ and $\mathfrak{gC}(G) = [\![stepseq(spo)]\!]$.*

Together with Thm. 6.62, Thm. 6.67 ensures that g-comtraces and so-structure are in fact equivalent models. It is illustrated in Figure 6.7, where the gso-structure G defines g-comtrace

$$[\![\{a,b\}c\{d,e\}]\!] = \left\{ \begin{array}{lll} \{a,b\}c\{d,e\} & abc\{d,e\} & a\{b,c\}\{d,e\} \\ bac\{d,e\} & bca\{d,e\} & \{b,c\}a\{d,e\} \end{array} \right\} .$$

6.7 Elementary Net Systems

In the last two sections, we will explain how the theory presented can be applied in the case of a concrete system model.

It is generally acknowledged that concepts related to fundamental notions of concurrency theory, such as causality and independence, can be

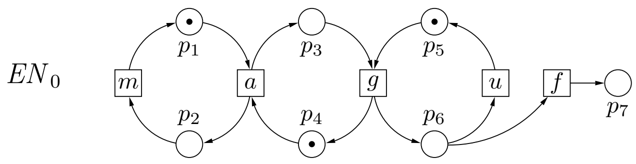


Fig. 6.8 A net with marking (EN-system) modelling a producer/consumer system.

well explained using the framework provided by Petri nets [39] (see also `http://www.informatik.uni-hamburg.de/TGI/PetriNets/`). A particularly convenient class of nets in this respect are Elementary Net systems [42] which are the most basic class of Petri nets.

Definition 6.68 (Nets). A *net* N is a triple (P, T, F) where P and T are disjoint finite sets of nodes, called respectively *places* and *transitions*, and $F \subseteq (T \times P) \cup (P \times T)$ is the *flow* relation. A *marking* of the net is a set of places. The *inputs* and *outputs* of a node x are the sets ${}^\bullet x$ and $x^\bullet$ of all y such that $(y, x) \in F$ and $(x, y) \in F$, respectively; moreover, ${}^\bullet x^\bullet$ are both the inputs and outputs of x. It is assumed that ${}^\bullet a \neq \varnothing \neq a^\bullet$, for every transition a.

The dot-notation extends to sets of nodes in the usual way, e.g., ${}^\bullet X \overset{\text{df}}{=} \bigcup\{{}^\bullet x \mid x \in X\}$. In diagrams, places (local states) are represented by circles, transitions (actions) by rectangles, the flow relation by directed arcs, and a marking (global state) by tokens (small black dots) drawn inside places.

Figure 6.8 shows a net modelling a concurrent system consisting of a producer, a buffer of capacity one, and a consumer, as well as an (initial) marking $M = \{p_1, p_4, p_5\}$. The producer can execute two actions: m ($\underline{\text{m}}$aking an item), and a ($\underline{\text{a}}$dding a new item to the buffer). The consumer can execute three actions: g ($\underline{\text{g}}$etting an item from the buffer), u ($\underline{\text{u}}$sing the acquired item), and f ($\underline{\text{f}}$inishing the work). Positioned in-between the producer and consumer, the buffer can cyclically execute the a and g actions. Intuitively, the three components progress independently but any action shared by two components must be executed if both of them do so.

Having introduced the net structure of Elementary Net systems, we define their dynamic behaviour which can be expressed in terms of sequences of steps of transitions.

Definition 6.69 (Steps). A *step* of a net N is a set U of its transitions such that ${}^\bullet a^\bullet \cap {}^\bullet b^\bullet = \varnothing$, for all distinct $a, b \in U$. It is *enabled* at a marking M if ${}^\bullet U \subseteq M$ and $U^\bullet \cap M = \varnothing$. In such a case, the *execution* of U leads to marking $M' \overset{\text{df}}{=} (M \setminus {}^\bullet U) \cup U^\bullet$. We denote this by $M[U\rangle M'$.

A *step sequence* from a marking M to a marking M' is a possibly empty sequence $\sigma = U_1 \ldots U_n$ of non-empty steps U_i such that $M[U_1\rangle M_1, \ldots, M_{n-1}[U_n\rangle M'$, for some markings $M_1, \ldots, M_{n-1}$. We denote this by $M[\sigma\rangle M'$, and call M' *reachable* from M. When all steps U_i are singletons, σ is called a *firing sequence*.

For the net shown in Figure 6.8, we have $M[m\rangle M'$ and $M[\{m, f\}\rangle M''$, where $M = \{p_2, p_3, p_6\}$ and $M' = \{p_1, p_3, p_6\}$ and $M'' = \{p_1, p_3, p_7\}$. Moreover, $M[a\{m, g\}\{a, f\}m\rangle M'$ and $M[amgafm\rangle M'$ where $M = \{p_1, p_4, p_5\}$ and $M' = \{p_1, p_3, p_7\}$.

Definition 6.70 (EN-systems). An *elementary net system* (or EN-system) is a tuple $EN \overset{\text{df}}{=} (P, T, F, M_{init})$ such that the first three components form its underlying net, and $M_{init} \subseteq P$ is its initial marking. Moreover, *steps*(EN) and *fseq*(EN) comprise respectively all the step and firing sequences from the initial marking M_{init}.

The EN-system in Figure 6.8 is *contact-free.* which means that, for all markings M reachable from M_{init} and transitions a, ${}^\bullet a \subseteq M$ implies $a^\bullet \cap M = \varnothing$. It is important to note that contact-freeness can always be enforced without influencing the behaviour, (step sequences and firing sequences) by *complementing* (all or some) places p using fresh places $\widetilde{p}$ satisfying ${}^\bullet p = \widetilde{p}^\bullet$, $p^\bullet = {}^\bullet \widetilde{p}$, and declaring that $\widetilde{p} \in M_{init}$ *iff* $p \notin M_{init}$. We will henceforth make a simplifying assumption that:

Assumption 6.71. All EN-systems as well as their extensions are contact-free.

Figure 6.8 depicts an EN-system EN_0 such that: *steps*$(EN_0) = \big\{\lambda, a, ag, am, a\{g, m\}, \ldots\big\}$ and *fseq*$(EN_0) = \big\{\lambda, a, ag, am, \ldots\big\}$. Moreover, it has two pairs of complementary places: p_1 with p_2, and p_3 with p_4, i.e., $p_1 = \widetilde{p}_2$ and $p_3 = \widetilde{p}_4$.

Semantical Framework. If one aims at a systematic presentation of causality semantics for different classes $\mathcal{PN}$ of Petri nets, it pays off to use a common scheme as introduced in [28] and further elaborated on in [25] It is reproduced here as Figure 6.9, where N is a net in $\mathcal{PN}$ (for example, an EN-system EN) and:

- $\mathcal{EX}$ are executions of nets in $\mathcal{PN}$ (for example, step or firing sequences);
- $\mathcal{LAN}$ are labelled acyclic nets, such as occurrence nets soon to be introduced, each labelled net in $\mathcal{LAN}$ representing a single non-sequential history;
- $\mathcal{LEX}$ are labelled executions of nets in $\mathcal{LAN}$; and
- $\mathcal{LCS}$ are labelled causal structures capturing an abstract causality semantics of nets in $\mathcal{PN}$ (for EN-systems, these are simply posets).

$$\begin{tikzcd}
N \in \mathcal{PN} \arrow[r, "\alpha"] \arrow[dd, "\omega"'] & \mathcal{LAN} \arrow[dd, "\lambda"'] \arrow[dr, "\kappa"] & \\
& & \mathcal{LCS} \arrow[dl, bend left, "\epsilon"] \\
\mathcal{EX} \arrow[uur, "\pi_N"] & \mathcal{LEX} \arrow[l, "\phi"] \arrow[ur, dashed, "\imath"] &
\end{tikzcd}$$

Fig. 6.9 Semantical framework for a Petri net N in $\mathcal{PN}$ [28]. The bold arcs indicate mappings to powersets and the dashed arc indicates a partial function.

The maps in Figure 6.9 relate the different semantical views given by $\mathcal{EX}$, $\mathcal{LAN}$ and $\mathcal{LCS}$. Their intended roles are as follows:

- ω returns sets of executions, defining *operational* semantics of N;
- α returns sets of labelled acyclic nets, defining *axiomatic process* semantics of N;
- λ returns sets of *labelled* executions of the processes of N, after applying ϕ to each such labelled execution one should obtain an execution of N;
- π_N returns, for each execution of N, a non-empty set of labelled acyclic nets, defining *operational process* semantics of N;
- κ associates a labelled *causal* structure with each process of N; and
- ϵ and $\imath$ allow one to go back and forth between labelled causal structures and the sets of their labelled executions (for EN-systems, ϵ yields all the linearisations or stratifications of a given poset, while $\imath$ yields the intersection of a set of total or stratified posets with the same domain).

The overall goal is to demonstrate that the different semantical views agree in the sense that processes ($\mathcal{LAN}$) and causal structures ($\mathcal{LCS}$) describe relations between actions consistent with the chosen operational semantics ($\mathcal{EX}$), as captured by a series of results called *aims*. As there exist simple requirements (called *properties*) guaranteeing such an agreement, one can concentrate on defining the semantical domains and maps appearing in Figure 6.9 and proving 'properties', from which the 'aims' are bound to follow.

Some properties are rather simple, e.g., each net should allow at least one execution.

Property I (Soundness of mappings). The maps ω, α, λ, ϕ, $\pi_N|_{\omega(N)}$, κ, ϵ and $\imath|_{\lambda(\mathcal{LAN})}$ are total. Moreover, ω, α, λ, $\pi_N|_{\omega(N)}$ and ϵ always return non-empty sets.

The next property relates individual executions with individual processes.

Property II (Consistency). For all $\xi \in \mathcal{EX}$ and $LN \in \mathcal{LAN}$:

$$\left.\begin{array}{r} \xi \in \omega(N) \\ LN \in \pi_N(\xi) \end{array}\right\} \quad \textit{iff} \quad \left\{\begin{array}{l} LN \in \alpha(N) \\ \xi \in \phi(\lambda(LN)) \end{array}\right.$$

The above properties ensure that the axiomatic (defined through α) and operational (defined through $\pi_N \circ \omega$) process semantics of nets in $\mathcal{PN}$ are in full agreement. Moreover, the operational semantics of N (defined through ω) coincides with the operational semantics of the processes of N (defined through $\phi \circ \lambda \circ \alpha$).

Aim A. $\alpha = \pi_N \circ \omega$.

Proof. To show the ($\subseteq$) inclusion, suppose that $LN \in \alpha(N)$. Then, by Property I for λ and ϕ, there exists $\xi \in \phi(\lambda(LN))$. Hence, by Property II, $\xi \in \omega(N)$ and $LN \in \pi_N(\xi)$. Thus $LN \in \pi_N(\omega(N))$.

To show the ($\supseteq$) inclusion, suppose that $LN \in \pi_N(\omega(N))$. Then there exists $\xi \in \omega(N)$ such that $LN \in \pi_N(\xi)$. Hence, by Property II, $LN \in \alpha(N)$. $\square$

Aim B. $\omega = \phi \circ \lambda \circ \alpha$.

Proof. To show the ($\subseteq$) inclusion, suppose that $\xi \in \omega(N)$. Then, by Property I for π_N, there exists $LN \in \pi_N(\xi)$. Hence, by Property II, $LN \in \alpha(N)$ and $\xi \in \phi(\lambda(LN))$. Thus $\xi \in \phi(\lambda(\alpha(N)))$.

To show the ($\supseteq$) inclusion, suppose that $\xi \in \phi(\lambda(\alpha(N)))$. Then there exists $LN \in \alpha(N)$ such that $\xi \in \phi(\lambda(LN))$. Hence, by Property II, $\xi \in \omega(N)$. $\square$

Corollary 6.72. $\omega = \phi \circ \lambda \circ \pi_N \circ \omega$.

The other kind of agreement concerns the abstract causality semantics of processes captured within the triangle-like part of the diagram in Figure 6.9. First, we require that the executions returned by ϵ should always contain enough information to uniquely reconstruct the original labelled causal structure.

Property III (Representation). $\imath \circ \epsilon = id_{\mathcal{LCS}}$.

When $\mathcal{LCS}$ are posets and $\mathcal{LEX}$ are total (or stratified) posets, Property III is simply Szpilrajn's Theorem. Second, we require that the operational semantics for the structures in $\mathcal{LCS}$ fits with the operational semantics chosen for $\mathcal{LAN}$.

Property IV (Fitting). $\lambda = \epsilon \circ \kappa$.

With these properties, the causality in a process of N (defined through κ) coincides with the causality structure implied by its operational semantics (through $\imath \circ \lambda$).

Aim C. $\kappa = \imath \circ \lambda$.

Proof. By properties III and IV, $\kappa = id_{\mathcal{LCS}} \circ \kappa = \imath \circ \epsilon \circ \kappa = \imath \circ \lambda$. $\square$

We can also relate the operational semantics of the net N and the set of labelled causal structures associated with it, in effect joining together the two parts of the diagram in Figure 6.9.

Corollary 6.73. $\omega = \phi \circ \epsilon \circ \kappa \circ \alpha$.

Proof. By Aim B and Property IV, $\omega = \phi \circ \lambda \circ \alpha = \phi \circ \epsilon \circ \kappa \circ \alpha$. $\square$

In practice, to take advantage of the semantical framework all we need to do is to establish the soundness of the various mappings (Property I), and then verify the consistency, representation and fitting properties (Properties II, III and IV). Having done so, fundamental semantical characteristics (Aims A, B and C) follow.

Semantical Framework for EN-systems. We have already indicated how some notions relating to EN-systems instantiate the general concepts of the semantical framework. We will now introduce the missing parts, starting with the definition of a class of labelled acyclic nets which can be used to capture the causality semantics of EN-systems.

Definition 6.74 (Occurrence nets ($\mathcal{LAN}$)). An *occurrence net* is a relational tuple $ON \overset{\text{df}}{=} (P', T', F', \ell)$ such that (P', T', F') is an underlying net, ℓ is a labelling for $P' \cup T'$, F' is an acyclic flow relation, and $|{}^\bullet p| \leq 1$ and $|p^\bullet| \leq 1$, for every place p. The default *initial* M_{init}^{ON} and *final* M_{fin}^{ON} markings respectively consist of all places without inputs and outputs.
The dot-notations, markings, execution rule, etc, for ON are as those defined for the underlying net. The places of ON are often called *conditions* and transitions *events.*

Figure 6.10 shows an occurrence net labelled by places and transitions of the EN-system EN_0 shown in Figure 6.8, with the default initial and final markings $\{b_1, b_2, b_3\}$ and $\{b_{11}, b_{12}, b_{13}\}$, respectively.

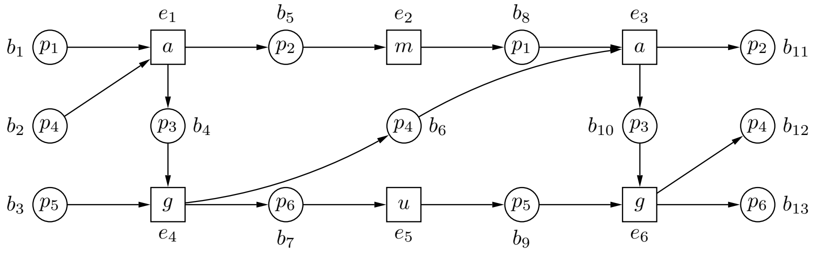


Fig. 6.10 An occurrence net ON_0 (labels are shown inside the nodes).

The behaviour of an occurrence net is captured by the set $\lambda_{step}(ON)$ of *labelled step sequences*, comprising all pairs $(\sigma, \ell|_{T'})$ such that σ is a step sequence from the default initial marking of ON to the default final marking. For each such labelled step sequence, $\phi(\sigma, \ell|_{T'}) \overset{\mathrm{df}}{=} \ell(\sigma)$. Moreover, $\lambda_{fseq}(ON)$ are the *labelled firing sequences* of ON, i.e., all the labelled step sequences $(\sigma, \ell|_{T'})$ such that σ is a sequence of singleton steps. Note that, due to the acyclicity of the flow relation and the lack of multiple inputs (or outputs) of places, means that each transition in T' appears exactly *once* in any labelled step sequence of ON.

In an occurrence net ON, the acyclic relation $(F' \circ F')|_{T' \times T'}$ represents direct causal relationships between net transitions. As a result, the relational structure $po(ON) \overset{\mathrm{df}}{=} (T', ((F' \circ F')|_{T' \times T'})^+, \ell|_{T'})$ is a poset representing all, direct and indirect, causal dependencies between the transitions in T'.

For the occurrence net of Figure 6.10, we have $a\{m, g\}\{a, u\}g \in \phi(\lambda_{step}(ON_0))$ as well as $amgaug \in \phi(\lambda_{step}(ON_0))$. Moreover, we have that e_4 causes e_5 directly, but there is only an indirect causal link from e_4 to e_6. Also, there is no causal link between e_2 and e_5 which means that they are concurrent or independent. This and other relationships can be read out from the diagram of the relation $(F' \circ F')|_{T' \times T'}$ shown below.

$$\begin{tikzcd}
\overset{e_1}{\boxed{a}} \arrow[r] \arrow[d] & \overset{e_2}{\boxed{m}} \arrow[r] & \overset{e_3}{\boxed{a}} \arrow[d] \\
\underset{e_4}{\boxed{g}} \arrow[r] \arrow[urr] & \underset{e_5}{\boxed{u}} \arrow[r] & \underset{e_6}{\boxed{g}}
\end{tikzcd}$$

To define processes of an EN-system, we need to provide an axiomatic characterisation of occurrence nets consistent with the structure of a given EN-system.

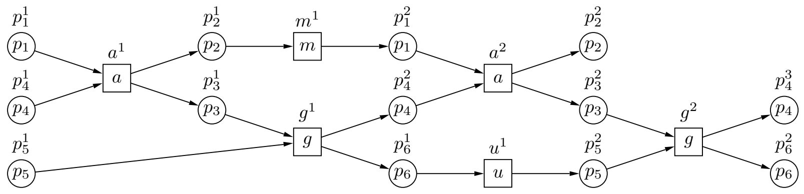


Fig. 6.11 Process generated for EN_0 and its step sequence $\sigma \overset{\mathrm{df}}{=} a\{m,g\}\{a,u\}g$.

Definition 6.75 (Processes of EN-systems (α)). A *process* of EN is an occurrence net ON with the labelling ℓ which:

- labels places of ON with places of EN.
- labels transitions of ON with transitions of EN.
- is injective on M_{init}^{ON} and $\ell(M_{init}^{ON}) = M_{init}$.
- is injective on ${}^\bullet a$ and $a^\bullet$ and, moreover, $\ell({}^\bullet a) = {}^\bullet\ell(a)$ and $\ell(a^\bullet) = \ell(a)^\bullet$, for every transition a of ON.

We denote this by $ON \in proc(EN)$.

The only missing component of the semantical framework for EN-systems is the mapping returning processes for individual step sequences.

Definition 6.76 (Processes construction (π_{EN})). The occurrence net $proc_{EN}(\sigma)$ *generated* by a step sequence $\sigma = U_1 \dots U_n$ of EN is the last element in the sequence $ON_0, \dots, ON_n$ where each ON_k is an occurrence net (P_k, T_k, F_k, ℓ_k) constructed thus.

Step 0: $P_0 \overset{\mathrm{df}}{=} \{p^1 \mid p \in M_{init}\}$ and $T_0 = F_0 \overset{\mathrm{df}}{=} \varnothing$.

Step k: Given ON_{k-1} we extend the sets of nodes and arcs as follows:

$$\begin{aligned}
P_k &\overset{\mathrm{df}}{=} P_{k-1} \cup \{p^{1+\triangle p} \mid p \in U_k^\bullet\} \\
T_k &\overset{\mathrm{df}}{=} T_{k-1} \cup \{a^{1+\triangle a} \mid a \in U_k\} \\
F_k &\overset{\mathrm{df}}{=} R_{k-1} \cup \{(p^{\triangle p}, a^{1+\triangle a}) \mid a \in U_k \wedge p \in {}^\bullet a\} \\
&\qquad \cup \{(a^{1+\triangle a}, p^{1+\triangle p}) \mid a \in U_k \wedge p \in a^\bullet\}\ .
\end{aligned}$$

In the above, the label of each node x^i is set to be x, and $\triangle x$ denotes the number of the nodes of ON_{k-1} labelled by x.

The occurrence net ON_0 of Figure 6.10 is a process of the EN-system of Figure 6.8.

Process construction from the last definitions is illustrated in Figure 6.11 for the EN-system EN_0 of Figure 6.8 and one of its step sequences. The resulting occurrence net is isomorphic to ON_0 of Figure 6.10 which, as we already noted, is a process of EN_0.

We will now show the semantical properties formulated above can be established for EN-systems and their firing sequences. Referring to Figure 6.9, we have the following, where: EN is an EN-system, ON an occurrence net, (σ, ℓ) a labelled firing sequence, po a poset, and $\mathcal{PO}$ a set of total posets with the same domain:

$\mathcal{PN}$	are	EN-systems	Defn. 6.70
$\mathcal{EX}$	are	firing sequences	
$\mathcal{LAN}$	are	occurrence nets	Defn. 6.74
$\mathcal{LEX}$	are	labelled firing sequences	
$\mathcal{LCS}$	are	labelled posets	
$\omega(EN)$	is	$\mathit{fseq}(EN)$	
$\alpha(EN)$	is	$\mathit{proc}(EN)$	Definition 6.75
$\lambda(ON)$	is	$\lambda_{\mathit{fseq}}(ON)$	
$\pi_{EN}(\sigma)$	is	$\mathit{proc}_{EN}(\sigma)$	Defn. 6.76
$\phi(\sigma, \ell)$	is	$\ell(\sigma)$	
$\kappa(ON)$	is	$\mathit{po}(ON)$	
$\epsilon(po)$	is	$\mathit{lin}(po)$	
$\imath(\mathcal{PO})$	is	$\bigcap \mathcal{PO}$.	

We first show, after omitting some obvious facts, that Property I holds.

Theorem 6.77. *Let EN be an EN-system and σ its firing sequence, ON be an occurrence net, po be a poset, and $\mathcal{PO}$ be a set of total posets with the same domain.*

(1) $\mathit{fseq}(EN)$, $\mathit{proc}(EN)$, $\lambda_{\mathit{fseq}}(ON)$ and $\mathit{lin}(po)$ are non-empty sets.
(2) $\mathit{po}(ON)$ and $\bigcap \mathcal{PO}$ are posets.
(3) $\mathit{proc}_{EN}(\sigma)$ is an occurrence net.

Proof. (1) Clearly, $\mathit{fseq}(EN) \neq \varnothing$ as the empty string is a firing sequence of EN. To show that $\mathit{proc}(EN) \neq \varnothing$ one can take the occurrence net consisting of the initial marking of EN with the identity labelling and no transitions. That $\mathit{lin}(po) \neq \varnothing$ follows from Thm. 6.4.

To show $\lambda_{\mathit{fseq}}(ON) \neq \varnothing$ we proceed by induction on the number of transitions in ON. In the base case, there are no transitions at all, and the empty labelled firing sequence belongs to $\lambda_{\mathit{fseq}}(ON)$. In the inductive case, one takes a transition a with all the inputs belonging to M_{init}^{ON} (such a transition does exists as ON is finite and acyclic). Transition a can be executed leading to a new marking M. We then delete a and all its inputs obtaining an occurrence net ON' with the default initial marking M to which the induction hypothesis can be applied. The desired labelled firing sequence of ON can then be obtained by pre-pending any labelled firing sequence of ON' with a.

(2) That $po(ON)$ is a poset follows from the acyclicity of ON, and that $\bigcap \mathcal{PO}$ is a poset follows from Thm. 6.4.

(3) The acyclicity of the constructed net follows directly from the construction of Defn. 6.76 (intuitively, all arrows point 'forward'). The same construction also guarantees that each place has at most one input, and at most one output. □

We next show that Property II holds.

Theorem 6.78. *Let EN be an EN-system, and σ be its firing sequence. Moreover, let ON be a process of EN, and $\sigma' \in \phi(\lambda_{fseq}(ON))$.*

(1) $proc_{EN}(\sigma)$ is a process of EN.
(2) $\sigma' \in fseq(EN)$ and $ON = proc_{EN}(\sigma')$.

Proof. (1) Follows directly from the construction in Defn. 6.76.

(2) Let $\sigma' = \phi((a_1 \dots a_n, \ell))$. Then there are markings $M_1, \dots, M_{n-1}$ such that $M_{init}^{ON}[a_1\rangle M_1 \dots M_{n-1}[a_n\rangle M_{fin}^{ON}$. One can then see that, by Defn. 6.75, $\ell(M_{init}^{ON})$ is the initial marking of EN and

$$\ell(M_{init}^{ON})[\ell(a_1)\rangle \ell(M_1) \dots \ell(M_{n-1})[\ell(a_n)\rangle \ell(M_{fin}^{ON})$$

in EN. Hence $\sigma' = \ell(a_1) \dots \ell(a_n) \in fseq(EN)$. We still need to show that ON is isomorphic to $proc_{EN}(\sigma')$. This can be shown by taking a mapping ψ mapping each transition z^i in $proc_{EN}(\sigma')$ to the i-th transition in $a_1 \dots a_n$ labelled with z, and then extending it bijectively to the inputs of z^i and $\psi(z^i)$, and the output places of z^i and $\psi(z^i)$. One can show that ψ is an isomorphism for ON and $proc_{EN}(\sigma')$. □

Since Property III is nothing but Szpilrajn's theorem in this case, we do not have to do anything, and so what follows next is a proof of Property IV.

Theorem 6.79. *If ON is an occurrence net then $\lambda_{fseq}(ON) = lin(po(ON))$.*

Proof. Referring to Defn. 6.74, if a and b are transitions satisfying $(a,b) \in F' \circ F'$ then, by the fact each place has at most one input and at most one output, a must appear before b in any labelled firing sequence of ON. Hence the ($\subseteq$) inclusion holds. To show the reverse one, let σ be a firing sequence defined by a total poset in $lin(po(ON))$. Then one can apply similar induction as in the second part of the proof of Thm. 6.77, after observing that the first transition in any total poset in $lin(po(ON))$ is such that all its inputs belong to M_{init}^{ON}, and so it can be executed. $\square$

As a result, we can claim the semantical aims for EN-systems and firing sequences.

Theorem 6.80. *Let EN be an EN-system, and ON be an occurrence net.*

$$\begin{array}{rcl} proc(EN) & = & proc_{EN}(fseq(EN)) \\ fseq(EN) & = & \phi(\lambda_{fseq}(proc(EN))) \\ po(ON) & = & \bigcap \lambda_{fseq}(ON)\ . \end{array}$$

Similarly, we may claim the semantical aims for EN-systems and step sequences. The necessary proofs are omitted as they are special cases of those for ENI-systems which will be provided in the next section.

Theorem 6.81. *Let EN be an EN-system, and ON be an occurrence net.*

$$\begin{array}{rcl} proc(EN) & = & proc_{EN}(steps(EN)) \\ steps(EN) & = & \phi(\lambda_{step}(proc(EN))) \\ po(ON) & = & \bigcap \lambda_{step}(ON)\ . \end{array}$$

EN-systems and Traces. In the last part of this section, we describe a very close relationship between the behaviours of an EN-system $EN = (P, T, F, M_{init})$ and the theory of traces. We start by defining the trace alphabet of EN, $\Gamma_{EN} \overset{\mathrm{df}}{=} (T, ind_{EN})$, where ind_{EN} comprises all pairs of transitions with disjoint neighbourhoods. The set of possible steps of EN is given by:

$$\mathbb{S}_{EN} \overset{\mathrm{df}}{=} \left\{ U \subseteq T \mid \forall a \neq b \in U :\ (a,b) \in ind_{EN} \right\} .$$

Note that the independence relation ind_{EN} is a structurally defined notion.

The trace alphabet of the EN-system in Figure 6.8 comprises pairs like (m,u) and (f,a), but it does not contain the pair (a,g) nor (u,f). Moreover, $\mathbb{S}_{EN_0} = \{\{m\}, \{a\}, \{g\}, \{u\}, \{f\}, \{m,g\}, \{m,u\}, \{m,f\}, \{a,u\}, \{a,f\}\}$.

We first consider Mazurkiewicz traces which correspond to the executions of firing sequences of EN. Then the induced equivalence on firing sequences is given by equations of the form ($ef = fe$), for all $(e, f) \in ind_{EN}$. One can easily check that each Mazurkiewicz trace gives rise to only one process of EN, since interchanging adjacent occurrences of independent transitions has no effect on the construction of a process in Defn. 6.76, i.e., if $\sigma \equiv \sigma'$ then $proc_{EN}(\sigma)$ is exactly the same[4] as $proc_{EN}(\sigma')$. Hence $proc_{EN}([\![\sigma]\!]) \overset{\text{df}}{=} proc_{EN}(\sigma)$ is a well defined occurrence net (process) associated to a trace. And, conversely, if we take all firing sequences of a process from its default initial to default final marking and apply the labelling, then what we get is exactly its defining trace. All this leads to a conclusion that there exists *a one-to-one* correspondence between the Mazurkiewicz traces and processes of an EN-system. In general, we have that the following are satisfied:

Theorem 6.82. *Let EN be an EN-system and σ be its firing sequence.*

$$\begin{aligned} fseq(EN) &= \biguplus_{\theta \in fseq(EN)} [\![\theta]\!] \\ canposet([\![\sigma]\!]) &= \kappa(proc_{EN}(\sigma)) \\ [\![\sigma]\!] &= \lambda_{fseq}(proc_{EN}(\sigma)) \ . \end{aligned}$$

Proof. The first part follows from the fact that if $(a, b) \in ind_{EN}$ and $M[ab\rangle M'$, then $M[ba\rangle M'$. The second part follows from the way in which the construction in Defn. 6.76 works. The third part follows from the second part and Thm. 6.79. $\square$

To conclude, the Mazurkiewicz trace semantics and the process semantics of EN-systems lead to the same poset semantics by providing for each EN-system identical (isomorphic) posets modelling the causalities in its concurrent executions. This provides a strong argument in favour of the view that both approaches capture the essence of causality in the behaviours of EN-systems.

Similar conclusions can be reached if one considers step sequences of an ENI-system. This time, however, the equivalence on executed behaviours is generated by equations of the form $AB \equiv A \cup B$ for all disjoint $A, B \in \mathbb{S}_{EN}$ satisfying $(A \times B) \cap ind_{EN} = \varnothing$. The remaining details are similar, leading up to

Theorem 6.83. *Let EN be an EN-system and σ be its step sequence.*

[4]I.e., in the sense of an actual equality of the two nets rather than isomorphism.

$$\begin{aligned} steps(EN) &= \biguplus_{\theta \in steps(EN)} [\![\theta]\!] \\ canposet([\![\sigma]\!]) &= po(proc_{EN}(\sigma)) \\ [\![\sigma]\!] &= \lambda_{step}(proc_{EN}(\sigma)) \ . \end{aligned}$$

Proof. This is a special case of Thm. 6.92 which holds for ENI-systems and their step sequences. □

6.8 EN-systems with Inhibitor and Mutex Arcs

This section extends the treatment of concurrency in system models to nets with inhibitor arcs. Let us take the EN-system of Figure 6.8 and add to it an inhibitor arc linking the place p_3 and transition f. This yields the net system ENI_0 shown in Figure 6.12. (Inhibitor arcs are drawn with small open circles as arrowheads.) Adding such an arc means that f cannot be enabled when the buffer is non-empty (a token in place p_3 signifies that the buffer contains an item).

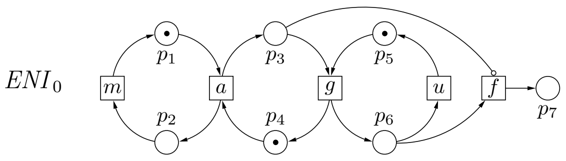


Fig. 6.12 An ENI-systems modelling a second version of the producer/consumer system.

Definition 6.84 (ENI-systems ($\mathcal{PN}$)). An *elementary net system with inhibitor arcs* (or ENI-system) is a relational tuple $ENI \overset{\text{df}}{=} (P, T, F, M_{init}, Inh)$ such that the first four components form an underlying EN-system and $Inh \subseteq P \times T$ is a set of *inhibitor* arcs.

Generally speaking, notions and notations relating to an ENI-system are inherited from its underlying EN-system. The only new notation is $^\circ a$ denoting the set of all the places p where the presence of a token inhibits the enabling of a transition a, i.e., $(p, a) \in Inh$. The dynamic aspects of an ENI-system are also derived from the underlying EN-system after stating that a step of transitions U of an ENI-system is enabled at a marking M if it is enabled at M in the underlying EN-system and, in addition, no place in $^\circ U$ belongs to M, where $^\circ U$ consists of all places connected by inhibitor arcs to transitions in U. The change of state effected by an executed step is exactly the same as in the underlying EN-system.

For the ENI-system of Figure 6.12, we have that $M[\{a,f\}\rangle M'$ and $M[fa\}\rangle M'$, where $M = \{p_1,p_4,p_6\}$ and $M' = \{p_2,p_3,p_7\}$. However, $M[af\}\rangle M'$ does not hold as after executing transition a, a token is deposited in place p_3 which acts as an inhibitor place of transition f.

To develop a causality semantics for ENI-systems we again take advantage of a suitably instantiated semantical framework, following what we have already done for EN-systems.

The labelled causal structures employed by the semantical framework instantiated for ENI-systems are the stratified order structures, while executions remain to be step sequences. To define processes we extend occurrence nets to handle nets with inhibitor arcs. (In [25] an alternative process definition for inhibitor nets is developed.) To this end we introduce so-called activator arcs. Each such arc plays a role dual to that of an inhibitor arc. An activator arc between a place and transition test for the $\underline{\text{presence}}$ of a token in the place, but this token is not affected (removed) by the occurrence of that transition.

Definition 6.85 (Activator occurrence nets ($\mathcal{LAN}$)). An *activator occurrence net* (or ao-net) is a relational tuple

$$AON \overset{\mathrm{df}}{=} (P', T', F', \ell, Act)$$

such that the first four components form an underlying occurrence net and $Act \subseteq P' \times T'$ is a set of *activator* arcs (drawn with small black circles as arrowheads). Moreover, it is assumed that the relational structure

$$\rho \overset{\mathrm{df}}{=} (T', \prec_{loc}, \sqsubset_{loc}, \ell|_{T'})$$

where $\prec_{loc}$ and $\sqsubset_{loc}$ are relations respectively given by

$$(F' \circ F')|_{T' \times T'} \cup (F' \circ Act) \quad \text{and} \quad Act^{-1} \circ F'$$

is $\diamondsuit$-acyclic. We then define $sos(AON)$ to be the $\diamondsuit$-closure of ρ.

An ao-net represents a concurrent execution or run of a system and so it has to avoid circularity. Intuitively, $\prec_{loc}$ stands for precedence (the first transition has to produce a token for consumption or testing by the second transition) and $\sqsubset_{loc}$ for weak precedence (the first transition cannot happen after the second one, since the latter consumes a token for which the former tests). Clearly, $sos(AON)$ is an so-structure.

Step sequences of an ao-net are defined as for its underlying occurrence net, except that a step U is enabled at a marking M if, in addition, ${}^{\blacklozenge}U \subseteq M$ where ${}^{\blacklozenge}U$ consists of all places connected by activator arcs with transitions in U.

Figure 6.13 shows an ao-net AON_0 labelled by places and transitions of the ENI-system EN_0 shown in Figure 6.12, with the default initial marking $\{b_1, b_2, b_3\}$ and default final marking $\{b_{11}, b_{12}, b_{13}\}$. Note also that transition e_5 weakly precedes transition e_4, i.e., $e_5 \sqsubset_{loc} e_4$. Moreover, we have that $\{m, g\}\{a, f\}$ and $a\{m, g\}fa$ belong to $\phi(\lambda_{step}(AON_0))$, however, $a\{m, g\}af$ does not.

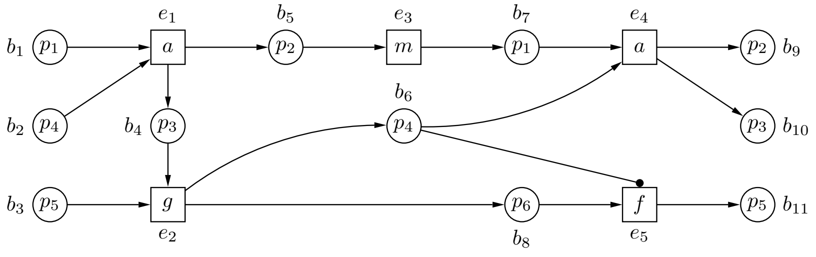


Fig. 6.13 An activator occurrence net AON_0.

Processes of ENI-systems are similar to those of EN-systems with the inhibitor arcs of the system represented by activator arcs which rather than testing for the absence of tokens are used to test for the presence of tokens in complement places. Hence, we assume that each place p of *ENI* adjacent to an inhibitor arc has a complement place $\widetilde{p}$ in the underlying EN-system.

Definition 6.86 (Processes of ENI-systems (α)). A *process* of *ENI* is an ao-net $AON = (P', T', F', \ell, Act)$ such that the underlying occurrence net of the latter is a process of the underlying EN-system of the former and, in addition, ℓ is injective on ${}^{\blacklozenge}e$ and $\ell({}^{\blacklozenge}e) = \widetilde{{}^{\circ}\ell(e)}$ for every transition e in T'. We denote this by $AON \in proc(ENI)$.

The generation of processes for a given step sequence is also based on the one given earlier for EN-systems, showing once again that the addition of inhibitor arcs leads to conservative extensions of notions and results presented earlier on.

Definition 6.87 (Processes construction (π_{ENI})). The activator occurrence net $proc_{ENI}(\sigma)$ *generated* by a step sequence $\sigma = U_1 \dots U_n$ of *ENI* is the last element in the sequence $AON_0, \dots, AON_n$ where each $AON_k \overset{\mathrm{df}}{=} (P_k, T_k, F_k, \ell_k, A_k)$ is an ao-net with the components constructed as in Defn. 6.76, and the following additions:

Step 0: $A_0 = \varnothing$.

Step k: $A_k = A_{k-1} \cup \{(\widetilde{p}^{\,\triangle \widetilde{p}}, a^{1+\triangle a}) \mid a \in U \wedge p \in {}^\circ a\}$.

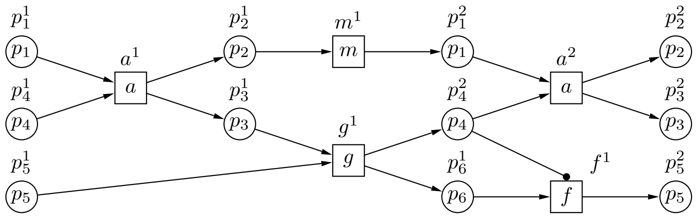


Fig. 6.14 Process generated for ENI_0 and its step sequence $\sigma \overset{\mathrm{df}}{=} a\{g, m\}\{a, f\}$.

We will now show the semantical properties formulated above can be established for ENI-systems and their firing sequences. Referring to the notation used in Figure 6.9, we have the following, where *ENI* is an ENI-system, AON an ao-net, (σ, ℓ) a labelled step sequence, *sos* an so-structure, and $\mathcal{SPO}$ a set of stratified posets with the same domain:

$\mathcal{PN}$	are	ENI-system s	Defn. 6.84
$\mathcal{EX}$	are	step sequences	
$\mathcal{LAN}$	are	ao-nets	Defn. 6.85
$\mathcal{LEX}$	are	labelled step sequences	
$\mathcal{LCS}$	are	labelled so-structures	
$\omega(ENI)$	is	$steps(ENI)$	
$\alpha(ENI)$	is	$proc(ENI)$	Defn. 6.86
$\lambda(AON)$	is	$\lambda_{step}(AON)$	
$\pi_{ENI}(\sigma)$	is	$proc_{ENI}(\sigma)$	Defn. 6.87
$\phi(\sigma, \ell)$	is	$\ell(\sigma)$	
$\kappa(AON)$	is	$sos(AON)$	
$\epsilon(sos)$	is	$spo(sos)$	
$\imath(\mathcal{SPO})$	is	$\bigcap \mathcal{SPO}$.	

We first show that Property I holds (as before, we omit some obvious facts).

Theorem 6.88. *Let ENI be an ENI-system and σ its step sequence, AON be an ao-net, sos be an so-structure, and $\mathcal{SPO}$ be a set stratified posets with the same domain.*

(1) $steps(ENI)$, $proc(ENI)$, $\lambda_{step}(AON)$ and $spo(sos)$ are non-empty sets.
(2) $sos(AON)$ and $\bigcap\mathcal{SPO}$ are so-structures.
(3) $proc_{ENI}(\sigma)$ is an ao-net.

Proof. (1) $steps(ENI) \neq \varnothing$ as the empty string is a valid step sequence of ENI. To show $proc(ENI) \neq \varnothing$ one can take the ao-net consisting of the initial marking of ENI with the identity labelling and no transitions. That $spo(sos) \neq \varnothing$ follows from Thm. 6.22.

Showing that $\lambda_{step}(AON) \neq \varnothing$ proceeds by induction on the number of transitions in AON. In the base case, there are no transitions at all, and the empty labelled step sequence belongs to $\lambda_{step}(AON)$. In the inductive case, one takes the set U of all transitions in AON such that ${}^\bullet U \subseteq M_{init}^{AON}$ and, for every $p \in {}^\bullet U$, if $(p,t) \in Act$ then $t \in U$. One can see that $U \neq \varnothing$ since the relational structure ρ in Defn. 6.85 is $\Diamond$-acyclic and AON is finite. Such a step U can be executed leading to a new marking M. We then delete U and all the inputs of its transitions obtaining an ao-net AON' with the default initial marking M to which the induction hypothesis can be applied. The desired labelled step sequence of AON can be obtained by pre-pending any labelled step sequence of AON' with U.

(2) That $sos(AON)$ is an so-structure follows from the $\Diamond$-acyclicity of the relational structure ρ in Defn. 6.85, and that $\bigcap\mathcal{SPO}$ is an so-structure follows from Thm. 6.22.

(3) The $\Diamond$-acyclicity of the relational structure ρ follows directly from the construction of Defn. 6.87. □

We next show that Property II holds.

Theorem 6.89. *Let ENI be an ENI-system, and σ be its step sequence. Moreover, let AON be a process of ENI, and $\sigma' \in \phi(\lambda_{step}(AON))$.*

(1) $proc_{ENI}(\sigma)$ is a process of ENI.
(2) $\sigma' \in steps(ENI)$ and $AON = proc_{ENI}(\sigma')$.

Proof. (1) Follows directly from the construction in Defn. 6.87.

(2) Let $\sigma' = \phi((U_1 \dots U_n, \ell))$. Then there are markings $M_0, M_1, \dots, M_{n-1}$ of AON such that $M_{init}^{AON}[U_1\rangle M_1 \dots M_{n-1}[U_n\rangle M_{fin}^{AON}$. One can then see

that, by Defn. 6.75, $\ell(M_{init}^{AON})$ is the initial marking of *ENI* and

$$\ell(M_{init}^{AON})[\ell(U_1)\rangle\ell(M_1)\ldots\ell(M_{n-1})[\ell(U_n)\rangle\ell(M_{fin}^{AON})$$

in *ENI*. Hence $\sigma' = \ell(U_1)\ldots\ell(U_n) \in \mathit{steps}(\mathit{ENI})$. We still need to show that AON is isomorphic to $\mathit{proc}_{\mathit{ENI}}(\sigma')$. This can be shown by taking a mapping ψ mapping each transition z^i in $\mathit{proc}_{\mathit{ENI}}(\sigma')$ to the i-th transition in $U_1\ldots U_n$ labelled with z, and then extending it bijectively to the inputs of z^i and $\psi(z^i)$, and the output places of z^i and $\psi(z^i)$. One can show that ψ is an isomorphism for AON and $\mathit{proc}_{\mathit{ENI}}(\sigma')$. □

We finally observe that Property III is nothing but Szpilrajn's theorem for stratified order structures and stratified posets in this case (i.e., Thm. 6.22), and Property IV is proved below.

Theorem 6.90. *If AON is an ao-net then $\lambda_{step}(AON) = \mathit{spo}(\mathit{sos}(AON))$.*

Proof. Referring to Defn. 6.85, if $a \prec_{loc} b$ or $a \sqsubset_{loc} b$ then, by the fact each place has at most one input and at most one output, a must appear before b and a must appear before or in the same step as b, respectively, in any labelled step sequence of AON. Hence the $(\subseteq)$ inclusion holds. To show the reverse one, let σ be a step sequence defined by a stratified poset in $\mathit{spo}(\mathit{sos}(AON))$. Then one can apply similar induction as in the second part of the proof of Thm. 6.88, after observing that the first step U in any stratified poset in $\mathit{spo}(\mathit{sos}(AON))$ is such that ${}^\bullet U \subseteq M_{init}^{AON}$ and, for every $p \in {}^\bullet U$, if $(p,t) \in \mathit{Act}$ then $t \in U$. Consequently, it can be executed. □

We can therefore claim the semantical aims for EN-systems and their firing sequences.

Theorem 6.91. *Let ENI be an ENI-system, and AON be an ao-net.*

$$\begin{array}{rcl} \mathit{proc}(\mathit{ENI}) & = & \mathit{proc}_{\mathit{ENI}}(\mathit{steps}(\mathit{ENI})) \\ \mathit{steps}(\mathit{ENI}) & = & \phi(\lambda_{step}(\mathit{proc}(\mathit{ENI}))) \\ \mathit{sos}(AON) & = & \bigcap \lambda_{step}(AON)\ . \end{array}$$

ENI-systems and Comtraces. The notions of independence and causality developed for EN-systems can be lifted to the level of ENI-systems. The comtrace alphabet of an ENI-system *ENI* is given by

$$\Theta_{\mathit{ENI}} \overset{\mathrm{df}}{=} (T, \mathit{sim}_{\mathit{ENI}}, \mathit{ser}_{\mathit{ENI}})$$

where the simultaneity and serialisability relations are given respectively by:

- $(e,f) \in sim_{ENI}$ if ${}^\bullet e^\bullet \cap {}^\bullet f^\bullet = {}^\circ f \cap {}^\bullet e = {}^\circ e \cap {}^\bullet f = \varnothing$.
- $(e,f) \in ser_{ENI}$ if $(e,f) \in sim_{ENI}$ and $e^\bullet \cap {}^\circ f = \varnothing$.

The set of possible steps of *ENI* is given by:

$$\mathbb{S}_{ENI} \overset{\mathrm{df}}{=} \left\{ U \subseteq T \mid \forall a \neq b \in U : \ (a,b) \in sim_{ENI} \right\} .$$

Again, it is worth noting that both causality relations are structurally defined.

We can now reach conclusions similar to those obtained for EN-systems and Mazurkiewicz traces. This time, however, the equivalence is generated by equations of the form ($AB = A \cup B$), for all $A, B \in \mathbb{S}_{ENI}$ satisfying $A \times B \subseteq ser_{ENI}$.

Since splitting and combining steps of transitions according to the simultaneity and serialisability relations defined by the net have no effect on the process construction, we obtain that $AON_\sigma = AON_\tau$ *iff* σ and τ are equivalent step sequences. Hence with each comtrace one process (up to isomorphism) is associated. Conversely the step language of a process of *ENI* is identical to its defining comtrace.

The remaining details are basically the same as in the case of EN-systems and Mazurkiewicz traces, leading up to the following set of results.

Theorem 6.92. *Let ENI be an ENI-system and σ be its step sequence.*

$$\begin{aligned} steps(ENI) &= \biguplus_{\theta \in steps(ENI)} [\![\theta]\!] \\ cansos([\![\sigma]\!]) &= sos(proc_{ENI}(\sigma)) \\ [\![\sigma]\!] &= \lambda_{step}(proc_{ENI}(\sigma)) \ . \end{aligned}$$

Comtraces and processes give the same views on the causalities in the behaviours of ENI-systems, again providing a justification for the fundamental soundness of the concurrency semantics they both capture. Also, the step sequences of an ENI-system can be partitioned into comtraces. Accordingly, we may state that the causal behaviour of ENI-systems can be captured by the so-structures corresponding to the comtraces partitioning their sets of step sequences.

Mutually Exclusive Transitions. In the last part of this chapter we introduce a new class of Petri nets based on EN-systems which allow the designer not only to use inhibitor arcs, but also mutex arcs which effectively prohibit transitions from occurring simultaneously (i.e., in the same step). This is illustrated in Figure 6.15 which portrays a variant of the producer/consumer scheme. In this case, the producer is allowed to retire

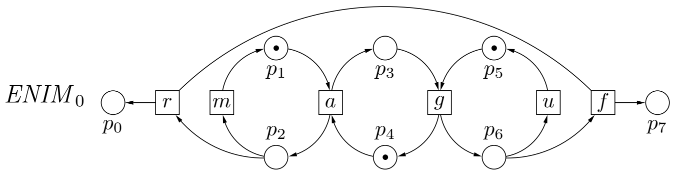


Fig. 6.15 An ENIM-system modelling a third version of the producer/consumer system.

(transition r), but never at the same time as the consumer finishes the job (transition f). Other than that, there are no restrictions on the executions of transitions r and f. The formal device we employ to model such a scenario is a mutex arc between transitions r and f (depicted as an edge without any arrowheads).

Definition 6.93 (ENIM-systems ($\mathcal{PN}$)). An *elementary net system with inhibitor and mutex arcs* (or ENIM-system) is a relational tuple $ENIM \overset{\text{df}}{=} (P, T, F, M_{init}, Inh, Mtx)$ such that the first five components form an underlying ENI-system and $Mtx \subseteq T \times T$ is a symmetric relation specifying *mutex* arcs.

Note that mutex arcs are relating transitions in a direct way, rather than via adjacent places. This should not be regarded as an unusual feature as, for example, Petri nets with priorities also impose certain relationships between transitions in a similarly direct way.

Wherever it is possible we retain the definitions introduced for the underlying ENI-system. The only new notation is ${}^{\diamond}a$ denoting the set of all the transitions b which are excluded from being executed simultaneously with transition a, i.e., $(a, b) \in Mtx$.

One of the consequences of introducing mutex arcs is that the class of potential valid steps needs to be restricted in order to reflect the new kind of constraint. We therefore define:

$$\mathbb{S}_{ENIM} \overset{\text{df}}{=} \left\{ U \subseteq T \mid \forall a \neq b \in U : \ {}^{\bullet}a^{\bullet} \cap {}^{\bullet}b^{\bullet} = \varnothing \ \wedge \ a \notin {}^{\diamond}b \right\} .$$

The remaining definitions pertaining to the dynamic aspects of an ENIM-system are exactly the same as for the underlying ENI-system (but, clearly, under a modified notion of allowed steps of transitions).

For the ENIM-system of Figure 6.15, we have that $M[rf\rangle M'$ and $C[fr\rangle M'$, where $M = \{p_2, p_4, p_6\}$ and $M' = \{p_0, p_3, p_7\}$. However, $M[\{r, f\}\}\rangle M'$ does not hold as r and f cannot be executed in the same step.

We now can move on to the execution semantics of ENIM-systems in the context of the semantical framework, basically following the already established pattern with one crucial modification, namely, the labelled causal structures employed by the semantical are the generalised stratified order structures. First, we extend the notion of ao-nets to include mutex arcs.

Definition 6.94 (Activator mutex occurrence nets ($\mathcal{LAN}$)). An *activator mutex occurrence net* (or amo-net) is a relational tuple

$$AMON \overset{\mathrm{df}}{=} (P', T', F', \ell, Act, Mtx)$$

such that the first five components form an underlying activator occurrence net, and $Mtx \subseteq T' \times T'$ is a symmetric relation specifying mutex arcs such that $Mtx \cap \sqsubset_{loc}^{*} \cap (\sqsubset_{loc}^{*})^{-1} = \varnothing$, where the relation $\sqsubset_{loc}$ is as in Defn. 6.85. The gso-structure associated with $AMON$ is $gsos(AON) \overset{\mathrm{df}}{=} (E, Mtx, \sqsubset)$, where $\sqsubset$ is defined in the same way as for the underlying ao-net.

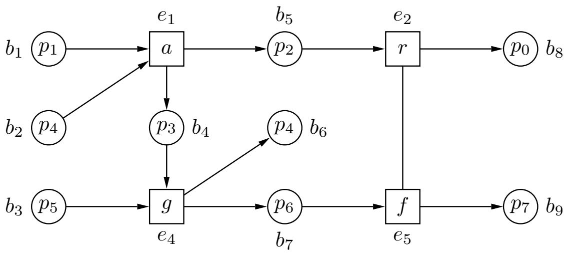


Fig. 6.16 An activator mutex occurrence net $AMON_0$.

As in the previous two cases, an amo-net represents a run of a system. The definitions of initial and final marking, as well step sequence execution are exactly the same as for the underlying ao-net under the proviso that steps do not contain transitions joined by mutex arcs.

Figure 6.16 depicts an amo-net labelled with places and transitions of the ENIM-system of Figure 6.15. For the amo-net $AMON_0$ of Figure 6.16, we have that both $agrf$ and $agfr$ belong to $\phi(\lambda_{step}(AMON_0))$, however, $ag\{f, r\}$ does not.

Similarly, the definition of processes of ENIM-systems closely follows that developed for ENI-systems.

Definition 6.95 (Processes of ENIM-systems (α)). A *process* of *ENIM* is an amo-net $AMON = (P', T', F', \ell, Act, Mtx)$ such that the underlying ao-net of the latter is a process of the underlying ENI-system of the former and, in addition, $(e, f) \in Mtx$ *iff* $e \in {}^{\diamond}\ell(f)$, for all transitions e, f in T'. We denote this by $AMON \in proc(ENIM)$.

Finally, the construction of process is a conservative extensions of that introduced for ENI-systems.

Definition 6.96 (Processes construction (π_{ENIM})). The amo-net $proc_{ENIM}(\sigma)$ *generated* by a step sequence σ of *ENIM* is the last element in the sequence $AMON_0, \ldots, AMON_n$ where each $AMON_k \overset{\mathrm{df}}{=} (P_k, T_k, F_k, \ell_k, A_k, M_k)$ is an amo-net such that

$$M_k \overset{\mathrm{df}}{=} \{(e, f) \in T_k \times T_k \mid (\ell_k(e), \ell_k(f)) \in Mtx\}$$

and all the remaining components are constructed as in Defn. 6.87.

The aom-net shown in Figure 6.16 is a process of the ENIM-system of Figure 6.15.

Figure 6.17 shows the result of the above construction for the ENIM-system of Figure 6.15 and one of its step sequences. Note that this amo-net is isomorphic to that shown in Figure 6.16.

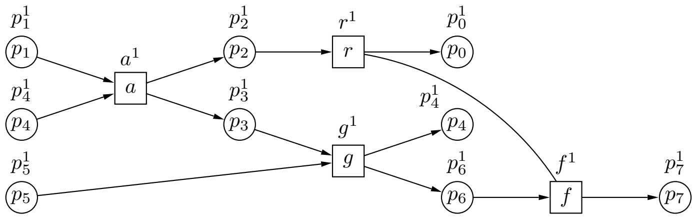


Fig. 6.17 Process generated for $ENIM_0$ and its step sequence $\sigma \overset{\mathrm{df}}{=} a\{g, r\}f$.

It can be shown that all the properties required of a semantical framework are also satisfied for ENIM-systems and their step sequences. Moreover, the treatment of g-comtraces follows the same pattern as in the case of EN-systems and traces, and ENI-systems and comtraces. We leave the formalisation and proofs of these results as an exercise for the reader,

6.9 Concluding Remarks

This chapter covers only part of the much wider field of applying language theory to the study of concurrent behaviours, and so there are several strands of related research which have not even been mentioned. For example, it is possible to develop traces for infinite system behaviours [9, 10], which also allows one to treat aspects such as fairness [30]. Many of those results could also be extended for comtraces and generalised comtraces. Moreover, we have not considered the modelling of conflicts between enabled actions while traces and processes represent single runs in which all the conflicts have already been resolved. Adding conflict amounts to the introduction of branching in processes and considering the prefix ordering of all traces which form the system behaviours. (Branching processes of Petri nets [5] are the basis for an efficient verification technique [34, 6, 27].) If, in addition, one only considers relations between events (transition occurrences) the result is the more abstract model of event structures [13, 47, 36] which have been used to study fundamental concepts of concurrency in a model-independent way. Finally, we only briefly touched upon the algebraic properties of trace concepts such as can be found in, e.g., [4, 3]. Major theoretical problems as recognisability or acceptability have solutions for traces [4] but do not have any for comtraces nor generalised comtraces. Finally, more general net classes such as PT-nets do not admit a purely structural notion of independence between net transitions, and so one needs to resort to notions such as state-related *local traces* [12] to obtain treatment similar to that of Mazurkiewicz traces. Extending local traces to PT-nets with inhibitor and mutex arcs is therefore a challenging task.

The observational semantics used in this chapter is assumed to be defined either in terms of sequences (i.e., total orders) or step-sequences (i.e., stratified orders). It was argued in [46] (and later confirmed in more formal terms in [16]) that any execution that can be observed by a single observer must be an interval order, which means that the most precise observational semantics is in fact in terms of interval orders. When the observational semantics is defined in terms of interval orders, the stratified order structures must be replaced by more complex *interval order structures* (introduced independently in [15] and [31]). Unfortunately not much is known so far about algebraic properties of interval order structures (cf. [23]) and no concepts corresponding to traces or comtraces have yet been proposed. Modelling concurrent histories with relational structures leads to a universal model covering ‘earlier than’, ‘not later than’ and ‘no simultaneity’, together with various assumptions about the structure of system runs (c.f. [14]). In case one is interested in modelling only 'not later than' for system runs modelled by step sequences, one can use the approach proposed by [44, 48] based on preorders.

Acknowledgments

We would like to thank Carlos Martín-Vide for his support and encouragement during the work on this chapter. Furthermore, we are grateful to the anonymous reviewers for their comments and constructive suggestions.

References

1. Burris, S. and Sankappanavar, H. P. (1981). *A Course in Universal Algebra* (Springer).
2. Cartier, P. and Foata, D. (1969). *Problèmes combinatoires de commutation et réarrangements*, *Lecture Notes in Mathematics*, Vol. 85 (Springer).
3. Diekert, V. and Métivier, Y. (1997). Partial commutation and traces, in G. Rozenberg and A. Salomaa (eds.), *Handbook of Formal Languages*, Vol. 3 (Springer), pp. 457–533.
4. Diekert, V. and Rozenberg, G. (eds.) (1995). *The Book of Traces* (World Scientific).
5. Engelfriet, J. (1991). Branching processes of Petri nets, *Acta Informatica* **28**, pp. 575–591.
6. Esparza, J., Römer, S. and Vogler, W. (1996). An improvement of McMillan's unfolding algorithm, in *TACAS*, *Lecture Notes in Computer Science*, Vol. 1055 (Springer), pp. 87–106.
7. Fishburn, P. C. (1985). *Interval Orders and Interval Graphs* (John Wiley).
8. Gaifman, H. and Pratt, V. R. (1987). Partial order models of concurrency and the computation of functions, in *LICS* (IEEE Computer Society), pp. 72–85.
9. Gastin, P. (1990). Infinite traces, in *Spring School of Theoretical Computer Science on Semantics of Systems of Concurrent Processes*, *Lecture Notes in Computer Science*, Vol. 469 (Springer), pp. 277–308.
10. Gastin, P. and Petit, A. (1992). Poset properties of complex traces, in *MFCS*, *Lecture Notes in Computer Science*, Vol. 629 (Springer), pp. 255–263.
11. Guo, G. and Janicki, R. (2002). Modelling concurrent behaviours by commutativity and weak causality relations, in *AMAST*, *Lecture Notes in Computer Science*, Vol. 2422 (Springer), pp. 178–191.
12. Hoogers, P., Kleijn, H. and Thiagarajan, P. (1995). A trace semantics for Petri nets, *Information and Computation* **117**, pp. 98–114.

13. Hoogers, P. W., Kleijn, H. C. M. and Thiagarajan, P. S. (1996). An event structure semantics for general Petri nets, *Theoretical Computer Science* **153**, pp. 129–170.
14. Janicki, R. (2008). Relational structures model of concurrency, *Acta Informatica* **45**, pp. 279–320.
15. Janicki, R. and Koutny, M. (1991). Invariants and paradigms of concurrency theory, in *PARLE*, *Lecture Notes in Computer Science*, Vol. 506 (Springer), pp. 59–74.
16. Janicki, R. and Koutny, M. (1993). Structure of concurrency, *Theoretical Computer Science* **112**, pp. 5–52.
17. Janicki, R. and Koutny, M. (1995). Semantics of inhibitor nets, *Information and Computation* **123**, 1, pp. 1–16.
18. Janicki, R. and Koutny, M. (1997). Fundamentals of modelling concurrency using discrete relational structures, *Acta Informatica* **34**, pp. 367–388.
19. Janicki, R. and Lauer, P. E. (1992). *Specification and Analysis of Concurrent Systems: The COSY Approach* (Springer).
20. Janicki, R., Lauer, P. E., Koutny, M. and Devillers, R. (1986). Concurrent and maximally concurrent evolution of nonsequential systems, *Theoretical Computer Science* **43**, pp. 213–238.
21. Janicki, R. and Lê, D. T. M. (2008). Modelling concurrency with quotient monoids, in *Petri Nets*, *Lecture Notes in Computer Science*, Vol. 5062 (Springer), pp. 251–269.
22. Janicki, R. and Lê, D. T. M. (2010). Modelling concurrency with comtraces and generalized comtraces, *Information and Computation (to appear)* .
23. Janicki, R., Lê, D. T. M. and Zubkova, N. (2009). Closure operators for order structures, in *FCT*, *Lecture Notes in Computer Science*, Vol. 5699 (Springer), pp. 217–229.
24. Juhás, G., Lorenz, R. and Mauser, S. (2006). Synchronous + concurrent + sequential = earlier than + not later than, in *ACSD* (IEEE Computer Society), pp. 261–272.
25. Juhás, G., Lorenz, R. and Mauser, S. (2008). Complete process semantics of Petri nets, *Fundamenta Informaticae* **87**, pp. 331–365.
26. Keesmaat, N. and Kleijn, H. (1997). Net-based control versus rational control: the relation between itnc vector languages and rational relations, *Acta Informatica* **34**, pp. 23–57.
27. Khomenko, V., Koutny, M. and Vogler, W. (2003). Canonical prefixes of Petri net unfoldings, *Acta Informatica* **40**, pp. 95–118.
28. Kleijn, H. C. M. and Koutny, M. (2004). Process semantics of general inhibitor nets, *Information and Computation* **190**, pp. 18–69.
29. Kleijn, J. and Koutny, M. (2008). Formal languages and concurrent behaviours, in G. Bel-Enguix, M. Jiménez-López and C. Martín-Vide (eds.), *New Developments in Formal Languages and Applications*, *Studies in Computational Intelligence*, Vol. 113 (Springer), pp. 125–182.
30. Kwiatkowska, M. Z. (1989). *Fairness for non-interleaving concurrency*, PhD Thesis, University of Leicester.
31. Lamport, L. (1986). The mutual exclusion problem: Part I - a theory of

interprocess communication; Part II - statements and solutions, *Journal of ACM* **33**, pp. 313–326.

32. Mazurkiewicz, A. (1977). Concurrent program schemes and their interpretations, Technical Report DAIMI-78, University of Aarhus.
33. Mazurkiewicz, A. (1995). Introduction to trace theory, in [4], pp. 3–41.
34. McMillan, K. L. (1993). Using unfoldings to avoid the state explosion problem in the verification of asynchronous circuits, in *CAV*, *Lecture Notes in Computer Science*, Vol. 663 (Springer), pp. 164–177.
35. Milner, R. (1980). *A Calculus of Communicating Systems*, *Lecture Notes in Computer Science*, Vol. 92 (Springer).
36. Nielsen, M., Plotkin, G. D. and Winskel, G. (1981). Petri nets, event structures and domains, part I, *Theoretical Computer Science* **13**, pp. 85–108.
37. Ochmański, E. (1995). Recognizable trace languages, in [4], pp. 167–204.
38. Petri, C. A. (1962). Fundamentals of a theory of asynchronous information flow, in *IFIP Congress* (North Holland), pp. 386–390.
39. Petri, C. A. (1973). Concepts of net theory, in *MFCS* (Slovak Academy of Sciences), pp. 137–146.
40. Reisig, W. and Rozenberg, G. (eds.) (1998a). *Lectures on Petri Nets I: Basic Models*, *Lecture Notes in Computer Science*, Vol. 1491 (Springer).
41. Reisig, W. and Rozenberg, G. (eds.) (1998b). *Lectures on Petri Nets II: Applications*, *Lecture Notes in Computer Science*, Vol. 1492 (Springer).
42. Rozenberg, G. and Engelfriet, J. (1998). Elementary net systems, in [40], pp. 12–121.
43. Shields, M. W. (1979). Adequate path expressions, in *International Symposium on Semantics of Concurrent Computation*, *Lecture Notes in Computer Science*, Vol. 70 (Springer), pp. 249–265.
44. Shields, M. W. (1989). Behavioural presentations, in *REX Workshop*, *Lecture Notes in Computer Science*, Vol. 354 (Springer), pp. 673–689.
45. Szpilrajn, E. (1930). Sur l'extension de l'ordre partiel, *Fundamenta Mathematicae* **16**, pp. 386–389.
46. Wiener, N. (1914). A contribution to the theory of relative position, *Proc. Camb. Philos. Soc* **17**, pp. 441–449.
47. Winskel, G. (1988). An introduction to event structures, in *Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency*, *Lecture Notes in Computer Science*, Vol. 354 (Springer), pp. 364–397.
48. Winskel, G. (1989). Event structure semantics for CCS and related languages, Technical Report DAIMI-159, University of Aarhus.

This page is intentionally left blank

Chapter 7

Correction Queries in Active Learning

Cristina Tîrnăucă
Departamento de Matemáticas, Estadística y Computación,
Universidad de Cantabria, Spain,
E-mail: `cristina. tirnauca@unican. es`

In this work we investigate the learning model in which a learner gets information of the hidden concept by using different types of correction queries (CQs). Specifically, three types of queries are taken into account, the so-called prefix, length bounded and edit distance CQs. In order to state the power of these models, we have considered two learning scenarios. In the first one, computational complexity issues are neglected and the matter is to decide, for a given class of concepts, whether learning in a finite number of steps is possible. We show power relationships among CQ models themselves, between these models and well-established query models, and with learning models in a different paradigm, namely the Gold-style learning in the limit model.

The last part of this chapter is focused on the second learning scenario where computational complexity issues are important. We provide new polynomial time learning algorithms using prefix correction queries for the class of pattern languages and the class of k-reversible languages. In addition, we show power relationships among the new models themselves and the membership query model.

7.1 Introduction

In the field of grammatical inference many of the existing learning models have as inspiration the process of human language acquisition. The introduction of the field itself is motivated by the hope to gain a better understanding of what learning really is, as Gold points out in his pioneering paper [11].

Another example is Valiant's PAC learning model [34], in which the learner has to output, with high probability, a language that is close enough to the target one (see [16] for further details). This formalism captures our intuition that, although many of us might never master to perfection a language, we should be able to get a good enough approximation eventually.

Last but not least, Angluin's query learning model [4] reflects another feature, namely children's ability to ask questions within the process of acquiring their native language. None of the traditional types of queries reflects though one important aspect of this process: although children do not receive explicit negative information (words that are not in the language or ungrammatical sentences), they are corrected when they make mistakes. And this is why a new type of queries, called CORRECTION QUERIES (CQs), was introduced (see [9]).

The first formal definition of CQs appears in [7]. The algorithm given there, a straightforward modification of Angluin's L^* [4], allows the learner to identify any minimal complete DFA from CQs and equivalence queries in polynomial time. In order to distinguish these queries from all the other types subsequently introduced, we will refer to them as *prefix correction queries* (PCQs) throughout this work. A learner, in response to a PCQ, gets the smallest (in the lex-length order) extension of the queried datum such that their concatenation belongs to the target language.

Note that in the case of DFAs, returning the answer to a PCQ can be easily done in polynomial time. However, when the target concept ranges over arbitrary recursive languages, the answer to a PCQ might be very long or not even computable (given a recursive language L and a string w, one cannot decide, in general, if w is a prefix of a string in L). A possible solution to avoid very long (or infinite) searches is to restrict the search space to only short enough suffixes. Therefore, we introduce the notion of *length bounded correction query* (LBCQ). Given a fixed number l, answering an l-bounded correction query consists in returning all strings having the length smaller than or equal to l that, concatenated with the queried datum, form a string in the target language [31, 32].

The third main type of CQs is based on the edit distance, and has been independently introduced by E. Kinber in [17] and by L. Becerra Bonache, C. de la Higuera, J.C. Janodet and F. Tantini in [8]. According to [8], the *edit distance correction* of a given string is, by definition, one string in the target language at minimum distance from the queried datum. If many such strings exist, the teacher randomly returns one of them. A slightly modified type of edit distance correction query (EDCQ) is introduced by Kinber in [17]. The author there imposes a supplementary condition: the teacher has to return only strings that have not previously appeared during the run of the algorithm. Clearly, this restriction makes sense only with respect to an algorithm, so we will consider this type of EDCQ only in the section dedicated to polynomial time learning algorithms.

In this work we present a series of results obtained concerning correction queries. After a preliminary section, we describe two well-known learning settings: Angluin's query learning model and Gold's learning in the limit model, among with relationships that exists between them. Next, for each of the three types of CQs mentioned above, we give necessary and sufficient conditions for a class of languages to be learnable with the given type of CQ. Moreover, we show that LBCQs and EDCQs have exactly the same learning power as MQs, whereas PCQs are strictly more powerful. Furthermore, a comparison with Gold-style learning models is provided, showing, for example, that learning with PCQs is a strictly weaker model than learning in the limit from text. The last section is dedicated to polynomial time learners. We give algorithms for learning pattern languages and k-reversible languages with PCQs, and we compare all three types of CQs with the traditional MQs.

7.2 Preliminaries

We follow standard definitions and notations in formal language theory. The reader is referred to [13, 14, 26] for further information about this domain.

Let Σ be a finite set of symbols called the alphabet and let Σ^* be the set of strings over Σ. A *language* L over Σ is a subset of Σ^*. The elements of L are called *strings* or *words*. Let u, v, w be strings in Σ^* and $|w|$ be the length of w. λ is a special word called the *empty* string and has length 0. We denote by uv or $u \cdot v$ the concatenation of the strings u and v. If $w = uv$ for some u, v in Σ^*, then u is a *prefix* of w and v is a *suffix* of w. A set L is called *prefix-closed* (*suffix-closed*) if for any w in L, all prefixes (suffixes) of w are also in L.

By $\mathit{Pref}(L)$ we denote the set $\{u \in \Sigma^* \mid \exists v \in \Sigma^* \text{ such that } uv \in L\}$ and by $\mathit{Tail}_L(u)$ the set $\{v \mid uv \in L\}$. Then, by $\Sigma^{\leq k}$ and Σ^k we denote the sets

$\{u \in \Sigma^* \mid |u| \leq k\}$ and $\{u \in \Sigma^* \mid |u| = k\}$, respectively.

Assume that Σ is a totally ordered set and let $\prec_l$ be the lexicographical order on Σ^*. Then, the *lex-length order* $\prec$ on Σ^* is defined by: $u \prec v$ if either $|u| < |v|$, or else $|u| = |v|$ and $u \prec_l v$. In other words, strings are compared first according to length and then lexicographically.

The edit distance between the strings w and w', denoted $d(w, w')$ in the sequel, is the minimum number of *edit operations* needed to transform w into w'. The edit operations are either (1) *deletion*: $w = uav$ and $w' = uv$, or (2) *insertion*: $w = uv$ and $w' = uav$, or (3) *substitution*: $w = uav$ and $w' = ubv$, where $u, v \in \Sigma^*, a, b \in \Sigma$ and $a \neq b$. The edit distance between two strings w and w' can be computed in $\mathcal{O}(|w| \cdot |w'|)$ time by dynamic programming [35].

Given a language L, one can define the following *equivalence relation* (i.e., a relation which is reflexive, symmetric and transitive) on strings: $u \equiv_L v$ if and only if $(\forall w \in \Sigma^*, u \cdot w \in L \Leftrightarrow v \cdot w \in L)$. This equivalence relation divides the set of all strings into one or more *equivalence classes.* For any language L over Σ and any w in Σ^* we denote by $[w]_L$ the equivalence class of w with respect to the language L, or simply $[w]$ when L is understood from the context. The number of equivalence classes induced by $\equiv_L$ is called the *index* of L.

A *deterministic finite automaton* (DFA) is a 5-tuple $\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$ where Q is the (finite) set of states, Σ is a finite alphabet, $q_0 \in Q$ is the initial state, $F \subseteq Q$ is the set of final states and δ is a partial function that maps $Q \times \Sigma$ to Q which can be extended to strings by doing $\delta(q, \lambda) = q$ and $\delta(q, wa) = \delta(\delta(q, w), a)$ whenever the right-hand side is defined. The number of states in Q gives the *size* of $\mathcal{A}$. A string w is accepted by $\mathcal{A}$ if $\delta(q_0, w) \in F$. The set of strings accepted by $\mathcal{A}$ is denoted by $\mathcal{L}(\mathcal{A})$ and is called a *regular language.*

We say that a DFA $\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$ is *complete* if for all q in Q and a in Σ, $\delta(q, a)$ is defined (i.e., δ is a total function). For any regular language L, there exists a minimum state complete DFA hereinafter denoted $\mathcal{A}_L$ such that $\mathcal{L}(\mathcal{A}_L) = L$. The Myhill-Nerode Theorem [28, 29] states that the size of $\mathcal{A}_L$ equals the index of L. An immediate consequence of this theorem is that a language L is regular if and only if L has finite index.

Let us now introduce some particular classes of languages which will be later used in this work.

- Following [8], one can define the language $B_r(w)$ of all strings whose edit distance is at most r from w where $w \in \Sigma^*$ and $r \in \mathbb{R}$. This language will subsequently be called the *ball of center w and radius r*. Formally, $B_r(w) = \{v \in \Sigma^* \mid d(v, w) \leq r\}$. We denote by $\mathcal{B}$ the class of all balls of strings over a fixed alphabet Σ.

- Given a nonnegative integer k, a language L is said to be *k-reversible* if whenever u_1vw, u_2vw are in L and $|v| = k$, $Tail_L(u_1v) = Tail_L(u_2v)$. We denote by *$k$-Rev* the class of all k-reversible languages.
- We assume a finite alphabet Σ such that $|\Sigma| \geq 2$, and a countable, infinite set of *variables* $X = \{x, y, z, x_1, y_1, z_1, \ldots, \}$. A *pattern* π is any nonempty string over $\Sigma \cup X$. The *pattern language* $L(\pi)$ consists of all the strings obtained by uniformly replacing the variables in π with arbitrary strings in Σ^+. Let us denote by $\mathbb{P}$ the set of all pattern languages over a fixed alphabet Σ. The pattern π is in *normal form* if the variables occurring in π are precisely $x_1, \ldots, x_k$, and for every i with $1 \leq i < k$, the leftmost occurrence of x_i in π is left to the leftmost occurrence of x_{i+1}.

7.3 Learning Models

Let $\mathcal{C}$ be a class of nonempty recursive languages over Σ^*. Then $\mathcal{C}$ is an *indexable class* (or *indexed family*) if there is an effective enumeration $(L_i)_{i\geq 1}$ of all and only the languages in $\mathcal{C}$ such that membership is uniformly decidable, i.e., there exists a computable function that, for any $w \in \Sigma^*$ and $i \geq 1$, returns 1 if $w \in L_i$, and 0 otherwise. Such an enumeration will subsequently be called an *indexing* of $\mathcal{C}$. In the sequel we might say that $\mathcal{C} = (L_i)_{i\geq 1}$ is an indexable class and understand that $\mathcal{C}$ is an indexable class and $(L_i)_{i\geq 1}$ is an indexing of $\mathcal{C}$.

7.3.1 *Query Learning*

In the query learning model a learner has access to an oracle that truthfully answers queries of a specified kind. A *query learner* is an algorithmic device that, depending on the reply of the previous queries, either computes a new query, or returns a hypothesis and halts.

More formally, let $\mathcal{C}$ be an indexable class and let L be an arbitrary language in $\mathcal{C}$. The query learner $\mathcal{A}lg$ *learns L using some type of queries* if it eventually halts and its only hypothesis, say i, correctly describes L (i.e., $L_i = L$). So, $\mathcal{A}lg$ returns its unique and correct guess i after only finitely many queries. Moreover, *$\mathcal{A}lg$ learns the class $\mathcal{C}$ using some type of queries* if it learns every language of that class using queries of the specified type.

The most investigated types of queries are:

- *Membership queries* (MQs). The input is a string w, and the answer is `Yes` or `No`, depending on whether or not w belongs to the target language L.
- *Equivalence queries* (EQs). The input is an index j of some language $L_j \in \mathcal{C}$. If $L_j = L$, the answer is `Yes`. Otherwise together with the answer `No`, a counterexample from $(L_j \backslash L) \cup (L \backslash L_j)$ is supplied.

The collections of all indexable classes $\mathcal{C}$ for which there is a query learner $\mathcal{A}lg$ such that $\mathcal{A}lg$ learns $\mathcal{C}$ using membership (or equivalence) queries are denoted by *MemQ* (*EquQ*, respectively).

7.3.2 *Gold-style Learning*

In order to present the Gold-style learning models we need some further notions, briefly explained below (for details, see [11, 2, 40]).

Let L be a nonempty language. A *text for L* is an infinite sequence $\sigma = w_1, w_2, w_3, \ldots$ such that $\{w_i \mid i \geq 1\} = L$. An *informant for L* is an infinite sequence $\sigma = (w_1, b_1), (w_2, b_2), (w_3, b_3), \ldots$ with $b_i \in \{0, 1\}$ for all $i \geq 1$ such that $\{w_i \mid i \geq 1 \text{ and } b_i = 1\} = L$ and $\{w_i \mid i \geq 1 \text{ and } b_i = 0\} = \Sigma^* \backslash L$.

Let $\mathcal{C} = (L_i)_{i \geq 1}$ be an indexable class. An *inductive inference machine* (IIM) is an algorithmic device that reads longer and longer initial segments σ of a text (informant), and outputs numbers as its hypotheses. An IIM returning some i is construed to hypothesize the language L_i. Given a text (an informant) σ for a language $L \in \mathcal{C}$, *$\mathcal{A}lg$ learns L from σ* if the sequence of hypotheses output by $\mathcal{A}lg$, when fed σ, stabilizes on a number i (i.e., past some point $\mathcal{A}lg$ always outputs the hypothesis i) with $L_i = L$. We say that *$\mathcal{A}lg$ learns $\mathcal{C}$ from text (informant)* if it identifies each $L \in \mathcal{C}$ from every corresponding text (informant).

A slightly modified version of the learning in the limit model is the model of *conservative learning* (see [41, 39] for more details). A *conservative* IIM is only allowed to change its mind in case its actual guess contradicts the data seen so far.

As above, *LimTxt* (*LimInf*) denotes the collection of all indexable classes $\mathcal{C}$ for which there is an IIM $\mathcal{A}lg$ such that $\mathcal{A}lg$ identifies $\mathcal{C}$ from text (informant). One can similarly define *ConsvTxt* and *ConsvInf*, for which the inference machines should be conservative IIMs.

Although an IIM is allowed to change its mind finitely many times before returning its final and correct hypothesis, in general it is not decidable whether or not it has already output its final hypothesis. Hence, the learner must go on processing information forever because there is always the possibility that some future information will force him to change his guess. As opposed to that, in the *finite identification model*, the learner is required to know when his answer is correct, that is, he has to stop the presentation of information at some finite time when he thinks it has received enough, and state the identity of the unknown object (see [11]). The corresponding models *FinTxt* and *FinInf* are defined as above.

7.3.3 *A Hierarchy of Learning Models*

There has been quite a lot of work done for comparing the aforementioned learning methods and finding characterizations for the classes of languages inferable within specific settings (see [18, 2, 41, 27, 22]). The resulting hierarchy is presented bellow.

$$\mathit{FinTxt} \subset \mathit{FinInf} = \mathit{MemQ} \subset \mathit{ConsvTxt} \subset \mathit{LimTxt} \subset \mathit{LimInf} = \mathit{EquQ}$$

7.4 Learning with Correction Queries

The way children learn their mother language is an amazing process. They receive examples of sentences in that language, and after some transitory period - in which they still make mistakes and are corrected by adults - they are able to express themselves fluently and errorless. It is clear that a child left alone with all kinds of teaching material would learn to speak slower (if ever) than one integrated in a community. The key difference between the above mentioned cases consists in the possibility for the second child to interact with others, and to be corrected when he or she makes mistakes.

Among the existing learning models, the one that best describes the interaction that takes place within the process of child acquiring his native language is the query learning model [4]. First introduced, and in the same time the most used types of queries, are MQs and EQs.

There are quite a few reasons though for which people working in grammatical inference, and especially in active learning, have been trying to find effective algorithms able to identify regular languages without the use of EQs. First of all, EQs are computationally costly, and many times unavailable, as it is the case of context free languages (unless we assume one has a magic black box that answers any kind of questions). Secondly, they are quite unnatural for a real life setting: no child would ever ask if his or her current hypothesis represents the correct grammar of the language (not to mention that children do not have conscious access to a representation for their native language). Finally, it might happen that the teacher does not even have a grammar of the target language - take, for example, the case of native speakers that have never studied grammar.

On the other hand, membership queries are not informative enough, not being able to capture the feedback received by the child when he or she makes mistakes. Inspired by the way adults guide the process of children's language acquisition by correcting them when necessary, a modified version of MQs, called *correction query*, has been proposed [9]. More precisely, the difference consists in the fact that for strings not belonging to the target language, the teacher must provide the learner with a *correction.* Whereas in the case of natural languages the correction for an ungrammatical sentence would be one in which the adult is replacing the error with a correct (sequence of) word(s), in formal language theory different objects may require different types of corrections. Therefore, several types of CQs have been proposed so far, and the goal of this work is to present the main results concerning them.

7.4.1 *Learning with Prefix Correction Queries*

We begin our study with the type of corrections that were chronologically the first ones introduced, namely the *prefix correction queries* (PCQs).

If L is a formal language over the alphabet Σ and w is any string in Σ^*, the *prefix correcting string* of w with respect to L, denoted simply by $C_L(w)$, is the smallest string in the lex-length order of the set $Tail_L(w)$, if this set is not empty, and the symbol $\Theta \notin \Sigma$ otherwise. Hence, C_L is a function from Σ^* to $\Sigma^* \cup \{\Theta\}$.

We denote by $PCorQ$ the collection of all indexable classes $\mathcal{C}$ for which there is a query learner $\mathcal{A}lg$ such that $\mathcal{A}lg$ learns $\mathcal{C}$ using prefix correction queries. A special attention has to be paid to those classes of languages for which a teacher can be effectively implemented. That is, those indexable classes $\mathcal{C} = (L_i)_{i \geq 1}$ that have the following property $\mathbb{A}$: there exists a recursive function $\phi : \mathbb{N} \times \Sigma^* \to \Sigma^* \cup \{\Theta\}$ such that $\phi(i, w) = v$ if and only if $C_{L_i}(w) = v$ for any $w \in \Sigma^*$ and $L_i \in \mathcal{C}$. In other words, an indexable class $\mathcal{C} = (L_i)_{i \geq 1}$ has property $\mathbb{A}$ if and only if for all indexes i, $\mathit{Pref}(L_i)$ is recursive (computing the correction of a string w with respect to a language L can be easily done once we know whether w is in $\mathit{Pref}(L)$). Note that for an arbitrary recursive language L, the prefix $\mathit{Pref}(L)$ is not necessary recursive.

So, let us denote by $\mathit{PCorQ}^{\mathbb{A}}$ the collection of those classes of languages in PCorQ for which condition $\mathbb{A}$ is satisfied. Clearly, for the language classes in $\mathit{PCorQ}^{\mathbb{A}}$ all answers to the PCQs can be effectively computed. So in this case we could speak about a teacher instead of an oracle.

7.4.1.1 *Necessary and Sufficient Conditions*

In this section we provide the reader with some necessary and sufficient conditions for a class of languages to be learnable with PCQs alone. Although these conditions are similar to the ones given in [27, 20] for the model of learning with MQs, there exists some differences. In [27], Mukouchi gives a characterization for finite learning from informant (and hence for learning with MQs) in terms of what he calls *pairs of definite finite tell-tales*, probably inspired by Angluin's *finite tell-tales* [2]. Basically, he shows that a language class is inferable with MQs if for each language in the class there exists a pair of finite sets - one with positive examples and the other one with negative examples - which makes it unique, and such a pair of sets can be effectively computed.

We will see on an example that this property is no longer enough to characterize the model of learning with PCQs (see [33]: Lemma 4, page 45 for a formal proof). If $K_1, K_2, K_3, \ldots$ is the collection of all finite nonempty sets of positive integers, and $\Sigma = \{a\}$, we define $L_i = \{a^n \mid n \in K_i\}$ for all $i \geq 1$, and $\mathcal{C} = (L_i)_{i \geq 1}$. One may imagine a simple algorithm that learns any language L in $\mathcal{C}$ with PCQs. Indeed, it is enough to ask PCQs for the strings $w_0, w_1, w_2, \ldots, w_n$ until the oracle returns the answer Θ (i.e., $C_L(w_n) = \Theta$), where $w_0 = \lambda$ and $w_{i+1} = w_i \cdot C_L(w_i) \cdot a$ for all $i \in \{0, \ldots, n-1\}$. On the other hand, for all possible pairs of finite sets $\langle T, F \rangle$ such that $T \subseteq L$ and $F \subseteq \Sigma^* \backslash L$, there is always a bigger language L' in $\mathcal{C}$ which includes T and does not contain any element of F. Hence, although the class $\mathcal{C}$ does not admit pairs of definite finite-tell tales, it is still learnable with PCQs.

From this simple example we deduct that in order to learn a class in PCorQ one also needs information about those strings that have the special symbol Θ as correction. Therefore, we introduce the notion of *triples of definite finite tell-tales*.

We say that a language L is *consistent with a triple of sets* $\langle T, F, U\rangle$ if $T \subseteq L$, $F \subseteq \Sigma^*\backslash L$ and $U \subseteq \Sigma^*\backslash \mathit{Pref}(L)$. The triple $\langle T, F, U\rangle$ is a *triple of definite finite tell-tales* of the language L in $\mathcal{C} = (L_i)_{i\geq 1}$ if :

(1) T, F and U are finite,
(2) L is consistent with $\langle T, F, U\rangle$, and
(3) for all $j \geq 1$, if L_j is consistent with $\langle T, F, U\rangle$, then $L_j = L$.

Going back to our example, it can be checked that $\langle T_i, F_i, U_i\rangle$ is a triple of definite finite tell-tales for L_i, where $T_i = L_i$, $l = \max\{n \mid n \in K_i\}$, $F_i = \{a^n \mid n \in \{1, \ldots, l\}\backslash K_i\}$ and $U_i = \{a^{l+1}\}$ (see [33]: Lemma 3, page 44).

Next, we present necessary and sufficient conditions for a class of languages to be learnable with PCQs based on triples of definite finite tell-tales. In what follows we use the notion of convergence in the following way: we say that a series of triples of sets $\langle T_j, F_j, U_j\rangle_{j\geq 1}$ converges, in the limit, to some triple $\langle T_*, F_*, U_*\rangle$ if there exists an $N \geq 1$ such that for all $n \geq N$, $\langle T_n, F_n, U_n\rangle = \langle T_*, F_*, U_*\rangle$.

Proposition 7.1 (Necessary condition). *If the class $\mathcal{C} = (L_i)_{i\geq 1}$ is in PCorQ, then there exists an effective procedure which enumerates, for any input $i \geq 1$, an infinite series of triples $\langle T_j, F_j, U_j\rangle_{j\geq 1}$ that converges in the limit to a triple of definite finite tell-tales of L_i.*

Proof. Let $\mathcal{C} = (L_i)_{i\geq 1}$ be an indexable class in *PCorQ*, and $\mathcal{A}lg$ a query learning algorithm that learns $\mathcal{C}$ using PCQs. Since the class $\mathcal{C}$ does not necessarily have property $\mathbb{A}$, the answers to PCQs might not always be computable. We use this observation to design an effective procedure as described above. Whenever the oracle is queried with the string w, our teacher will return the value $C_{L^{\leq n}}(w)$ where n is a fixed natural number and $L^{\leq n} = \{w \in L \mid |w| \leq n\}$. Of course, $C_{L^{\leq n}}(w)$ and $C_L(w)$ might be different, so $\mathcal{A}lg$ is not sure to converge anymore (and even if it does, it might converge to a language that is different from the target one). That is why we only run it for at most a finite number of steps, avoiding possible loops.

The following procedure outputs an infinite series of triples that converges in the limit to a triple of definite finite tell-tales of L.

We show that the sequence of triples produced by Algorithm 1 converges

Algorithm 1 A series convergent to a triple of definite finite tell-tales

```
1: Input: the target language $L$
2: $n := 0$
3: while TRUE do
4:     $n := n + 1$
5:     run $\mathcal{A}lg$ on $L$ at most $n$ steps, and collect the sequence of queries and
6:     answers from the teacher w.r.t. the language $L^{\leq n}$ in $QA_n$
7:     $T_n := \{wv \mid (w, v) \in QA_n$ and $v \neq \Theta\}$
8:     $F_n := \{wv' \mid (w, v) \in QA_n, v \neq \Theta$ and $v' \prec v\}$
9:     $U_n := \{w \mid (w, \Theta) \in QA_n\}$
10:    output $\langle T_n, F_n, U_n \rangle$
11: end while
```

to a triple of definite finite tell-tales of L.

Let QA_* be the sequence of queries and answers processed by $\mathcal{A}lg$ when learning L (recall that $\mathcal{A}lg$ is a query learning algorithm that learns $\mathcal{C}$ using PCQs) and let m be the cardinality of QA_*. Take $T_* = \{wv \mid (w, v) \in QA_*, v \neq \Theta\}$, $F_* = \{wv' \mid (w, v) \in QA_*, v \neq \Theta$ and $v' \prec v\}$ and $U_* = \{w \mid (w, \Theta) \in QA_*\}$. Clearly, T_*, F_* and U_* are all finite. Moreover, $T_* \subseteq L$, $F_* \subseteq \Sigma^* \backslash L$ and $U_* \subseteq \Sigma^* \backslash \mathit{Pref}(L)$. Let us now take i such that L_i is consistent with $\langle T_*, F_*, U_* \rangle$. It can be shown that for all $(w, v) \in QA_*$, $C_{L_i}(w) = v = C_L(w)$. Since the algorithm $\mathcal{A}lg$ is assumed to identify a unique language from the class $\mathcal{C}$, we obtain $L_i = L$. Hence, $\langle T_*, F_*, U_* \rangle$ is a triple of definite finite tell-tales of L.

If we take $l = \max\{|wv| \mid (w, v) \in QA_*\}$, where the length of Θ is defined as 0, we have that for all $n \geq l$ and all pairs (w, v) in QA_*, $C_L(w) = v = C_{L^{\leq n}}(w)$. So, if $N = \max\{l, m\}$ then for all $n \geq N$, $\langle T_n, F_n, U_n \rangle = \langle T_*, F_*, U_* \rangle$. □

The following corollary is then obvious.

Corollary 7.2. *If the class $\mathcal{C} = (L_i)_{i \geq 1}$ is in PCorQ, then a triple of definite finite tell-tales of L_i does exist for any index i.*

So, we know that for all language classes $\mathcal{C}$ in *PCorQ*, every language in $\mathcal{C}$ has a triple of definite finite tell-tales. Proposition 7.3 shows that having a way of computing such a triple is a sufficient condition for an indexable class of languages to be in *PCorQ* (the reader is referred to [33] for the proof).

Proposition 7.3 (Sufficient condition). *Let $\mathcal{C} = (L_i)_{i\geq 1}$ be an indexable class. If a triple of definite finite tell-tales of L_i is uniformly computable for any i, then $\mathcal{C}$ is in PCorQ.*

It is an open question whether there exists a characterization in terms of finite sets for the whole class *PCorQ*. In the sequel we present such a characterization for those classes in *PCorQ* that have property $\mathbb{A}$.

Theorem 7.4. *Let $\mathcal{C} = (L_i)_{1\geq 1}$ be an indexable class with property $\mathbb{A}$. Then $\mathcal{C}$ belongs to PCorQ if and only if a triple of definite finite tell-tales of L_i is uniformly computable for any index i.*

A question that naturally arises is whether *PCorQ* and $PCorQ^{\mathbb{A}}$ are equal. The following class shows us that this is not the case.

Consider a TM M over an input alphabet Σ, an alphabet Γ and an encoding of computations $Cod_M : \Sigma^* \to \Gamma^*$ such that the language $\textsc{Valc}(M) = \{w\natural Cod_M(w) \mid w \in \Sigma^*$ and $Cod_M(w)$ is an accepting computation$\}$ over the alphabet $\Omega = \Sigma \cup \Gamma \cup \{\natural\}$ is recursive (see [30]). Clearly, we can choose M with $\mathcal{L}(M)$ not recursive and construct the class $\mathcal{C}^1 = (L_i^1)_{i\geq 1}$ as follows: $L_1^1 = \textsc{Valc}(M)$ and $L_i^1 = \{a^i\natural\}$ for all $i > 1$, where a is any symbol in Σ. It is an easy exercise to design a PCQ algorithm that learns $\mathcal{C}^1 = (L_i^1)_{i\geq 1}$ (asking one PCQ for the string λ suffices). However, $\mathcal{C}^1 = (L_i^1)_{i\geq 1}$ does not have property $\mathbb{A}$ (since $Pref(\textsc{Valc}(M)) \cap \Sigma^*\natural = \mathcal{L}(M)$ we deduct that $Pref(L_1^1)$ is not recursive), so $\mathcal{C}^1 = (L_i^1)_{i\geq 1} \in PCorQ \backslash PCorQ^{\mathbb{A}}$.

7.4.1.2 *Learning with PCQs versus Learning with MQs*

Let us recall that the notion of CQ itself appeared as an extension of the well-known and widely studied MQ. The inspiration for introducing them comes from a real-life setting (which is the case for MQs also): when children make mistakes, the adults do not reply by a simple `Yes` or `No` (the agreement is actually implicit), but they also provide them with a corrected word (or phrase). Clearly, CQs can be thought of as some more informative MQs. So, it is only natural to compare the two learning settings (learning with CQs versus learning with MQs), and to analyze their expressive power. The characterization presented in Section 7.4.1.1 leads to the following results:

- The sets *MemQ* and $PCorQ^{\mathbb{A}}$ are incomparable,
- The set *MemQ* is strictly included in *PCorQ*

For the first claim, it is enough to check that $\mathcal{C}^1 = (L_i^1)_{i\geq 1}$ is in $\mathit{MemQ}\backslash\mathit{PCorQ}^{\mathbb{A}}$ and $\mathcal{C}^2 = (L_i^2)_{i\geq 1}$ in $\mathit{PCorQ}^{\mathbb{A}}\backslash\mathit{MemQ}$, where $L_i^2 = \{a^n \mid n \in K_i\}$ for all $i \geq 1$ and $K_1, K_2, K_3, \ldots$ is the collection of all finite nonempty sets of positive integers.

The second claim can be proved using the characterization of the set *MemQ* in terms of definite finite tell-tales and Proposition 2, which gives a sufficient condition for an indexable class to be learnable with PCQs. The inclusion is strict because $\mathcal{C}^2 = (L_i^2)_{i\geq 1}$ is in $\mathit{PCorQ}\backslash\mathit{MemQ}$.

We have seen that PCQs are strictly more powerful than MQs, which means that they cannot be simulated by a finite number of MQs.

7.4.1.3 *Learning with PCQs versus Gold-style Learning Models*

Learning from queries is a one-shot learning model (i.e., the learner's first hypothesis is also the correct one) in which the learner receives rather global information (in the sense that, at any point, the oracle can be interrogated about any of the strings in the alphabet) about the object to be learned, being able to affect the sample of information, as opposed to Gold-style learning models where the information received is local (the learner cannot influence the sample) and the learner is allowed to change its current hypothesis when new information is received. Although this two approaches seem rather unrelated at first glance, there are several results that indicate the contrary [18, 21–24]. For example, it has been shown that learning with MQs is equivalent with finite learning from informant, or that an indexable class is learnable using extra superset queries if and only if there is a conservative IIM that identifies this class from text [22].

In the sequel we present where is the model of learning with PCQs placed in the existing hierarchy of both query and Gold-style learning models:

- The set $\mathit{PCorQ}^{\mathbb{A}}$ is strictly included in *ConsvTxt*,
- The set *PCorQ* is strictly included in *LimTxt*,
- The set $\mathit{LimTxt}\backslash(\mathit{PCorQ} \cup \mathit{ConsvTxt})$ is not empty,
- The sets *PCorQ* and *ConsvTxt* are incomparable.

For the first claim, in order to prove the inclusion one has to use the characterizations of the two models in terms of finite sets, that is, Theorem 7.4 that characterizes the class $\mathit{PCorQ}^{\mathbb{A}}$, and the following theorem, due to T. Zeugmann, S. Lange and S. Kapur, that characterizes the class *ConsvTxt*.

Theorem 7.5 ([41]). *The class $\mathcal{C} = (L_i)_{i\geq 1}$ is in ConsvTxt if and only if there exists an uniformly computable family $(T_i^j)_{i,j\geq 1}$ of finite sets such that*

(1) for all $L \in \mathcal{C}$, there exists i with $L_i = L$ and $T_i^j \neq \emptyset$ for almost all $j \geq 1$;
(2) for all $i, j \geq 1$, $T_i^j \neq \emptyset$ implies $T_i^j \subseteq L_i$ and $T_i^j = T_i^{j+1}$;
(3) for all $i, j, k \geq 1$, $\emptyset \neq T_i^j \subseteq L_k$ implies $L_k \not\subset L_i$

If $\mathcal{C} = (L_i)_{i\geq 1}$ is in $PCorQ^{\mathbb{A}}$ then, by Theorem 7.4, a triple of definite finite tell-tales $\langle T_i^*, F_i^*, U_i^* \rangle$ of L_i is uniformly computable for any index i. Moreover, we can assume without loss of generality that for all $i \geq 1$, T_i^* is not empty. For all $i, j \geq 1$, we define T_i^j to be the set T_i^*. Clearly, $(T_i^j)_{i,j\geq 1}$ is a uniformly computable family of finite sets. Let us show that it also satisfies the three conditions of Theorem 7.5.

(1) *for all $L \in \mathcal{C}$, there exists i with $L_i = L$ and $T_i^j \neq \emptyset$ for almost all $j \geq 1$;*
True - they are all nonempty.
(2) *for all $i, j \geq 1$, $T_i^j \neq \emptyset$ implies $T_i^j \subseteq L_i$ and $T_i^j = T_i^{j+1}$;*
True.
(3) *for all $i, j, k \geq 1$, $\emptyset \neq T_i^j \subseteq L_k$ implies $L_k \not\subset L_i$.*
This last condition translates to: *for all $i, k \geq 1$, $T_i^* \subseteq L_k$ implies $L_k \not\subset L_i$.* So, let us assume that there exist $i, k \geq 1$ such that $T_i^* \subseteq L_k$ and $L_k \subset L_i$. It follows that $Pref(L_k) \subseteq Pref(L_i)$, and hence $\Sigma^* \backslash Pref(L_k) \supseteq \Sigma^* \backslash Pref(L_i)$. Keeping in mind that $U_i^* \subseteq \Sigma^* \backslash Pref(L_i)$ we obtain that $U_i^* \subseteq \Sigma^* \backslash Pref(L_k)$. Moreover, $F_i^* \subseteq \Sigma^* \backslash L_i$ and $\Sigma^* \backslash L_k \supseteq \Sigma^* \backslash L_i$ imply $F_i^* \subseteq \Sigma^* \backslash L_k$. Since L_k is consistent with the triple $\langle T_i, F_i, U_i \rangle$, we have $L_i = L_k$ which contradicts our assumption.

An example of a class that separates the two models is $\mathcal{C}^3 = (L_i^3)_{i\geq 1}$, where $L_i^3 = \{a^n \mid n \in R_i\}$, $\Sigma = \{a\}$, $R_i = \cup_{p\in P_i} I(p)$, $P_1, P_2, P_3, \ldots$ is the collection of all finite nonempty sets of prime positive integers indexed, for example, in order of increasing $\prod_{p\in P_i} p$ and $I(p)$ is the set of all positive integral multiples of p.

The proof of the second inclusion is more intricate (see [33]: Theorem 7, page 48). The separating class is again $\mathcal{C}^3 = (L_i^3)_{i\geq 1}$.

Finally, for the third and forth claim the constructions of the separating classes are rather complicated, so we will not go into further details here. The reader is referred to [33], pages 48-50 for complete proofs.

7.4.2 *Learning with Length Bounded Correction Queries*

The particular choice made for the prefix correcting string is closely related to the intrinsic structure of DFAs and their properties. One may argue though that the smallest correcting string might be arbitrarily long. For example, if we take n to be any natural number, then there exists a string w in Σ^* and a (regular) language L such that the correction of w with respect to L is longer than n (take $w = \lambda$ and $L = \{a^{n+1}\}$ where a is an arbitrary symbol in Σ). So, let us see what happens when, instead of returning the smallest string in $\mathit{Tail}_L(w)$, we return all tails shorter than a given fixed natural number. Clearly, if there is no possible short correction, the oracle will return the empty set.

Let us fix an integer l. Given a language L and a string w, we define the *l-bounded correction of w with respect to L* (denoted $C_L^l(w)$) as the set of all strings v of length at most l such that $w \cdot v$ is in L. Formally, C_L^l is a function from Σ^* to $\mathcal{P}(\Sigma^*)$ such that for any w in Σ^*, $C_L^l(w) = \{v \in \mathit{Tail}_L(w) \mid |v| \leq l\}$. Note that $\lambda \in C_L^l(w)$ if and only if $w \in L$.

So, let us consider the model in which the learner must identify the target language after asking a finite number of *l-bounded correction queries* (lBCQs). Since any 0BCQ can be simulated by an MQ and the other way around, it is clear that for $l = 0$, learning with lBCQs is equivalent to learning with MQs. It can also be shown that the same property holds for an arbitrary l. So let us denote by *lBCorQ* the collection of all indexable classes $\mathcal{C}$ for which there exists a query learner $\mathcal{A}lg$ such that $\mathcal{A}lg$ learns $\mathcal{C}$ using lBCQs. The following result holds.

Proposition 7.6. *For any $l \geq 0$, lBCorQ = MemQ.*

This proposition is basically saying that having an oracle that can return at once the answers for more than one MQ (one lBCQ contains the answer for $1 + |\Sigma| + \ldots + |\Sigma|^l$ MQs) does not increase the learnability power of the model (that is, the learning with MQs model). The result was somehow expected if we recall that time complexity issues are neglected in our analysis. Moreover, this allows us to talk about the model of learning with *length bounded correction queries* (LBCQs) in general, without specifying a given length bound. Therefore, we denote by *LBCorQ* the collection of all language classes $\mathcal{C}$ for which there exists an $l \geq 0$ and a query learner $\mathcal{A}lg$ such that $\mathcal{A}lg$ learns $\mathcal{C}$ using a finite number of lBCQs.

7.4.3 *Learning with Edit Distance Correction Queries*

Following [8], we define the *edit distance correction* of a string w with respect to the nonempty language L by:

$$\textsc{Edc}_L(w) = \begin{cases} \texttt{Yes}, & \text{if } w \in L, \text{ and} \\ \text{one string of } \{w' \in L \mid d(w,w') \text{ is minimum}\}, & \text{if } w \notin L. \end{cases}$$

Similarly, we denote by $\textsc{Mq}_L(w)$ the oracle's answer when it is queried with the string w for the target language L. That is, $\textsc{Mq}_L(w) = \texttt{Yes}$ if $w \in L$, and `No` otherwise.

Note that $\textsc{Edc}_L(w) = \texttt{Yes}$ if and only if w is in L. Clearly, any oracle answering *edit distance correction queries* (EDCQs) would also give us the answer for the corresponding MQ. If we denote by *EditCorQ* the collection of all indexable classes $\mathcal{C}$ for which there exists a query learner $\mathcal{A}lg$ such that $\mathcal{A}lg$ learns $\mathcal{C}$ using edit distance correction queries, it is clear that *EditCorQ* includes *MemQ*. It can be shown that the following equality holds.

Theorem 7.7. *EditCorQ* = *MemQ*.

Proof. Let us first note that having an MQ oracle allows us to compute the value of $\textsc{Edc}_L(w)$ for any language $L \neq \emptyset \subseteq \Sigma^*$ and any w in Σ^* using a finite number of MQs. For that, we observe that for a given $w \in \Sigma^*$ and $r \in \mathbb{N}$ there are only a finite number of strings $v \in \Sigma^*$ such that $d(w,v) = r$, and there exists an algorithm who can generate all these strings (remember that we are not concerned with the complexity of the resulting algorithm - the only requirement is to return the answer after finite steps). So all we have to do is to ask MQs for all strings u at a certain distance i ($i = 0, 1, 2, \ldots$) from w until we receive an answer `Yes` from the oracle.

Now, if we take $\mathcal{C}$ to be a language class in *EditCorQ*, then there exists an algorithm $\mathcal{A}lg$ that learns $\mathcal{C}$ using EDCQs. We can modify $\mathcal{A}lg$ to use the MQ oracle to get the answers for the EDCQs as described above. We obtained an algorithm that learns $\mathcal{C}$ using MQs only, so *EditCorQ* $\subseteq$ *MemQ* which concludes our proof. □

7.4.4 *Remarks and Further Research*

Section 7.4 was dedicated to the learnability power of several types of CQs used alone when complexity issues are neglected. A complete picture displaying the relations between all discussed versions of query learning and Gold-style learning is obtained (Figure 7.1).

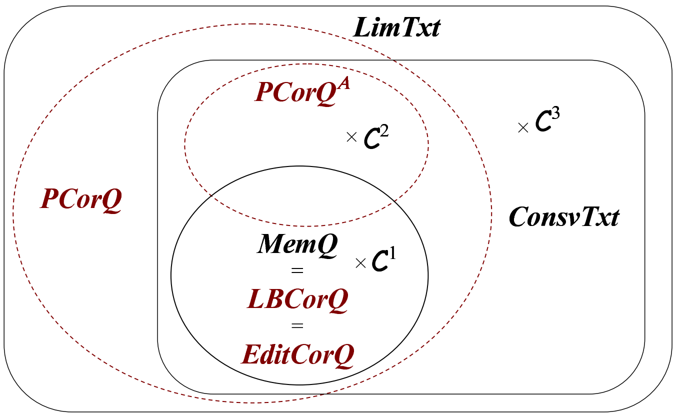


Fig. 7.1 The hierarchy of Gold-style and query learning models.

Our results can be summarized as follows:

- learning with LBCQs and learning with EDCQs are as powerful as learning with MQs;
- learning with PCQs is strictly more powerful than learning with MQs, and strictly less powerful than learning in the limit from text;
- the sets *PCorQ* and *ConsvTxt* are incomparable, but any class which is in *PCorQ* and not in *ConsvTxt* has the following property: at least one of the languages in the class has the prefix not recursive.

There are several directions that deserve further investigation. For example, one may study language learning with CQs in other standard settings like *bounded learning* [37] or *incremental learning* (see [19] and the references therein), or completely new ones: learning from positive examples and CQs, or learning with CQs and (a bounded number of) EQs.

7.5 Polynomial Time Learning with Correction Queries

In the previous section we have investigated the power of the query learning model when an oracle answering correction queries is available. Several types of CQs have been considered, and for each of them a characterization in terms of finite sets has been given. Moreover, we compared the newly introduced models with other well-known query learning and Gold-style learning models.

Although from a theoretical point of view it is important to know what language classes are inferable in finite time steps, what matters in practice is the efficiency of the algorithms. And indeed, if we need a machine capable of learning a given language class, it does not help us too much if we know that we will get the answer in 153 years. That is why this section is dedicated to polynomial time query learners that have access to CQ oracles. We show that there are several nontrivial language classes which are polynomial time learnable with CQs, and we investigate the relations existing between the different types of CQs introduced so far when complexity issues are taken into consideration. Moreover, we distinguish situations when although two query types are equally powerful in the general case, the equality is not preserved under efficiency constraints.

The reader may notice that in this section we switch from learning languages to learning grammars. Why? First of all, because in practice what we usually want to learn is a grammar, and not a sequence of words, whereas from a theoretical point of view it is less important if a given language class is learnable with respect to a specific (and hence restrictive) hypotheses space. Secondly, if we talk about efficient algorithms, then one needs to define first what *polynomial time learning* is, and hence a reasonable measure for the size of a language is required. Note that for infinite languages, having a polynomial algorithm in the number of elements of the language does not make much sense. Moreover, any grammar is a compact way of describing the language generated by it. That is why from now on, by the *size of a language* we understand the size of the smallest grammar (from a given hypotheses space) generating it.

So, while in the previous section the hypotheses space was the language class itself, in this section the hypotheses space is by default a *class preserving* one, that is, $\mathcal{H} = (G_i)_{i\geq 1}$ is such that for any $i \geq 1$, $\mathcal{L}(G_i)$ is a language in the class to be learned (see [40] for more details). We will omit specifying which is the hypothesis space whenever it is clear from the context.

We introduce first some terminology and notations. Let $\mathcal{C} = (L_i)_{i\geq 1}$ be an indexable class. We say that $\mathcal{C}$ is *polynomially time learnable with MQs* if there exists a polynomial $\mathtt{p}(\cdot)$ and an algorithm $\mathcal{A}lg$ that learns any language L in $\mathcal{C}$ in time $\mathcal{O}(\mathtt{p}(size(L)))$ by asking a finite number of MQs. We denote the collection of all indexable classes $\mathcal{C}$ which are polynomially time learnable with MQs by *PolMemQ*. *PolPCorQ*, *PollBCorQ* and *PolEditCorQ* are defined similarly for the classes of languages polynomially time learnable with PCQs, lBCQs and EDCQs, respectively.

7.5.1 *Polynomial Time Learning with PCQs*

Let us start our study with the type of CQ that was chronologically the first one introduced, and that gives, in the same time, the biggest power to the learner amongst all of them. Intuitively, one should be looking for language classes for which this sort of corrections offers some information about the intrinsic structure of the language. We investigate two well-know language classes: the class of pattern languages and the class of k-reversible languages.

7.5.1.1 *The Class of Pattern Languages*

Initially introduced by Angluin [1] to show that there are nontrivial classes of languages learnable from text in the limit, the class of pattern languages has been intensively studied in the context of language learning ever since. Polynomial time algorithms have been given for learning pattern languages using one or more examples and queries [25], or just superset queries [5], or for learning k-variables pattern languages from examples [15], etc.

Let us denote by $\mathbb{P}$ the class of all pattern languages over Σ.

Theorem 7.8. *The class* $\mathbb{P}$ *is in PolPCorQ.*

Proof. We exhibit an algorithm that learns any pattern language in polynomial time using a finite number of PCQs.

Suppose that the target language is a pattern language $L(\pi)$, where π is in normal form. Our algorithm is based on the following simple observations.

If w is the smallest string (in lex-length order) in $L(\pi)$ and $n = |w|$, then for all i in $\{1, \ldots, n\}$, we have:

- if $\pi[i] = a$ for some a in Σ, then for all $b \in \Sigma \backslash \{a\}$, $C_L(w[1 \ldots i-1]b)$ is either Θ, or longer than $w[i+1 \ldots n]$.
- if $\pi[i]$ is a variable x such that i is the position of the leftmost occurrence of x in π, then $|C_L(w[1 \ldots i{-}1]a)| = |w[i{+}1 \ldots n]|$ for any symbol $a \in \Sigma$; moreover, we can detect the other occurrences of the variable x in π by just checking the positions where the strings $C_L(w[1 \ldots i-1]a)$ and $w[i+1 \ldots n]$ do not coincide, where a is any symbol in $\Sigma \backslash \{w[i]\}$;

From the above mentioned observations, it is clear that the algorithm described above terminates in finite steps and outputs the pattern π after asking a finite number of PCQs (recall that, by convention, the length of the symbol Θ is 0).

Algorithm 2 An algorithm for learning the class $\mathbb{P}$ with PCQs.

```
1: $w := C_L(\lambda), n := |w|, var := 0$
2: for $i := 1$ to $n$ do
3:    $\pi[i] := null$
4: end for
5: for $i := 1$ to $n$ do
6:    if $(\pi[i] = null)$ then
7:       choose $a \in \Sigma \backslash \{w[i]\}$ arbitrarily
8:       $v := C_L(w[1 \dots i-1]a), m := |v|$
9:       if $(v = \Theta$ or $m > n - i)$ then
10:          $\pi[i] := w[i]$
11:      else
12:          $var := var + 1, \pi[i] := x_{var}$
13:          for all $j \in \{1, \dots, m\}$ for which $v[j] \neq w[i+j]$ do
14:             $\pi[i+j] := x_{var}$
15:          end for
16:      end if
17:   end if
18: end for
19: output $\pi$
```

Moreover, for each symbol in the pattern, the algorithm makes at most $n + 1$ comparisons, where n is the length of the pattern. This implies that the total running time of the algorithm is bounded by $n(n + 1)$, that is $\mathcal{O}(n^2)$. It is easy to see that the query complexity is linear in the length of the pattern since the algorithm does not ask more than $n + 1$ PCQs. □

7.5.1.2 *The Class of k-Reversible Languages*

Angluin introduces the class of k-reversible languages in [3], and shows that it is inferable from positive data in the limit. Later on, she proves that there is no polynomial algorithm that exactly identifies DFAs for 0-reversible languages using only equivalence queries [6].

We study the learnability of the class k-*Rev* in the context of learning with PCQs, and provide the reader with a polynomial time algorithm that identifies any k-reversible language after asking a finite number of PCQs. In order to do so, we exhibit one important property of k-reversible languages. For any string u in Σ^*, we define the function $row_k(u) : \Sigma^{\leq k} \to \Sigma^* \cup \{\Theta\}$ by $row_k(u)(v) = C_L(uv)$.

Proposition 7.9. *Let L be a k-reversible language. Then, for all $u_1, u_2 \in \Sigma^*$, $u_1 \equiv_L u_2$ if and only if $row_k(u_1) = row_k(u_2)$.*

Proof. We have to show that given a k-reversible language L, two strings u_1 and u_2 are equivalent with respect to the language L if and only if $C_L(u_1v) = C_L(u_2v)$ for all v in $\Sigma^{\leq k}$.

Note that the "only if" direction trivially holds for all regular languages. So, let us assume that there exists strings u_1, u_2 in Σ^* such that $u_1 \not\equiv_L u_2$ and $C_L(u_1v) = C_L(u_2v)$ for all v in $\Sigma^{\leq k}$. Hence, there must exist $w \in \Sigma^*$ such that either

- $u_1w \in L$ and $u_2w \notin L$, or
- $u_1w \notin L$ and $u_2w \in L$.

Let us assume the former case (the other one is similar).

(1) If $|w| \leq k$, then $w \in \Sigma^{\leq k}$, and hence $C_L(u_1w) = C_L(u_2w)$. But $u_1w \in L$ implies $C_L(u_1w) = \lambda$, and so $C_L(u_2w) = \lambda$ which contradicts $u_2w \notin L$.

(2) If $|w| > k$, then there must exist $v, w' \in \Sigma^*$ such that $w = vw'$ and $|v| = k$. By assumption, $u_1vw' \in L$ and $u_2vw' \notin L$, so $u_1v \not\equiv_L u_2v$. On the other hand, since $v \in \Sigma^{\leq k}$ we have $C_L(u_1v) = C_L(u_2v) = v'$. Because $u_1v \cdot w' \in L$, $Tail_L(u_1v) \neq \emptyset$ and hence $C_L(u_1v) \in \Sigma^*$. Since L is k-reversible, $u_1vv' \in L, u_2vv' \in L$ and $|v| = k$, we get $Tail_L(u_1v) = Tail_L(u_2v)$ which contradicts $u_1v \not\equiv_L u_2v$.

We showed that if $C_L(u_1v) = C_L(u_2v)$ for all v in $\Sigma^{\leq k}$, then $u_1 \equiv_L u_2$ which concludes our proof. $\square$

This proposition is saying that each equivalence class in $\Sigma^*/_{\equiv_L}$ is uniquely identified by the values of function row_k on $\Sigma^{\leq k}$. So, if $\mathcal{A}_L = (Q, \Sigma, \delta, q_0, F)$ is the minimal complete DFA for the k-reversible language L, then the values of function $row_k(u)$ on $\Sigma^{\leq k}$ uniquely identify the state $\delta(q_0, u)$.

The algorithm follows the lines of L*. We have an *observation table* denoted by $(S, \Sigma^{\leq k}, C)$ in which lines are indexed by the elements of a prefix-closed set S, columns are indexed by the elements of $\Sigma^{\leq k}$, and the element of the table situated at the intersection of line u with column v is $C_L(uv)$.

We say that the observation table $(S, \Sigma^{\leq k}, C)$ is *k-closed* if for all $u \in S$ and $a \in \Sigma$, there exists $u' \in S$ such that $row_k(u') = row_k(ua)$. Moreover,

$(S, \Sigma^{\leq k}, C)$ is *k-consistent* if for all $u_1, u_2 \in S$, $row_k(u_1) \neq row_k(u_2)$. It is clear that if the table $(S, \Sigma^{\leq k}, C)$ is k-consistent and S has exactly n elements, where n is the index of L, then the strings in S are in bijection with the elements of $\Sigma^*/_{\equiv_L}$.

We start with $S = \{\lambda\}$, and then increase the size of S by adding elements with distinct row values. An important difference between our algorithm and L* is that in our case the number of columns of the table is never modified during the run of the algorithm (in L*, there is only one column in the beginning, and then more columns are gradually added when needed).

For any k-closed and k-consistent table $(S, \Sigma^{\leq k}, C)$, we construct the automaton $\mathcal{A}(S, \Sigma^{\leq k}, C) = (Q, \Sigma, \delta, q_0, F)$ as follows: $Q = \{row_k(u) \mid u \in S\}$, $q_0 = row_k(\lambda)$, $F = \{row_k(u) \mid u \in S \text{ and } C_L(u) = \lambda\}$, and $\delta(row_k(u), a) = row_k(ua)$ for all $u \in S$ and $a \in \Sigma$.

To see that this is a well-defined automaton, note that since S is a nonempty prefix-closed set, it must contain λ, so q_0 is defined. Because S is k-consistent, there are no two elements u_1, u_2 in S such that $row_k(u_1) = row_k(u_2)$. Thus, F is well defined. Since the observation table $(S, \Sigma^{\leq k}, C)$ is k-closed, for each $u \in S$ and $a \in \Sigma$, there exists u' in S such that $row_k(ua) = row_k(u')$, and because it is k-consistent, this u' is unique. So δ is well defined.

Theorem 7.10. *The class k-Rev is in PolPCorQ.*

Proof. The following algorithm learns any k-reversible language L with PCQs and has a total running time bounded by a polynomial in the size of the target language. □

Algorithm 3 does not work in general for arbitrary regular languages, as one can see from Example 7.11.

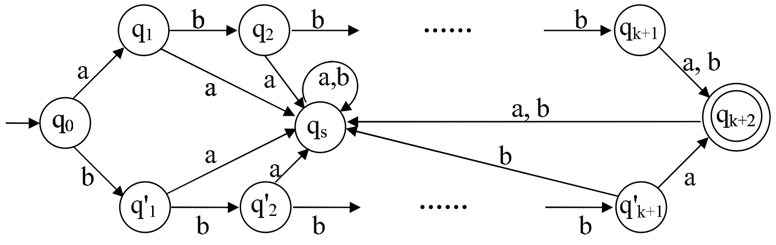


Fig. 7.2 The minimal DFA for the language $L_k = \{ab^k a, ab^k b, b^{k+1} a\}$.

Algorithm 3 An algorithm for learning the class k-*Rev* with PCQs.

```
1: $S := \{\lambda\}$
2: $closed :=$ TRUE
3: update the table by asking PCQs for strings in $\Sigma^{\leq k+1}$
4: repeat
5:     if $\exists u \in S$ and $a \in \Sigma$ such that $row_k(ua) \notin row_k(S)$ then
6:         add $ua$ to $S$
7:         update the table by asking PCQs for strings in $\{uabv \mid b \in \Sigma, v \in \Sigma^{\leq k}\}$
8:         $closed :=$ FALSE
9:     end if
10: until $closed$
11: output $\mathcal{A}(S, \Sigma^{\leq k}, C)$ and halt.
```

Example 7.11. *Let us fix* $k \geq 0$, *and consider the finite (and hence regular) language* $L_k = \{ab^k a, ab^k b, b^{k+1}a\}$ *(*$\mathcal{A}_{L_k}$ *is given in Figure 7.2).*

It is easy to check that if we run Algorithm 3 on the language L_k we get an automaton in which the strings a and b represent the same state.

7.5.1.3 *Polynomial Time Learning with PCQs versus MQs*

Let us make a step further toward understanding the differences and similarities between MQs and PCQs, by taking into consideration the efficiency of the learning algorithms. We have seen in Section 7.4.1.2 that learning with MQs is a strictly weaker model then learning with PCQs when time complexity issues are neglected. One may think that from this it can be automatically inferred that polynomial time learnability with MQs implies polynomial time learnability with PCQs. We show by an example that one should not rush into drawing such conclusions.

Indeed, let us first recall that learning with EQs is strictly more powerful than learning with PCQs when ignoring time complexity: *PCorQ* is strictly included in *LimTxt* and *LimTxt* is strictly included in *EquQ* (see Section 7.3.3). On the other hand, the class of 0-reversible languages is polynomially learnable with PCQs (one may use Algorithm 3), but not identifiable in polynomial time with EQs [6].

The following result trivially holds.

Lemma 7.12. *PolMemQ* $\subseteq$ *PolPCorQ.*

We show that the inclusion is strict using pattern languages as the separating case.

Theorem 7.13. *The class $\mathbb{P}$ is in PolPCorQ$\backslash$PolMemQ.*

Proof. We know that the class $\mathbb{P}$ is in *PolPCorQ* (see Section 7.5.1.1). Assume now that $\mathbb{P}$ is in *PolMemQ*, and consider the class $\mathcal{S} = (L_w)_{w \in \Sigma^*}$, $L_w = \{w\}$ of singletons over the fixed alphabet Σ. Because every language L_w in $\mathcal{S}$ can be written as a pattern language ($L_w = L(w)$, where w is a pattern without any variables), our assumption would imply that $\mathcal{S}$ is also in *PolMemQ*. It is clear though that any algorithm that learns $\mathcal{S}$ using MQs might need to ask $|\Sigma| + |\Sigma|^2 + \ldots + |\Sigma|^{|w|}$ MQs in the worst case to learn a given language L_w (the learner's best strategy in this case is to try all strings in Σ^* in lexicographical order), which leads to a contradiction. $\square$

Note that although $\mathbb{P}$ is not polynomially time learnable with MQs, it is in *MemQ* (see [27], page 266). However, there are classes of languages in *PolPCorQ* which cannot be learned at all (polynomially or not) using MQs, as we will see in the sequel.

Theorem 7.14. *The class k-Rev is in PolPCorQ$\backslash$MemQ.*

Proof. First, let us recall that the class *k-Rev* is in *PolPCorQ* (see Theorem 7.10). To show that *k-Rev* is not in *MemQ*, we use Mukouchi's characterization of the class *MemQ* in terms of *pairs of definite finite tell-tales* [27]: an indexable class $\mathcal{C} = (L_i)_{i \geq 1}$ belongs to *MemQ* if and only if a pair of definite finite tell-tales of L_i is uniformly computable for any index i. The finite sets $\langle T, F\rangle$ form a a pair of definite finite tell-tales for a language L in $\mathcal{C}$ if $T \subseteq L$, $F \subseteq \Sigma^* \backslash L$ and for all L' in $\mathcal{C}$, if$T \subseteq L'$ and $F \subseteq \Sigma^* \backslash L'$ imply $L = L'$.

So, let us assume that *k-Rev* is in *MemQ*. Consider the alphabet Σ such that $\{a, b\} \subseteq \Sigma$, and the language $L = \{a\}$. Clearly, L is in *k-Rev* for all $k \geq 0$ and hence a pair of definite finite tell-tales $\langle T, F\rangle$ is computable for L. This means that $T \subseteq L$ and F is a finite set included in $\Sigma^* \backslash \{a\}$. Let us take m to be $\max\{|w| \mid w \in F\}$ if $F \neq \emptyset$ and 0 otherwise, and consider the language $L' = \{a, ba^m b\}$. It is clear that L' is in *k-Rev* for all $k \geq 0$, $T \subseteq L'$ and $F \subseteq \Sigma^* \backslash L'$. Moreover, $L' \neq L$ which leads to a contradiction. $\square$

On the other hand, very simple classes of languages cannot be learned in polynomial time using PCQs. For example, if we take $\bar{\mathcal{S}}$ to be $\bar{\mathcal{S}} =$

$(\bar{L}_w)_{w\in\Sigma^*}$, where $\bar{L}_w = \Sigma^*\backslash\{w\}$, then any algorithm would require at least $1+|\Sigma|+|\Sigma|^2+\ldots+|\Sigma|^{|w|}$ PCQs in order to learn $\bar{L}_w$ (learning $\bar{L}_w$ is equivalent to "guessing" the unique w in Σ^* that has as correction a nonempty string; note that in this case no adversary strategy can be used by the teacher, since he is required to provide the correct answer). Figure 7.3 displays the relations between the two models.

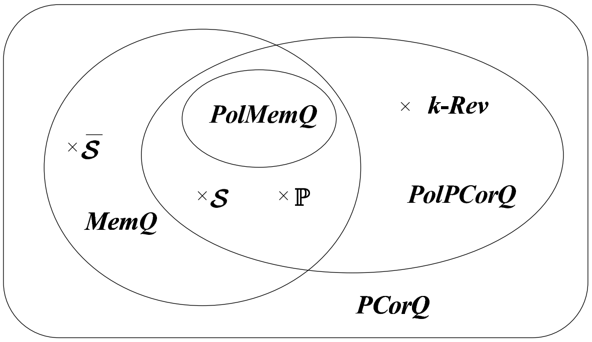


Fig. 7.3 PCQ learning versus MQ learning.

7.5.2 *Polynomial Time Learning with LBCQs*

Let l be a fixed nonnegative integer, and let us denote by *PollBCorQ* the collection of all indexable classes $\mathcal{C}$ for which there exists a polynomial $\mathtt{p}(\cdot)$ and an algorithm $\mathcal{A}lg$ that learns any language L in $\mathcal{C}$ in time $\mathcal{O}(\mathtt{p}(size(L)))$ by asking a finite number of lBCQs. Clearly, we only consider those language classes $\mathcal{C}$ for which the size of its languages is independent of l (more details are presented further on in this section).

Lemma 7.15. *Pol0BCorQ = PolMemQ.*

The result is straightforward from the definition of $C_L^0(w)$.

Lemma 7.16. *Pol(l-1)BCorQ = PollBCorQ for any $l \geq 1$.*

Proof. Since one can easily extract the answer to an $(l-1)$BCQ from the corresponding lBCQ, it is clear that *Pol(l-1)BCorQ* is included in *PollBCorQ*. Let us now show that *PollBCorQ* is included in *Pol(l-1)BCorQ*. Assume $\mathcal{C}$ is an indexed family of languages in *PollBCorQ* and let $\mathcal{A}lg$ be a polynomial time algorithm that learns $\mathcal{C}$ with lBCQs. Note that for any language L over Σ, any $w \in \Sigma^*$ and any $l \geq 1$,

$$\begin{aligned}C_L^l(w) &= \{u \in \Sigma^{\le l} \mid wu \in L\}\\ &= \{u \in \Sigma^{\le l-1} \mid wu \in L\} \cup \{au \mid a \in \Sigma, u \in \Sigma^{l-1} \text{ and } wau \in L\}\\ &= C_L^{l-1}(w) \cup \{au \mid a \in \Sigma, u \in C_L^{l-1}(wa)\}\\ &= C_L^{l-1}(w) \cup \textstyle\bigcup_{a\in\Sigma} aC_L^{l-1}(wa).\end{aligned}$$

So, one can modify $\mathcal{A}lg$ such that instead of asking an lBCQ for the string w, to ask a finite number of $(l-1)$BCQs ($|\Sigma|+1$ queries to be precise) for the strings wa with a in $\{\lambda\} \cup \Sigma$. Clearly, the modified algorithm is still polynomial. Hence, *Pol(l-1)BCorQ* equals *PollBCorQ*. □

The following theorem is a direct consequence of Lemmas 7.15 and 7.16.

Theorem 7.17. *PollBCorQ = PolMemQ for any* $l \geq 0$.

Therefore, we can introduce the notation *PolLBCorQ* for the collection of all language classes that are efficiently learnable with LBCQs. Moreover, by combining the theorem above with Lemma 7.12 and Theorem 7.13, one gets the following corollary.

Corollary 7.18. *PollBCorQ* $\subsetneq$ *PolPCorQ*.

In the beginning of this section we pointed out that we should think of l as a constant. By failing to do so, we may end up with contradictory results, as one can see in Example 7.19.

Example 7.19. *Let* $\mathcal{S}_l = (L_w)_{w\in\Sigma^l}$, $L_w = \{w\}$, *be the class of all singleton languages of size* $l+2$, *where the hypotheses space contains only canonical DFAs accepting singleton languages. We can imagine a very simple lBCQ algorithm to learn this class. The learner would simply ask one lBCQ, for the string* λ, *and then output the string received as an answer. Since in this case the set of possible corrections contains just one string of length* $size(L) - 2$ *where* L *is the target language, it means that such an algorithm would work in linear time in the size of the language. On the other hand, with an MQ oracle, any learner would have to ask at least* $|\Sigma|^l - 1$ *MQs, hence an exponential number of queries in the size of the target language. Thus, one might think that this language class invalidates the result of Theorem 7.17. The trick here is that* l *is no longer a constant for the class* $\mathcal{S}_l$, *and that if we do think of* l *as a constant, then* $|\Sigma|^l - 1$ *is a constant as well.*

So, what we have learned in this section is that having the possibility to get answers for more than one MQ at once does not add any more learning power, even if we impose time restrictions.

7.5.3 *Polynomial Time Learning with EDCQs*

We continue the analysis done in Section 7.4.3 on the power of learning with edit distance based correction queries, this time by taking into account time complexity issues. We have seen that what happens in the general model does not necessary carry on to the polynomially bounded model. Let us recall the results we have so far:

$PCorQ \subset EquQ$	$PolPCorQ \not\subset PolEquQ$
$MemQ \subsetneq PCorQ$	$PolMemQ \subsetneq PolPCorQ$
$MemQ = LBCorQ$	$PolMemQ = PolLBCorQ$
$MemQ = EditCorQ$	$PolMemQ \overset{?}{=} PolEditCorQ$

So in the case of LBCQs versus MQs, as well as in the case of PCQs versus MQs, the relation existing in the general case is preserved in the polynomial learning model, while for the EQs versus PCQs it does not happen. On the other hand, we already know that $MemQ = EditCorQ$, and the question is whether or not $PolMemQ = PolEditCorQ$, where by $PolEditCorQ$ we denote, as usual, the collection of all indexable classes $\mathcal{C}$ for which there exists a polynomial $\mathtt{p}(\cdot)$ and an algorithm $\mathcal{A}lg$ that learns any language L in $\mathcal{C}$ in time $\mathcal{O}(\mathtt{p}(size(L)))$ by asking a finite number of EDCQs.

In this section we answer this question in the negative way, and we describe some of the algorithms which use alternative CQs based on edit distance existing in the literature.

Lemma 7.20. *$PolMemQ \subsetneq PolEditCorQ$.*

Proof. One may show that $PolMemQ$ is included in $PolEditCorQ$ using an argument similar with the one presented in the proof of Lemma 7.12, the only difference being that in the case of EDCQs, the new algorithm $\mathcal{A}lg'$ has to check whether or not $\textsc{Edc}_L(w)$ equals `Yes`, for all the strings w submitted to the oracle by the learner of the original algorithm $\mathcal{A}lg$ (L is the target language).

Moreover, if $\mathcal{S}$ is the class of singleton languages over the alphabet Σ, then $\mathcal{S}$ can be used as a separating language class:

- $\mathcal{S} \notin PolMemQ$ (see the proof of Theorem 7.13),
- one may imagine a very simple edit distance query algorithm for this class. Indeed, it is enough to ask an EDCQ for an arbitrarily chosen

string w. The algorithm just outputs w, if the oracle's answer is Yes, and w' if the answer returned by the oracle is the string w'. Since the algorithm described above is polynomial in the size of the target language, we conclude that $\mathcal{S} \in$ *PolEditCorQ*.

□

So, we have seen that singleton languages can be learned in polynomial time using EDCQs. Another example of a language class in *PolEditCorQ* is the class of *balls of strings*, named like this because of their resemblance to a disk when we imagine their geometrical interpretation. We recall that given a string w and a real number r, the ball of center w and radius r is $B_r(w) = \{v \in \Sigma^* \mid d(v, w) \leq r\}$. In [8], an algorithm for learning the class of all balls of strings using EDCQ is given. Moreover, the authors of the above mentioned paper show that the number of EDCQs used by this algorithm can be bounded by $\mathcal{O}(|\Sigma| + |w| + r)$. Furthermore, for what they called q-good balls (the balls $B_r(w)$ for which there exists a polynomial $\mathsf{q}(\cdot)$ such that the radius r is no longer than $\mathsf{q}(|w|)$), the algorithm runs in polynomial time in the size of (the representation of) the language.

7.5.4 *Remarks and Further Research*

In Section 7.4.4 we exhibited a complete picture of the relations existing between several models of learning with CQs and other learning models (both Gold-style and query learning). We have seen that when we neglect time complexity issues we can characterize the newly introduced query models in terms of finite sets, and that learning with LBCQs and EDCQs is basically the same as learning with MQs, whereas PCQs are the only ones adding some power to the model.

When we restrict to polynomial time algorithms, things are changing. And although having an LBCQs oracle does still not improve on the learnability power with respect to the MQ learning model, an EDCQ oracle or a PCQ oracle does. It is not clear what relation is between learning with EDCQs and learning with PCQs when we restrict to efficient algorithms. We conjecture that the two classes are incomparable (see Figure 7.4).

Let us first notice that the class of singleton languages belongs to *PolPCorQ* $\cap$ *PolEditCorQ*. Moreover, we argue that *PolPCorQ*$\setminus$*PolEditCorQ* $\neq \emptyset$. Indeed, the class *k-Rev* is in *PolPCorQ* and not in *MemQ* (by Theorem 7.14), and since *EditCorQ* $=$ *MemQ* (by Theorem 7.7), we obtain that k-reversible languages are not learnable with EDCQs either. Hence, *k-Rev* $\in$ *PolPCorQ*$\setminus$*PolEditCorQ*.

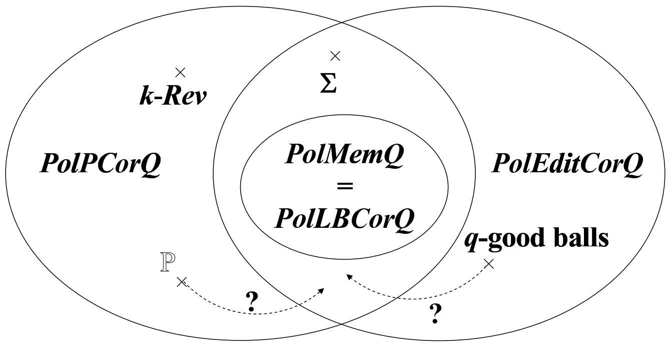


Fig. 7.4 Different types of correction queries.

If for the class of k-reversible languages it was easy to decide whether or not it is in *PolEditCorQ*, we cannot say the same thing for the class of pattern languages. Kinber [17] describes an efficient algorithm that learns $\mathbb{P}$ with the modified type of EDCQs with the strong requirement that the oracle must not return as a correction any of the strings which appeared before. We strongly believe that this requirement is actually mandatory, i.e., there is no algorithm that can learn the class $\mathbb{P}$ with standard EDCQs. Far from being a proof, Example 7.21 shows, nevertheless, on what our intuition is based on.

Example 7.21. *Let $\pi = 1xyyy111$ and $w_0 = 11000111$ a string of minimum length in $L(\pi)$. Then, although it is easy to determine the position of single variables in the pattern (asking whether or not $w = 1\boldsymbol{0}000111$ is in the language suffices), it is quite hard to find a way to distinguish between constants and multiple variables (i.e., variables which appear more than once in the pattern) if we are faced with an unfriendly oracle. Suppose the learner asks, for example, if $11\boldsymbol{1}00111$ is in the target language and the oracle returns as a correction the string w_0. In this case, there is no way the learner can deduce whether the 3^{rd} symbol of the pattern is 0, or a variable which appears more then once in π. On the other hand, changing pairs (or triples, quadruples, etc.) of symbols at once increases dramatically the number of queries needed.*

To complete the picture, we would like to be able to say if there are language classes in $PolEditCorQ \setminus PolPCorQ$. A possible candidate is the class of q-good balls of strings, which is known to be learnable with EDCQs in polynomial time.

Another future work direction is to find characterizations for the language classes polynomial time learnable with (each type of) CQs. One can investigate, for example, the *teaching dimension of a concept class* [12] (that is, the minimum number of corrections a teacher must reveal to uniquely identify any concept in the class), or the computational power of polynomial time query learning systems for different correction query types as in [36, 38]. Slightly relaxed learning criteria may lead to new learnability results. For example, in the *bounded learning* framework introduced by Watanabe [37], the learning condition does not insist in exactly identifying a target language L, but only requires that a learner should return, for any given length bound m, a language L' such that L and L' coincide up to length m (i.e., $L \cap \Sigma^{\leq m} = L' \cap \Sigma^{\leq m}$).

References

1. Angluin, D. (1979). Finding patterns common to a set of strings (extended abstract), in *Proc.* 11th *Annual ACM Symposium on Theory of Computing (STOC '79)* (ACM Press, New York, NY, USA), pp. 130–141.
2. Angluin, D. (1980). Inductive inference of formal languages from positive data, *Information and Control* **45**, 2, pp. 117–135.
3. Angluin, D. (1982). Inference of reversible languages, *Journal of the ACM* **29**, 3, pp. 741–765.
4. Angluin, D. (1987). Learning regular sets from queries and counterexamples, *Information and Computation* **75**, 2, pp. 87–106.
5. Angluin, D. (1988). Queries and concept learning, *Machine Learning* **2**, 4, pp. 319–342.
6. Angluin, D. (1990). Negative results for equivalence queries, *Machine Learning* **5**, 2, pp. 121–150.
7. Becerra-Bonache, L., Bibire, C. and Dediu, A. H. (2005). Learning DFA from corrections, in H. Fernau (ed.), *TAGI*, WSI-2005-14 (Technical Report, University of Tubingen), pp. 1–11.
8. Becerra-Bonache, L., de la Higuera, C., Janodet, J.-C. and Tantini, F. (2007). Learning balls of strings with correction queries, in J. N. Kok, J. Koronacki, R. L. de Mántaras, S. Matwin, D. Mladenic and A. Skowron (eds.), *Proc.* 18th *European Conference on Machine Learning (ECML '07)*, *Lecture Notes in Computer Science*, Vol. 4701 (Springer-Verlag, Berlin, Heidelberg), ISBN 978-3-540-74957-8, pp. 18–29.
9. Becerra-Bonache, L. and Yokomori, T. (2004). Learning mild context-

sensitiveness: Toward understanding children's language learning, in G. Paliouras and Y. Sakakibara (eds.), *ICGI*, *Lecture Notes in Computer Science*, Vol. 3264 (Springer-Verlag, Berlin, Heidelberg), pp. 53–64.

10. Clark, A., Coste, F. and Miclet, L. (eds.) (2008). *Proc.* 9th *International Colloquium on Grammatical Inference (ICGI '08)*, *Lecture Notes in Artificial Intelligence*, Vol. 5278 (Springer-Verlag, Berlin, Heidelberg).
11. Gold, E. M. (1967). Language identification in the limit, *Information and Control* **10**, 5, pp. 447–474.
12. Goldman, S. A. and Kearns, M. J. (1991). On the complexity of teaching, in M. Fulk and J. Case (eds.), *Proc.* 4th *Annual Workshop on Computational Learning Theory (COLT '91)* (Morgan Kaufmann Publishers Inc., San Francisco, CA, USA), pp. 303–314.
13. Hopcroft, J. E. and Ullman, J. D. (1979). *Introduction to Automata Theory, Languages, and Computation* (Addison-Wesley, Reading, Massachusetts).
14. Hopcroft, J. E. and Ullman, J. D. (1990). *Introduction To Automata Theory, Languages, And Computation* (Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA), ISBN 020102988X.
15. Kearns, M. and Pitt, L. (1989). A polynomial-time algorithm for learning k-variable pattern languages from examples, in *Proc.* 2nd *annual workshop on Computational learning theory (COLT '89)* (Morgan Kaufmann Publishers Inc., San Francisco, CA, USA), pp. 57–71.
16. Kearns, M. J. and Vazirani, U. V. (1994). *An Introduction to Computational Learning Theory* (MIT Press, Cambridge, MA).
17. Kinber, E. (2008). On learning regular expressions and patterns via membership and correction queries, in [10], pp. 125–138.
18. Lange, S. (2000). Algorithmic learning of recursive languages, Mensch und Buch Verlag.
19. Lange, S. and Grieser, G. (2002). On the power of incremental learning, *Theoretical Computer Science* **288**, 2, pp. 277–307.
20. Lange, S. and Zeugmann, T. (1992). Types of monotonic language learning and their characterization, in *Proc.* 5th *Annual Conference on Computational Learning Theory (COLT '92)* (ACM Press, New York, NY, USA), pp. 377–390.
21. Lange, S. and Zilles, S. (2004a). Comparison of query learning and gold-style learning in dependence of the hypothesis space, in S. Ben-David, J. Case and A. Maruoka (eds.), *Proc.* 15th *International Conference on Algorithmic Learning Theory (ALT '04)*, *Lecture Notes in Computer Science*, Vol. 3244 (Springer-Verlag, Berlin, Heidelberg), ISBN 3-540-23356-3, pp. 99–113.
22. Lange, S. and Zilles, S. (2004b). Formal language identification: query learning vs. Gold-style learning, *Information Processing Letters* **91**, 6, pp. 285–292.
23. Lange, S. and Zilles, S. (2004c). Replacing limit learners with equally powerful one-shot query learners, in J. Shawe-Taylor and Y. Singer (eds.), *Proc.* 17th *Annual Conference on Computational Learning Theory (COLT '04)*, *Lecture Notes in Computer Science*, Vol. 3120 (Springer-Verlag, Berlin, Heidelberg), ISBN 3-540-22282-0, pp. 155–169.

24. Lange, S. and Zilles, S. (2005). Relations between gold-style learning and query learning, *Information and Computation* **203**, 2, pp. 211–237.
25. Marron, A. and Ko, K.-I. (1987). Identification of pattern languages from examples and queries, *Information and Computation* **74**, 2, pp. 91–112.
26. Martín-Vide, C., Mitrana, V. and Păun, G. (eds.) (2004). *Formal Languages and Applications*, Studies in Fuzzyness and Soft Computing 148 (Springer-Verlag, Berlin, Heidelberg).
27. Mukouchi, Y. (1992). Characterization of finite identification, in K. P. Jantke (ed.), *Proc.* 3rd *International Workshop on Analogical and Inductive Inference (AII '92)*, *Lecture Notes in Artificial Intelligence*, Vol. 642 (Springer, London, UK), pp. 260–267.
28. Myhill, J. (1957). Finite automata and the representation of events, Tech. Rep. TR-57-624, WADD, Wright Patterson AFB, Ohio.
29. Nerode, A. (1958). Linear automaton transformations, in *Proceedings of the American Mathematical Society*, Vol. 9, pp. 541–544.
30. Okhotin, A. (2005). On language equations with symmetric difference, Techn. Rep. 734, TUCS, Turku University.
31. Tîrnăucă, C. (2008a). Learning reversible languages from correction queries only, Available on: `http://grlmc-dfilrom.urv.cat/grlmc/PersonalPages/cristina/publications.htm`.
32. Tîrnăucă, C. (2008b). A note on the relationship between different types of correction queries, in [10], pp. 213–223.
33. Tîrnăucă, C. (2009). *Language Learning with Correction Queries*, Ph.D. thesis, Rovira i Virgili University, Tarragona.
34. Valiant, L. G. (1984). A theory of the learnable, *Communications of the ACM* **27**, 11, pp. 1134–1142, doi:http://doi.acm.org/10.1145/1968.1972.
35. Wagner, R. A. and Fischer, M. J. (1974). The string-to-string correction problem, *Journal of ACM* **21**, 1, pp. 168–173, doi:http://doi.acm.org/10.1145/321796.321811.
36. Watanabe, O. (1990). A formal study of learning via queries, in M. Paterson (ed.), *Proc.* 17th *International Colloquium on Automata, Languages and Programming (ICALP '90)*, *Lecture Notes in Computer Science*, Vol. 443 (Springer-Verlag, Berlin, Heidelberg), pp. 139–152.
37. Watanabe, O. (1994). A framework for polynomial time query learnability, *Mathematical Systems Theory* **27**, pp. 211–229.
38. Watanabe, O. and Gavaldà, R. (1994). Structural analysis of polynomial-time query learnability, *Mathematical Systems Theory* **27**, 3, pp. 231–256.
39. Zeugmann, T. (2006). Inductive inference and language learning, in J. Cai, S. B. Cooper and A. Li (eds.), *Proc.* 3rd *International Conference on Theory and Applications of Models of Computation (TAMC '06)*, *Lecture Notes in Computer Science*, Vol. 3959 (Springer-Verlag, Berlin, Heidelberg), pp. 464–473.
40. Zeugmann, T. and Lange, S. (1995). A guided tour across the boundaries of learning recursive languages, in K. P. Jantke and S. Lange (eds.), *Combinatorics and Computer Science*, *Lecture Notes in Computer Science*, Vol. 961 (Springer, London, UK), ISBN 3-540-60217-8, pp. 190–258.

41. Zeugmann, T., Lange, S. and Kapur, S. (1995). Characterizations of monotonic and dual monotonic language learning, *Information and Computation* **120**, 2, pp. 155–173.

This page is intentionally left blank

Chapter 8

Applications of Grammatical Inference in Software Engineering: Domain Specific Language Development

Marjan Mernik[1] and Dejan Hrnčič
University of Maribor,
Faculty of Electrical Engineering and Computer Science,
Smetanova 17, SI-2000 Maribor, Slovenia,
E-mail: {marjan.mernik,dejan.hrncic}@uni-mb.si

Barrett R. Bryant
[1] *The University of Alabama at Birmingham,*
Department of Computer and Information Sciences,
Birmingham, AL 35294-1170, USA,
E-mail: {mernik,bryant}@cis.uab.edu

Faizan Javed
Regions Financial Corporation,
Mortgage Shared Systems,
Birmingham, AL 35244, USA,
E-mail: faizan.javed@regions.com

In this chapter, application of grammatical inference to facilitate development of domain-specific languages (DSLs) is presented. Grammatical inference techniques have been applied to infer DSL grammar from DSL programs. Such a scenario would be feasible when domain experts can provide complete DSL programs or excerpts of such programs. The results of grammatical inference, namely the inferred grammar, can be directly used to generate the DSL parser or be further examined by a software language engineer with the aim to further enhance the design of the language. To achieve this goal we developed a memetic algorithm which enables incremental grammar inference. It is a population based evolutionary algorithm enhanced with local search and a generalization phase. Some examples of DSLs from which grammars have been successfully inferred from positive samples are presented.

8.1 Introduction

Computer languages play a central role in computer science, including specifying computations which need to be performed, specifying the intended behavior of a system, and modeling the architecture of an application. Tony Hoare has said that computer languages are a programmer's most basic tools [23]. Indeed, thousands of computer languages have been developed in the very short period of the computing era. Unfortunately, many of them were designed ad-hoc and without proper language engineering principles. This is particularly true for Domain-Specific Languages (DSLs) [6] [61] [35], which are currently flourishing [20] [21] [41] [50] [51]. In contrast with general-purpose languages (GPLs), where one can address large classes of problems (e.g., scientific computing, business processing, symbolic processing, etc.), a DSL facilitates the solution of problems in a particular domain (e.g., aerospace, automotive, graphics, etc.). GPLs have been usually designed with great care and comply to language design principles [23] [52] [64] [67], especially after enough experience in language design has been accumulated. Unfortunately, this is not the case for DSLs. Design and implementation of GPLs may last several months, while development of DSLs should be more cost effective. However, there is a serious threat that such cost-effectiveness will negatively impact language design and/or implementation.

In [35] the following DSL development phases have been identified: decision, analysis, design, implementation, and deployment. For the sake of completeness we should also add a maintenance phase. Mernik, Heering and Sloane [35] also provide recurrent patterns in all phases except deployment and maintenance:

- Decision patterns identify common situations suitable for designing a new DSL,
- Analysis patterns identify common approaches to domain analysis,
- Design patterns identify common approaches of how to design a DSL, and

- Implementation patterns identify common approaches of how to implement a DSL.

The focus of this chapter is on the earlier DSL development phases. Overall, the beginning phases of DSL development are less well understood and lack proper tool support. Let us mention a few problems:

- How should results from domain analysis drive the language design process?
- Should a DSL be designed by a domain expert, GPL designer or software language engineer?
- How much domain analysis and language design is actually needed?
- Can we skip some initial DSL development phases? What are the consequences?
- Can we build tools which support us in earlier phases of DSL development?

There are no straightforward answers to these questions and actual decisions depend on many factors, such as DSL users, project budget, time to market, DSL life span, etc. For example, if the DSL user community will be small and the project budget is very tight then it might be quite feasible that some initial phases are skipped or performed in a less extensive manner. It might be the case that the only user of the language is the language designer himself, who is trying to eliminate some repetitious and tedious tasks by developing the new DSL. Is it worth performing complete domain analysis and language design in such case? While we strongly believe that good domain analysis and language design will pay off for even a medium-sized DSL community and DSL life span we do think that in the former case some other approaches might also be feasible. In this chapter we explore the applicability of grammatical inference [15] to infer DSL grammar from DSL programs. Such a scenario would be feasible when domain experts can provide complete DSL programs or excerpts of such programs. Strembeck and Zdun called it also "Mockup Language Driven DSL Development" [55]. This is also the case when domain notation is already established and the notation decision pattern [35] can be used. The results of grammatical inference, namely the inferred grammar, can be directly used to generate the DSL parser or be further examined by a software language engineer with the aim to further enhance the design of the language.

The structure of this chapter is as follows. Section 2 explores the relationship between domain analysis and DSL design. In Section 3, DSL design issues for DSLs are reviewed. GenInc, our previous grammatical inference approach for designing DSLs, and MAGIc, a newly developed memetic grammatical inference algorithm are presented in Section 4. Section 5 provides a case study in the domain of computer graphics whereby an existing DSL was designed by domain experts who were able to provide enough examples for MAGIc to infer a grammar for the DSL. Related work is described in Section 6 and we conclude in Section 7.

8.2 Analysis Phase of DSL Development

A DSL is a programming language dedicated to a particular domain and it provides appropriate built-in abstractions and notations [61] [35]. Hence, domain analysis should be performed with the aim to [13]:

- Select and define the domain, and
- Build the domain model (an explicit representation of the common and the variable properties of the system in a domain, the semantics of the properties and domain concepts, the dependencies between the properties).

Some typical domain analysis activities are analysis of similarity, analysis of variations, and analysis of combinations [11]. Despite that many domain analysis methods exist, such as:

- Feature-Oriented Domain Analysis (FODA) [29]
- Draco [39]
- Domain Analysis and Reuse Environment (DARE) [18]
- Family-Oriented Abstraction, Specification, and Translation (FAST) [65]
- Ontology-based Domain Engineering (ODE) [17]

they are rarely used in DSL development and domain analysis is usually done informally and in an incomplete manner. There is an urgent need in DSL research to identify reasons why this is so and possible solutions for improvement. The first observation might be that information gathered during domain analysis cannot be automatically used in the language design process. Another reason might be that complete domain analysis is too complex and outside of software engineers' capabilities. Regarding the first observation, information usually gathered during the domain analysis includes: terminology, concepts, and common and variable properties of concepts and their interdependencies. Although this is extremely useful information, further steps are not at all obvious. Only general advice can be given, such as [35]:

- *The list of variations indicate precisely what information is required to specify an instance of a system; this information must be directly specified in or be derivable from DSL programs,*
- *Commonalities should be built into DSL execution environment through a set of common operations and primitives (e.g., types) of the language.*

From domain analysis of a particular domain several possible DSLs can be developed, but all share important information found in domain analysis. As an example, Figure 8.1 [60] and Figure 8.2 [14] show two different DSLs for feature diagram description designed by different software engineers.

```
Car: all(carBody, Transmission, Engine, HorsePower, pullsTrailer?)
Transmission: one-of(automatic, manual)
Engine: more-of(electric, gasoline)
HorsePower: one-of(lowPower, mediumPower, highPower)
```

Fig. 8.1 First variant of feature description language.

```
car (carBody,
     transmission (automatic | manual),
     engine (electric + gasoline),
     horsePower (lowPower | mediumPower | highPower),
     trailer [ pullsTrailer])
```

Fig. 8.2 Second variant of feature description language.

The reader can observe that variation points have been expressed differently. In the case of deeply nested features, DSL programs in the second case will become less human readable, while in the first case this problem is nicely solved by binding [64]. Moreover, additional features can be built into the language in a manner that some properties can be more easily analyzed, checked and verified. For example, composite features in Figure 8.1 are written using capital letters. Such design decisions enable easier discovery of composite and atomic features. Additionally, the language can be extended with new symbols and keywords to enable easier parsing by both the programmer and the computer. Of course, this is a very simple example with the aim to show that after analysis of the same domain quite different DSLs can be designed. Let us conclude this section with the important open problem in DSL research: *Can we build tools which help us in this extremely creative, time consuming, and risky process?*

8.3 Design Phase of DSL Development

Designing a language (Figure 8.3 - adapted from [59]) involves defining the constructs in the language and giving the semantics [22] to the language, whether formal or informal. The semantics of the language describe the meaning of each construct in the language but also some fixed behavior that it is not specified by the program. Designing a language is highly creative, but not an easy task. As stated before, information gathered in the domain analysis should be used along with additional constraints (e.g., language level is high or low, capability to perform various analysis, readability, etc). Of particular importance for DSLs are the level of abstraction the language has and the degree to which it may be analyzed.

From previous experiences with programming language design [10] [23] [58] [67] [68] and from criteria for programming language evaluation [24], researchers developed criteria for good language design, such as readability, writeability, reliability, and cost. However, criteria for language design are often in contradiction (e.g., reliability vs. cost of execution, readability vs. writeability). Hence, language designers should find good trade-offs among these factors. To help language designers in this process several rules of thumb have been proposed, such as:

- Don't include untried ideas - consolidation, not innovation.
- Simplicity is really the key - avoid complexity. Too many solutions make the language hard to understand.
- Avoid requiring something to be stated more than once.
- Automate mechanical, tedious, or error-prone activities by providing higher level features.
- Regular rules, without exceptions, are easier to learn, use, describe, and implement.

Despite the immense experience in designing GPLs during the last fifty years [7] [10] [23] [58] [64] [67] we have noticed that often DSLs are badly designed. **Is DSL design radically different from GPL design?** Many believe that DSL design is not very much different from GPL design, while

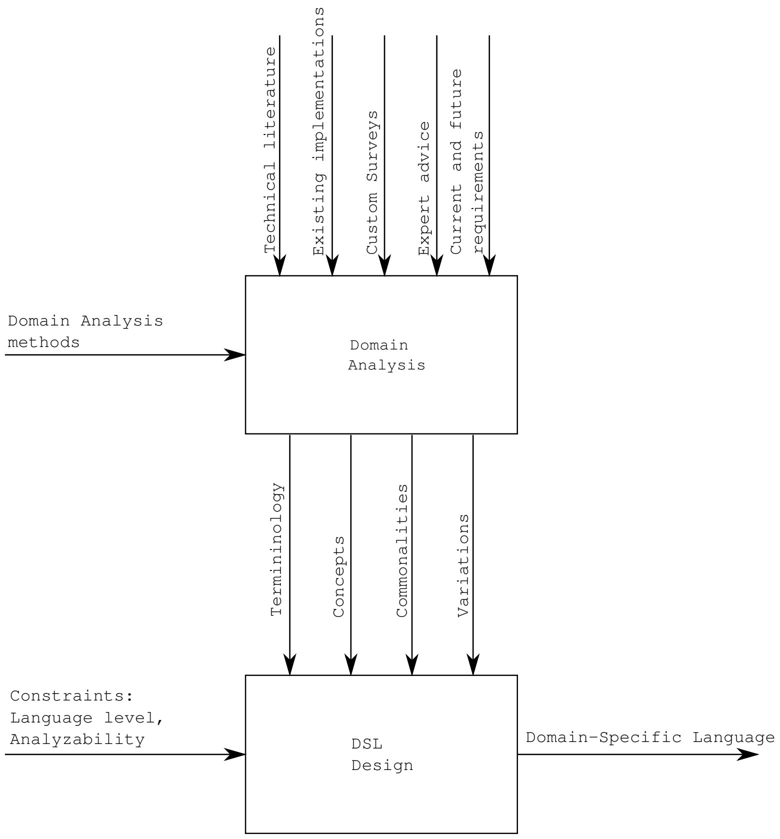


Fig. 8.3 Language design process.

others [54] [66] point to some important differences and the vast array of different choices [35]. Let us first explain the latter. Approaches to DSL design can be characterized along two orthogonal (independent) dimensions:

- The formal nature of the design description, and
- The relationship between the DSL and existing languages.

The first dimension is about informal versus formal language design. In an informal design the specification is usually in some form of natural language probably including a set of illustrative DSL programs, while a formal design would consist of a specification written using one of the available formal definition methods (e.g., regular expressions and grammars for syntax specifications, and attribute grammars, denotational semantics, operational semantics, and abstract state machines for semantic specification). The second dimension is about language exploitation versus language invention [35]. The easiest way to design a DSL is to base it on an already existing language (language exploitation), where

- Existing language is partially used (piggyback pattern),
- Existing language is restricted (language specialization pattern), and
- Existing language is extended (language extension pattern).

One possible benefit is familiarity for users and the other is easier DSL implementation. If there is no relationship between the DSL and an existing language, then a new language has to be invented. In such a case, the DSL should be designed according to the previously mentioned principles. But, the DSL designer has to keep in mind both the special character of DSLs as well as the fact that users need not be programmers. The latter fact is extremely important and might cause the admirable principles of orthogonality and simplicity to not necessarily be well-applied to DSL design. For example, instead of generalizing the language, the DSL has to be customized. GPLs have been always designed by highly professional programmers. The GPL design process was driven by their personal aesthetics and theoretical judgments resulting in a programming language they would like to use. The DSL for end-users should not be designed by such intuition since GPL designers are not end-users themselves. On the contrary, DSL design in this case should be driven by empirical studies, involvement of end-users, or by psychology of programming research [8]. Wile [66] reports on some interesting lessons learned while applying DSLs to end-users without computer science background. He observed several obstacles which come from technological, organizational and social perspectives. Some most interesting lessons are [66]:

- *Adopt whatever formal notations the domain experts already have, rather than invent new ones.*
- *You are almost never designing a programming language.*
- *Understand the organizational roles of the people who will be using your language.*
- *Be sure that the intended technology transfer process from your product into their organization's infrastructure is consistent with their business model.*
- *Find an advocate for your technology in their organization.*

- *Do not expect the domain experts to know what the computer can (should) do for them.*
- *Do not expect your users to overlook or forgive your design mistakes.*

While we believe that these lessons are valid for some particular domains, Wile's study [66] includes only three different DSLs. More experimental DSL research is needed to claim more general conclusions. Moreover, the principles valid for DSL design should be identified. One of the first attempts in this direction is presented in [32], which identified the following principles and requirements for DSLs:

- Conformity - how the DSL corresponds to domain concepts.
- Orthogonality - principle already known from GPL design.
- Supportability - how the DSL is supported by other indispensable language based tools (e.g., editors, debuggers, test engines).
- Integrability - how the DSL is integrated into other software engineering processes.
- Longevity - the DSL should be used long enough that the development of the DSL and accompanying tools pays off.
- Simplicity - principle already known from GPL design.
- Quality - how the DSL supports building quality and reliable software systems.

Again, let us conclude this section with the important open problem in DSL research: *Can we build tools which help us in DSL design?*

8.4 Grammatical Inference and Language Design

There are many open problems remaining, as shown in the previous two sections, in the design of DSLs, especially the lack of tool support for designing such languages. For very complex domains, domain experts are indispensable. Unfortunately domain experts are usually not language designers. Hence, we would like to explore the applicability of grammatical inference in providing tool support to achieve the task of designing DSLs for domain experts not proficient in language design. Domain experts can provide examples of DSL programs which can serve as input to the inference process. In the rest of this section we describe the grammatical inference algorithm MAGIc which endeavors towards automatic DSL design. MAGIc borrows some ideas from our previous work on GenInc [27]. Hence, GenInc is briefly described first in the next subsection.

8.4.1 *GenInc*

The authors have previously worked on GenInc [27], an unsupervised incremental context-free grammar (CFG) learning algorithm for facilitating DSL development for domain experts not well versed in programming language design. GenInc is based on the identification in the limit model of inductive inference [19] where (iteratively) an example of the target language is presented to the learner and after each sample the learner provides a representation of the language (as opposed to the active learning model [3] which assumes the existence of an oracle to which membership and grammar equivalence queries can be posed). GenInc learns from ordered positive samples and operates under the PACS learning paradigm (Probably Approximately Correct learning under Simple distributions) [63]. GenInc follows the incremental learning model as defined in [34]; it analyzes one training sample at a time, does not reprocess any previous samples when inferring a new hypothesis (CFG), and maintains only one candidate CFG in memory. GenInc makes use of positive samples only because often in practice negative samples are not available. These samples are characteristic samples (i.e., they exercise every rule of a grammar) and this property allows grammars to be inferred from positive samples only. To facilitate the grammar inference process, GenInc relies on an ordered positive sample presentation instead of an arbitrary presentation of samples. Smaller (in terms of number of tokens and number of language features) samples are presented before more complex ones; this allows an incremental learning algorithm to encode the sources of simple variances in the samples, before attempting to capture the long-range grammar rule dependencies. However, a different ordering of samples might result in a wrong grammar. Another limitation of GenInc is that difference among samples must be small; only one new feature per sample is allowed. More details on GenInc and various examples of successful DSL grammar inference are presented in [27].

8.4.2 *MAGIc*

A memetic algorithm [36] is a population-based evolutionary algorithm with local search. MAGIc, **M**emetic **A**lgorithm for **G**rammatical **I**nferen**c**e (Algorithm 1) infers CFGs from positive samples, which are divided into a learning set and a test set. The learning set is used only in local search, while grammar fitness is calculated on samples from the learning and test sets. Using samples from the test set in the grammar inference process is the main difference between our approach and many machine learning approaches, where the test set is used for testing the result accuracy. Although the initial population has been created mostly randomly in evolutionary algorithms such an approach has been proven insufficient for grammar inference [12]. Indeed, a more sophisticated approach is needed and an initial population of grammars is generated using the Sequitur algorithm [40], which generates a grammar that only parses a particular sample from learning set. Hence, Sequitur does not generalize productions. Moreover, the initialization procedure can be enhanced with seeds of partially correct grammars or grammar dialects, which are useful for learning grammar versions [16].

After the initial population is built, the evolutionary process of grammar learning is launched. It consists of the following main steps: local search, mutation, generalization, and selection. After each step, except selection, grammars are evaluated on all samples from the learning and test sets. The number of positive samples that are parsed is the grammar fitness [5], which is used in the selection process.

8.4.2.1 *Local Search*

Evolutionary algorithms usually perform better if they are augmented with local search. First, we explain what kind of local search is implemented in MAGIc. In a current population the grammar G_j is picked randomly and its associated sample S_i from the learning set. Note that in the initial population for every positive sample from the learning set a grammar exists which parses this sample. Next, the positive sample S_m from the learning set is picked randomly which are not parsed by the grammar G_j. The idea is to compare both positive samples S_i and S_m and based on the differences between them change the grammar G_j. Since changes in the grammar G_j can be done in many places several new grammars are injected into the population. Hence MAGIc is a kind of variable population size evolutionary algorithm. In doing this MAGIc exploits the LR(1) parsing history. When the parser fails to parse the sample, information about the parser stack and the LR(1) item set [2] are examined with the aim to change the grammar G_j in a way that both samples S_i and S_m are parsed by transformed grammar G_j. In addition to LR(1) configuration it is also important to know the

Algorithm 1 Memetic algorithm

```
**Method** Memetic_algorithm (learning set (LS), test set (TS), $p_m$, pop_size, num_gen,
{initial_grammars})
**begin**
  {Initialization}
  initialize population P
  **if** grammar versioning **then**
    add initial_grammars to P
  **else**
    **for all** positive samples $S_i$ in LS **do**
      grammar $G_i$ = SEQUITUR($S_i$)
      add $G_i$ to P
    **end for**
  **end if**
  current_G $\leftarrow$ 0
  **while** current_G $<$ num_gen **do**
    {Improve&Evaluate}
    **for** j = 1 ... pop_size **do**
      pick $G_j$ from P
      pick $S_i$ from LS **where** $S_i \in L(G_j)$
      pick $S_m$ from LS **where** $S_m \notin L(G_j)$
      $G_{pop\_size+1} \leftarrow$ LocalSearch($G_j$, diff($S_i$,$S_m$), LR(1)_config)
      ...
      $G_{pop\_size+\Delta j} \leftarrow$ LocalSearch($G_j$, diff($S_i$,$S_m$), LR(1)_config)
    **end for**
    **for** s = 1 ... $\Delta j$ **do**
      evaluate($G_{pop\_size+s}$, LS, TS)
      **if** fitness $> 0$ **then**
        add $G_{pop\_size+s}$ to P
      **end if**
    **end for**
    {Mutation}
    **for** j = 1 ... pop_size+$\Delta$ **do**
      **for** k = 1 ... size($G_j$) **do**
        **if** rnd $< p_m$ **then**
          case 1: $G_j^*$ = iteration$^+$($G_j$)
          case 2: $G_j^*$ = iteration$^*$($G_j$)
          case 3: $G_j^*$ = option($G_j$)
          evaluate($G_j^*$, LS, TS)
          **if** fitness $> 0$ **then**
            add $G_j^*$ to P
          **end if**
        **end if**
      **end for**
    **end for**
    {Generalization}
    **for** j = 1 ... pop_size+$\Delta$ **do**
      $G_{pop\_size+\Delta+j}$ = generalize($G_j$)
      evaluate($G_{pop\_size+\Delta+j}$, LS, TS)
      **if** fitness $> 0$ **then**
        add $G_{pop\_size+\Delta+j}$ to P
      **end if**
    **end for**
    {Selection}
    deterministically select best pop_size grammars
    current_G++
  **end while**
**end**
```

type of the difference among both samples S_i and S_m. Since we are using the Linux *diff* command [25] the differences can be:

- REPLACE (what to replace in the first sample to get the second one)
- DELETE (what to delete in the first sample to get the second one) or
- ADD (what to add in the first sample to get the second one).

Since comparison has to be done at the token level, not at the character level, we inserted the lexical analysis phase [2] before the *diff* command. Using data from the LR(1) configuration we can successfully change the grammar according to the following differences among two samples. Note that only the first two cases are explained, the last one is achieved in a similar, but more complicated, manner.

Case REPLACE:
When *diff* returns a replacement difference, the true positive sample is $\mathtt{s}_1\mathtt{s}_2\ldots\mathtt{s}_k\mathtt{a}_1\ldots\mathtt{a}_n\mathtt{s}_{k+1}\ldots\mathtt{s}_j$ and $\mathtt{s}_1\mathtt{s}_2\ldots\mathtt{s}_k\mathtt{b}_1\ldots\mathtt{b}_m\mathtt{s}_{k+1}\ldots\mathtt{s}_j$ is the false negative sample. In order to successfully parse the false negative sample, a production with the $\mathtt{b}_1\ldots\mathtt{b}_m$ part of the false negative has to be added into the grammar. This new production represents an alternative for part of the grammar, where the first symbol is $\mathtt{a}_1$ and the last symbol is $\mathtt{a}_n$. The problem here is to find the beginning and the end where to insert the alternative production. To solve this problem, we modified the LR(1) parser. When the true positive sample is parsed, we memorize for each token the dots in configuration items before and after this token. This way we can get all the configurations items where the dots are before $\mathtt{a}_1$ and all configurations items where the dots are after $\mathtt{a}_n$. The new grammar can be made when the dot before $\mathtt{a}_1$ and the dot after $\mathtt{a}_n$ are in the same production.

A case where the grammar can be changed is shown in production `Nx ::=` α_1 `<` β `>` α_2, where β represents part of grammar containing tokens $\mathtt{a}_1\ldots\mathtt{a}_n$. Note that α_1 and α_2 can be also ϵ. For easier understanding we introduce a symbol `<` that represents the dot before $\mathtt{a}_1$ and symbol `>` that represents the dot after $\mathtt{a}_n$. Both dots are in the same production, therefore β can be replaced with a new nonterminal and the following change to the grammar is made.

`Nx ::=` α_1 `N1` α_2
`N1 ::=` β

N1 ::= $b_1 \ldots b_m$

Case DELETE:
Strings $s_1s_2 \ldots s_k a_1 \ldots a_n s_{k+1} \ldots s_j$ and $s_1s_2 \ldots s_k s_{k+1} \ldots s_j$ represent true positive and false negative samples, respectively. In this case part of the true positive sample $a_1 \ldots a_n$ has to be found in the grammar and made optional. To find the part $a_1 \ldots a_n$ the configurations dots before a_1 and after a_n are used. This information is obtained from the output of parsing the true positive sample. The grammar can be changed if both configuration dots are in the same production. In case of production Nx ::= α_1 < β > α_2 the following change to the grammar is made.

Nx ::= α_1 N1 α_2
N1 ::= β
N1 ::= ϵ

8.4.2.2 *Mutation*

After local search grammars undergo transformation through mutation. Mutation exploits the knowledge of grammars, where grammar symbols often appear optionally or iteratively (option operator, iteration+ operator, and iteration* operator). Randomly chosen grammar symbols are mutated where the type of mutation is also random. The results of these operators on the production Nx ::= α_1 Ny α_2, where grammar symbol Ny is selected for mutation, are:

Option operator:

Nx ::= α_1 Nz α_2
Nz ::= Ny
Nz ::= ϵ

Iteration+ operator:

Nx ::= α_1 Nz α_2
Nz ::= Ny Nz
Nz ::= Ny

Iteration* operator:

Nx ::= α_1 Nz α_2
Nz ::= Ny Nz
Nz ::= ϵ

8.4.2.3 *Generalization*

The generalization step is an important step of MAGIc algorithm. In this part the generated grammars are checked if they can be generalized. The generalization procedure is composed of three steps. In the first step the algorithm checks if the RHS of particular production occurs in RHS of other productions. In this case the part of RHS that is the same as the RHS of the searched production is replaced with a nonterminal that represents the LHS of the searched production. For example, let the generated grammar have the following productions

```
Nx ::= α β
Ny ::= β
Ny ::= γ
```

The RHS of the second production β also occurs in the first production. Therefore it can be replaced with LHS of the second production. As the result of the first step the grammar is changed to:

```
Nx ::= α Ny
Ny ::= β
Ny ::= γ
```

Note, that generalization occurs in a manner that now both sentential form $\alpha\ \beta$ and $\alpha\ \gamma$ are parsed with the changed grammar.

In the second step grammars are examined and searched for repeating nonterminals. If two equal nonterminals occur consequently then there is a possibility to generalize the grammar in a manner that such nonterminal iterate. For example, let the generated grammar have the following productions:

```
Nx ::= α1 Nz Nz α2
Nz ::= β
Nz ::= γ
Nz ::= ϵ
```

In the first production there are two consequent nonterminals `Nz`. In this case one of them is removed and placed on tail of all the productions where `Nz` represents LHS (except the ϵ production):

```
Nx ::= α1 Nz α2
Nz ::= β Nz
Nz ::= γ Nz
Nz ::= ϵ
```

In the third step of generalization unit productions of form `Nx ::= Ny` are removed.

8.4.2.4 *Selection*

Deterministic selection is used where all grammars are ranked and only the best *pop_size* grammars are selected into the next population. It is planned in our future work to implement other selection schemes [5] and do thorough analysis of the impact of the selection scheme on MAGIc. However, the current deterministic selection performs well. Note that during the evolutionary cycle, the population of grammars is not fixed and can grow, but at the end of evolutionary cycle only *pop_size* grammars survive. The algorithm runs for *num_gen* generations, where *num_gen* is an input parameter of the algorithm. From our previous experience on evolutionary algorithms (EAs, e.g., [9]) we are aware of the fact that results of EAs heavily depend on a good parameter control mechanism (e.g., adaptive or self-adaptive parameter control). The impact of different control parameter mechanisms to MAGIc is also planned in the near future.

8.4.2.5 *Example*

To illustrate the approach let us look into the example for the simple domain specific language DESK [43], which is a simple desk calculation language. Its statements are of the form: `PRINT <expression> WHERE <definitions>`, where `<expression>` is an arithmetic expression over numbers and defined constants, and `<definitions>` is a sequence of constant definitions of the form `<constant name> = <number>`. By preparing input samples, the language designer needs to follow an important idea in grammatical inference, that positive samples have to include all language constructs and possibly also their legal combinations. We have prepared 12 different samples (Figure 8.4) on which MAGIc was tested using the control parameter tuning approach. The influence of control parameters (p_m, *pop_size*, *num_gen*) on successfulness of inferred grammars with respect to success rate (SR) and to average number of evaluations to solution (AES) was examined. Additionally, we measured also the structure of inferred grammars using several parameters (i.e., ANN - average number of nonterminals, ANP - average number of productions, ARHS - average size of production's right hand size) and tracked the average number of grammars found (ANG), the minimal number of samples (MNS) used whenever a successful grammar was inferred, and the average number of samples (ANS). The best results (Table 8.1) were achieved using $p_m = 0.01$. When $p_m = 0.02$ or $p_m = 0.05$, the inferred grammars did not successfully parse all positive samples in some runs, although this occurred mostly in 1 (or 2) runs out of 30 runs achieving SR=0.97 (or SR=0.93). On the other hand, small population size ($pop\_size = 20$ and $pop\_size = 30$) did not always lead to SR = 1.0, while bigger population size ($pop\_size = 50$) required many more evaluations. Hence, after this experiment we decided to use $pop\_size = 40$ and $p_m = 0.01$ for DESK grammar learning.

```
1.  print a
2.  print 5
3.  print b+2
4.  print a+b+c
5.  print c where d=54
6.  print 2 where i=4
7.  print 5+o where o=10
8.  print 15+30 where s=4
9.  print d where d=14;v=15
10. print 13 where d=17;e=42
11. print 75+4 where f=3;g=6
12. print a+b+c where a=1;b=9;c=11
```

Fig. 8.4 Samples of DESK language on which MAGIc was tested.

Our algorithm outputs as a result an array of inferred grammars that parse all samples from the learning and test sets. One of the grammar that MAGIc inferred based on the 12 generated samples is shown in Figure 8.5. This grammar was inferred using samples 7, 10, 12 and 1 . If we compare it with the original grammar of DESK language, which is shown in Figure 8.6, the inferred grammar has fewer productions and is equivalent to the original grammar .

From the results obtained we found that not all samples were needed to infer the successful grammar. Hence, we performed another small experiment. We have separated samples from Figure 8.4 into two parts. In the learning set samples 1, 7, 10 and 12 were chosen. Those samples were found, by running the MAGIc on 12 samples, to be the most important ones

Table 8.1 Parameter Tuning for DESK language (p_m - probability of mutation, PS - population size, NG - number of generations, SR - success rate, AES - average number of evaluations, MNS - minimum samples needed to infer grammar, ANS - average number of samples needed to infer grammar, ANN - average number of different nonterminals, ANP - average number of productions, ARHS - average size of RHS, ANG - average number of found new grammars per generation).

$\mathbf{p}_m$	**PS**	**NG**	**SR**	**AES**	**MNS**	**ANS**	**ANN**	**ANP**	**ARHS**	**ANG**
0.01	20	30	1.00	2287.10	4	7.42	7.49	14.19	3.30	11.30
0.01	20	50	1.00	2641.63	4	7.58	7.33	13.85	3.32	11.17
0.01	20	70	1.00	2885.33	4	8.20	8.05	15.25	3.31	12.23
0.01	30	30	1.00	3430.53	3	7.89	7.33	14.22	3.39	17.47
0.01	30	50	1.00	3741.50	4	7.92	7.75	14.89	3.38	16.93
0.01	30	70	1.00	4370.23	4	7.85	7.48	14.22	3.31	16.07
0.01	**40**	**30**	**1.00**	**4206.27**	**3**	**7.56**	**7.45**	**14.07**	**3.32**	**19.90**
0.01	**40**	**50**	**1.00**	**4925.00**	**3**	**7.82**	**7.35**	**13.91**	**3.34**	**20.77**
0.01	40	70	1.00	5501.10	4	7.32	7.22	13.67	3.34	23.23
0.01	50	30	1.00	5100.33	3	7.64	7.44	14.32	3.38	27.67
0.01	50	50	1.00	5895.97	3	7.80	7.51	14.26	3.31	27.00
0.01	50	70	1.00	6670.33	4	7.42	7.54	14.29	3.32	24.57
0.02	20	30	0.97	2808.52	4	7.74	7.68	14.72	3.39	10.86
0.02	20	50	1.00	3278.70	3	7.56	7.56	14.41	3.26	11.30
0.02	20	70	1.00	4099.23	4	7.33	7.32	13.71	3.30	9.57
0.02	30	30	0.93	4137.64	3	7.83	7.71	14.64	3.40	14.71
0.02	30	50	0.97	5012.52	4	7.22	7.44	14.02	3.26	15.14
0.02	30	70	1.00	5832.73	3	7.42	7.44	14.09	3.38	15.27
0.02	40	30	1.00	5041.03	3	7.47	7.42	14.11	3.32	21.73
0.02	40	50	1.00	6212.63	4	7.33	7.53	14.38	3.32	20.60
0.02	40	70	1.00	7710.50	4	7.55	7.40	14.00	3.31	19.77
0.02	50	30	1.00	6115.97	4	7.69	7.58	14.65	3.37	27.17
0.02	50	50	1.00	7629.00	3	7.40	7.64	14.43	3.24	24.73
0.02	50	70	1.00	8883.63	3	7.39	7.40	14.02	3.31	24.53
0.05	20	30	0.97	4247.38	3	7.41	8.03	15.14	3.18	10.97
0.05	20	50	1.00	5622.57	3	7.11	7.58	14.41	3.27	8.80
0.05	20	70	1.00	8505.93	4	7.78	8.13	15.39	3.22	10.37
0.05	30	30	1.00	5925.93	3	7.11	7.57	14.22	3.25	14.67
0.05	30	50	0.97	8655.86	3	7.03	7.94	14.77	3.15	14.48
0.05	30	70	1.00	10930.07	3	6.98	7.68	14.55	3.23	14.47
0.05	40	30	1.00	8634.67	3	6.78	7.70	14.33	3.18	17.73
0.05	40	50	1.00	11495.43	3	7.17	8.06	15.10	3.18	18.90
0.05	40	70	1.00	15367.87	3	7.07	7.69	14.46	3.21	19.30
0.05	50	30	1.00	9431.63	4	7.23	7.99	15.12	3.20	26.90
0.05	50	50	1.00	14697.77	3	7.18	8.16	15.39	3.22	22.33
0.05	50	70	1.00	18976.70	4	7.11	7.82	14.77	3.21	22.87

by inferring complete grammars. The test set contained the remaining 8 samples. The algorithm was run for 30 times and the results were as follows: SR=0.83, AES=14554, MNS=4, ANS=4, ANN=5.61, ANP=10.21,

```
1. N1 ::= print N3 N5
2. N2 ::= + N3
3. N2 ::= ϵ
4. N3 ::= number N2
5. N3 ::= id N2
6. N4 ::= ; id = number N4
7. N4 ::= ϵ
8. N5 ::= where id = number N4
9. N5 ::= ϵ
```

Fig. 8.5 Inferred grammar with MAGIc for DESK language.

```
1. DESK ::= print E C
2. E ::= E + F
3. E ::= F
4. F ::= id
5. F ::= number
6. C ::= where Ds
7. C ::= ϵ
8. Ds ::= D
9. Ds ::= Ds ; D
10.D ::= id = number
```

Fig. 8.6 Original grammar of DESK language [43].

and ANG=9.84. The success rate was lower than running the algorithm on all 12 samples in the learning set. The reason is obvious since grammars which parse all samples from the learning set, but not all from the test set, cannot improve anymore by local search. But still, in more than 80% of the cases it was enough to have only 4 samples to infer the successful grammar (Figure 8.5).

MAGIc can also be used to infer grammars for language dialects. In this case, the initial grammar is not generated from input sample but it is read from a file. The samples given in the learning set contain the missing concepts that the user wants to add to the current language. We have tested MAGIc on DESK language dialects. Two different tests were done with two different initial grammars (Figures 8.7 and 8.8). In the first grammar we have removed the `<definitions>` part and in the second grammar the

`<expression>` part. In both cases we have used the samples from Figure 8.4. The results are shown in Figures 8.9 and 8.10. In both cases MAGIc successfully inferred new grammars that parse all the input samples.

```
1. N1 ::= print N2
2. N2 ::= N2 + N3
3. N2 ::= N3
4. N3 ::= id
5. N3 ::= number
```

Fig. 8.7 Initial DESK grammar without `<definitions>` part.

```
1. N1 ::= print id N2
2. N2 ::= where N3
3. N2 ::= ϵ
4. N3 ::= N4
5. N3 ::= N3 ; N4
6. N4 ::= id = number
```

Fig. 8.8 Initial DESK grammar without `<expression>` part.

```
1.  N1 ::= print N2 N4
2.  N2 ::= N2 + N3
3.  N2 ::= N3
4.  N3 ::= id
5.  N3 ::= number
6.  N4 ::= where N3 = N3 N5
7.  N4 ::= ϵ
8.  N5 ::= ; N3 = N3 N5
9.  N5 ::= ϵ
```

Fig. 8.9 Grammar with inferred definitions part.

We have tested MAGIc also on other languages like FDL, WHILE, and on the example presented in Section 5. MAGIc is able to infer CFGs, which are non-ambiguous and of type LR(1).

To make MAGIc successful in inferring a grammar from positive sam-

```
1.  N1 ::= print N5 N6 N2
2.  N2 ::= ϵ where N3
3.  N2 ::= ϵ
4.  N3 ::= N4
5.  N3 ::= N3 ; N4
6.  N4 ::= N5 = N5
7.  N5 ::= id
8.  N5 ::= number
9.  N6 ::= + N5 N6
10. N6 ::= ϵ
```

Fig. 8.10 Grammar with inferred expression part.

ples a domain expert should try to cover all possible language constructs and their valid combinations in the provided positive samples. But, can the number of positive samples that MAGIc needs as input to infer the correct grammar be calculated? First, we should be aware that grammar, albeit using only a finite set of productions, can describe a language with an infinite number of different statements. However, a key observation here is that we need to figure out which productions can generate these statements. Therefore, we do not need all possible statements, which are often infinite, but only those which are generated through all possible combinations of the production rules in the grammar. If the grammar includes recursive rules, then these rules need to be exercised only once because multiple runs through the rules will not append any new construct. Taking this into account we can calculate exactly how many different samples need to be generated to exhibit all possible combination of productions. The calculation is similar to the variability calculation of feature diagrams [13], with additional rules to handle recursion. The number of all possible different statements is calculated by the following rules, where `A` stands for non-terminals, `a` is a terminal symbol, β denotes a sequence of nonterminal and terminal symbols, and `Var` stands for Variability:

```
Var (A ::= A β) = Var(β) + 1 if A ::= ε also exists
Var (A ::= β A) = Var(β) + 1 if A ::= ε also exists
Var (A ::= A β) = Var(β) if A ::= ε does not exist
Var (A ::= β A) = Var(β) if A ::= ε does not exist
Var (A ::= β1...βn) = Var(β1) *...* Var(βn)
Var (A ::= β1 |...| βn) = Var(β1) +...+ Var(βn)
Var (A ::= a) = 1
Var (A ::= ε) = 1
```

Let us compute variability for the DESK grammar (Figure 8.6):

```
Var(DESK) = Var(print) * Var(E) * Var(C)
          = 1 * 4 * 3 = 12
Var(E) = Var(E + F) + Var(F) = Var(+ F) + Var(F)
       = Var(+) * Var(F) + Var(F)
       = 1 * 2 + 2 = 4
Var(F) = Var(id) + Var(num)
       = 1 + 1 = 2
Var(C) = Var(where) * Var(Ds) + Var(ε)
       = 1 * 2 + 1 = 3
Var(Ds) = Var(D) + Var(Ds ; D) = Var(D) + Var(; D)
        = Var(D) + Var(;) * Var(D)
        = 1 + 1 * 1 = 2
Var(D) = Var(id) * Var(=) * Var(num)
       = 1 * 1 * 1 = 1
```

According to the above rules, variability of the DESK grammar is 12, which means that with 12 positive samples we can exercise all possible combination of grammar productions. Hence, if we know the original grammar we can estimate the number of positive samples needed for grammatical inference. We have run MAGIc on 12 positive samples using different control parameters and tracked the minimal number of samples (MNS) used whenever a successful grammar was inferred and the average number of samples (ANS). From Table 8.1 we can see that in some cases only from 3 positive samples (e.g., from samples No. 5, 10, 8) the correct grammar was inferred, while the average number across all 30 runs was 7.45 samples, which is considerably less than Var(DESK). Grammar variability can become a good measure of how good and fast different grammatical inference algorithms are since the number of true positive samples have a big impact on algorithm performance. From Algorithm 1 we can conclude that the complexity of the algorithm is polynomial $\approx O(N^3)$. The average processing time for 12 DESK language samples with $p_m = 0.01$, $pop\_size = 40$ and $num\_gen = 30$ was about 30 seconds. For experimental runs, we have used an Intel Core 2 Duo P8600 processor with 2.4 GHz. On the other hand, MAGIc can infer the grammar G with a number of samples less than Var(G) since MAGIc consists also of mutation and generalization steps (see

Algorithm 1). Note also that, for example from sample 11, which contains 12 terminal symbols, we can construct $8.76478739164 \times 10^{11}$ possible binary derivation trees [12]. This is just one example of how big the search space can be even for small sentences.

In the realistic scenario the variability of yet to discovered grammar is not known. However, we can still do approximation and calculate variability of the inferred grammar. If it is much bigger than the current number of positive samples then current samples might not be sufficient and the domain expert should provide additional positive samples.

8.4.2.6 *Limitations*

In the current state MAGIc has also some limitations which are planned to be removed in the future. The first limitation is bounded to the *diff* output. In some cases the *diff* command does not return the difference needed to infer some language features, like recursion. For example in FDL language, if the true positive and false negative samples are:

```
c:keyword(atomicFeature,atomicFeature)
c:keyword(keyword(atomicFeature,atomicFeature),atomicFeature)
```

the *diff* command returns:

```
4a5,6
> keyword
> (
8a11,13
> ,
> atomicFeature
> )
```

which means add tokens "`keyword (`" (on position 5-6) and "`, atomicFeature )`" (on position 11-13) from false negative sample into true positive sample after the fourth and eight tokens, respectively. The change to the grammar that parses the true positive sample:

```
G: N1 ::= c : keyword ( atomicFeature , atomicFeature )
```

is:

```
G¹: N1 ::= c : keyword ( N2 atomicFeature , atomicFeature ) N3
    N2 ::= keyword (
    N2 ::= ϵ
    N3 ::= , atomicFeature )
    N3 ::= ϵ
```

From grammar G^1 it is difficult or almost impossible to see the recursive structure and change the grammar in a way to parse the recursive structures found in FDL language.

The required difference between the above samples, to express the recursive feature of the language, would be:

```
5c5,10
< atomicFeature
---
> keyword
> (
> atomicFeature
> ,
> atomicFeature
> )
```

Replace the fifth token `atomicFeature` in true positive sample with tokens `keyword(atomicFeature,atomicFeature)` (on positions 5 - 10) from the false negative sample. The change to the grammar G in this case would be:

```
G²: N1 ::= c : keyword ( N2 , atomicFeature )
    N2 ::= atomicFeature
    N2 ::= keyword ( atomicFeature , atomicFeature )
```

After the generalization step described in subsection 8.4.2.3, the grammar G^2 would be generalized to the next form:

```
G²′: N1 ::= c : N2
     N2 ::= atomicFeature
     N2 ::= keyword ( N2 , N2 )
```

Grammar $G^{2'}$ is correct and easier to read than grammar G^1. A more detailed look at grammar G^1 also shows that it can parse also the negative samples, for example:

`c:keyword(atomicFeature,atomicFeature),atomicFeature)` or
`c:keyword(keyword(atomicFeature,atomicFeature)`

Aware of this MAGIc limitation we are working on our own "*diff*" algorithm, that is more suitable for our use. Another limitation of the MAGIc algorithm is lack of negative samples and therefore some of the inferred grammars can be overgeneralized. To avoid overgeneralization, negative samples will also be used in the future. A less serious current limitation of MAGIc is that final result can produce more grammars which all parse the positive samples. In the case that a large set of grammars results, a language designer will face a tedious problem to finally select the most suitable one. To eliminate this problem we are investigating the use of the MDL principle [47] which prefers smaller grammars over bigger ones.

8.5 Case Study

The capability of using grammatical inference techniques in DSL design, in particular the MAGIc algorithm, have been tested on a real example from the computer graphics domain. In [56], Strnad and Guid developed a method for modeling trees with hypertextures, a method for describing 3-D shapes and textures [45]. The method is based on a volumetric representation of trees generated by three dimensional variation of an Iterated Function System (IFS), which is a technique for fractal generation. Using this method a tree is a fractal object described by nonlinear and nondeterministic IFS where a combination of linear transformations (scalings, translations and rotations) and nonlinear shears are used. Scaling, translations, and rotations are used to describe fractal subparts (size, position, and orientation), while nonlinear shears are used to bend them in two coordinate directions. Nondeterminism of transformations is achieved by randomly chosen values of transformation parameters (e.g., angle of rotation). Random parameters allow the same set of transformations to produce visually different hypertrees with similar basic structure. A hypertree consists of branches which are like smaller trees. This similarity goes several levels deep and depends on the tree family. Branches need not be exact copies of the whole tree, but they may only closely resemble it. Branching structure is generated by condensation transformation. Moreover, the ideal structure of hypertrees can be distorted with noise perturbation. The whole description of a hypertree consists of resolution, number of iterations, fractal depth, number of branch levels, the description of the generator (POINTINIT or LINEINIT), the coloring scheme (DEPTHCOLOR) and transformations. In Figure 8.11 an excerpt from a DSL program is given, while in Figure 8.12 two generated hypertrees are displayed.

Strnad and Guid [56] have no experience with language engineering and

```
RESOLUTION 300 400 300
ITERATIONS 3000000
POINTINIT 0 0 0
TREEDEPTH 5
BRANCHDEPTH 1
HYPERVOLUME -0.6 0.6 -1 0.6 -0.6 0.6
DEPTHCOLOR 0-1 0.7+/-0.0 0.7+/-0.0 0.5+/-0.0
DEPTHCOLOR 2-5 0.25+/-0.25 0.75+/-0.25 0.25+/-0.25
TRANSFORM 1 0
TRANSLATE (0,0,0) (1,1,1) (0,0,0)
SHEAR (0,0,0) (0.5,0.5,0.5) (2,2,2)
SHEAR_XZ SCALE (0.3,0.3,0.3) (0.4,0.4,0.4) (0.3,0.3,0.3)
ROTATE (-80,-80,-80) (0,0,0) (0,0,0)
ROTATE (0,0,0) (45,45,45) (0,0,0)
TRANSLATE (0,0,0) (-0.72,-0.72,-0.72) (0,0,0)
TRANSFORM 1 0
TRANSLATE (0,0,0) (1,1,1) (0,0,0)
SCALE (0.6,0.6,0.6) (0.6,0.6,0.6) (0.6,0.6,0.6)
ROTATE (0,0,0) (50,50,50) (0,0,0)
TRANSLATE (0,0,0) (-0.4,-0.4,-0.4) (0,0,0)
TRANSFORM 1 0
TRANSLATE (0,0,0) (1,1,1) (0,0,0)
SCALE (0.8,0.8,0.8) (0.8,0.8,0.8) (0.8,0.8,0.8)
ROTATE (0,0,0) (150,150,150) (0,0,0)
TRANSLATE (0,0,0) (-0.8,-0.8,-0.8) (0,0,0)
CONDENSATION 1 CONE -1.0 0.5 0.02 0.0 CONE_Y
```

Fig. 8.11 Excerpt of domain specific program for hypertree generation.

they implemented the language from scratch [57]. The parsing algorithm was hard coded and not based on the grammar. Because of the lack of the knowledge in language engineering the underlying grammar was not even identified. If it were, the parsing code could be automatically generated by parser generators. On the other hand, the maintenance of parsing code is now extremely difficult in the event that this language will evolve in the future.

The purpose of this section is to show that current advances in grammatical inference are now able to infer grammars for simple DSLs, such as

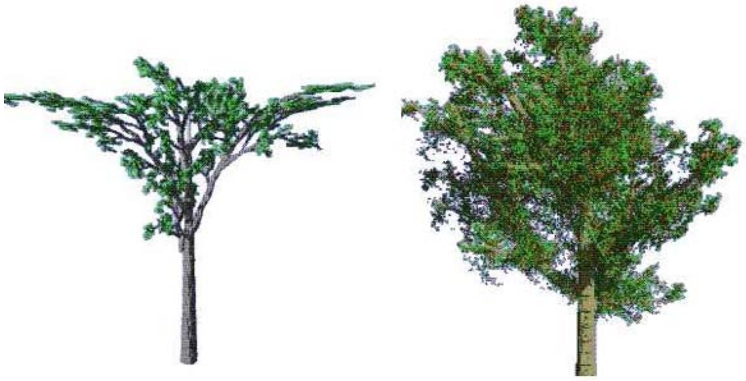


Fig. 8.12 Two samples of generated trees.

presented in Strnad and Guid [56]. We have run the MAGIc algorithm on a sample of programs provided by Strnad and Guid [56]. A simple step in the inferring of the final grammar is presented in Figures 8.13 - 8.17, where current samples S_i and S_m, the Sequitur initial grammar for samples S_i, the current grammar G_j and the newly inferred grammar are presented. Note that the difference between sample S_i and S_m is in the construct *pointinit* which is replaced with construct *lineinit*, and in possible iteration of construct *depthcolor*. Hence, one of the GenInc limitations is now eliminated in MAGIc. After 30 runs (note that MAGIc is a stochastic algorithm) the following measures on MAGIc were collected: $SR = 1.0$ and AES = 8071, while the inferred grammars have the following characteristics: ANN = 10.6, ANP = 21.2, and ARHS=6.4.

```
RESOLUTION 100 150 200
ITERATIONS 10000
POINTINIT 1 0 3
TREEDEPTH 4
BRANCHDEPTH 2
HYPERVOLUME 4 3 3 4 4 5
DEPTHCOLOR 0-1 0.6+/-0.3 0.2+/-0.1 0.5+/-0.2
CONDENSATION 1 CONE 2 2 1 1 CONE_Y
```

Fig. 8.13 Sample S_i.

```
N1 ::= resolution N2
        iterations num
        pointinit N2
        treedepth num
        branchdepth num
        hypervolume N2 N2
        depthcolor range bpp bpp bpp
        condensation num cone
        N2 num coney
N2 ::= num num num
```

Fig. 8.14 Sequitur initial grammar for sample S_i.

```
N1 ::= resolution N2
        iterations num
        pointinit N2
        treedepth num
        branchdepth num
        hypervolume N2 N2 N3
        condensation num cone
        N2 num coney
N2 ::= num num num
N3 ::= depthcolor range
        bpp bpp bpp
N3 ::= ϵ
```

Fig. 8.15 Grammar G_j.

The final inferred grammar found in generation 15 is shown in Figure 8.18.

8.6 Related Work

Early grammar inference research primarily focused on regular grammars and resulted in successful algorithms and learning systems such as L* [4], regular positive negative inference [42], finite automata learning from simple examples [44] and automata learning with merge constraints [33] which

```
RESOLUTION 100 200 100
ITERATIONS 30000
LINEINIT 0.3 0.3 0.3 0.5 0.5 0.5 0.5
TREEDEPTH 6
BRANCHDEPTH 1
HYPERVOLUME 5 5 5 6 6 6
DEPTHCOLOR 0-2 0.4+/-0.1 0.6+/-0.2 0.4+/-0.1
DEPTHCOLOR 0-5 0.2+/-0.1 0.4+/-0.1 0.5+/-0.1
CONDENSATION 1 CONE 1 1 1 1 CONE_Y
```

Fig. 8.16 Sample S_m.

```
N1 ::= resolution N2
        iterations num N4 N2
        treedepth num
        branchdepth num
        hypervolume N2 N2 N3
        condensation num cone
        N2 num coney
N2 ::= num num num
N3 ::= depthcolor range bpp
        bpp bpp N3
N3 ::= ϵ
N4 ::= pointinit
N4 ::= lineinit num num num num
```

Fig. 8.17 Inferred grammar $G_{j+\Delta 1}$.

made advances in theory of learning and were applied to practical problems. While preliminary (mainly theoretical) results concerning CFG learning were negative, recent interest in potential applications of CFG learning in diverse domains has resulted in renewed efforts at CFG learning. Many grammatical inference efforts are focused on natural language learning. EMILE [1] falls under the PACS learning paradigm and uses clustering characteristic expressions and contexts to infer natural language grammar from a corpus. The Alignment-based learning (ABL) [62] algorithm uses the principle of substitutability of constituents to compare all the sentences in a corpus in order to learn a grammar. Each sample is then compared

```
N1 ::= resolution N2
        iterations num N3 N2
        treedepth num
        branchdepth num
        hypervolume N2 N2
        condensation num cone N2 num coney
N2 ::= num num num N4
N3 ::= pointinit
N3 ::= lineinit num num num num
N4 ::= depthcolor range bpp bpp bpp N4
N4 ::= ϵ
N4 ::= name progname N4
N4 ::= scale lpar num comma num comma num rpar
        lpar num comma num comma num rpar
        lpar num comma num comma num rpar N4
N4 ::= rotate lpar num comma num comma num rpar
        lpar num comma num comma num rpar
        lpar num comma num comma num rpar N4
N4 ::= translate lpar num comma num comma num rpar
        lpar num comma num comma num rpar
        lpar num comma num comma num rpar N4
N4 ::= transform num num N4
N4 ::= shear lpar num comma num comma num rpar
        lpar num comma num comma num rpar
        lpar num comma num comma num rpar shearxz N4
N4 ::= perturb lpar num comma num comma
        num comma num rpar
        lpar num comma num comma num comma num rpar
        lpar num comma num comma num comma num rpar
        lpar num comma num comma num comma num rpar N4
```

Fig. 8.18 Inferred grammar.

to every other sample during a single learning phase and the differences are noted. ADIOS [53] uses a statistical method to extract hierarchical structure and learns a complete syntax from a language corpus as well as generates grammatically novel sentences in an unsupervised manner. Klein and Manning [31] describe an approach to infer natural language using unsupervised learning together with a constituent-context model. They infer a hierarchical syntactic structure in the form of a distributional model of constituents for the underlying natural language from a few thousand training sentences. The main difference between these research efforts and MAGIc is their focus on learning natural language; consequently most of these algorithms assume existence of certain syntactic categories or word classes (part-of-speech tags like nouns and verbs).

Recently there has been a growing set of grammatical inference research endeavors focused on programming languages and software engineering applications. Synapse, an incremental inductive CYK algorithm which learns simple CFGs from positive and negative examples, is discussed in [38]. Improvements to the Synapse system in the form of a process called bridging (to generate rules to bridge an incomplete parse tree) and serial and global search methodologies to find the minimum set of rules are described in [37]. The latest improvement to Synapse [26] utilizes source code and corresponding object codes described using an inductive logic programming notation to allow inference of unambiguous grammars and basic semantics. Compared to Synapse and its various extensions MAGIc only uses positive examples as input, a more extreme scenario of learning. Synapse also expects an ordered presentation of both positive and negative samples for efficient learning while MAGIc is not sensitive to order effects in input. A technique for inferring grammar rules for programming language dialects is discussed in [16]. While MAGIc focuses on inferring complete grammars of DSLs, this technique uses a set of samples of the dialect and the grammar of the standard language to infer rules specific to the dialect but missing in the standard language. Preliminary work in using evolutionary algorithms enhanced with local search is presented in [48] and tested on Tomita's simple regular languages. A system which utilizes a CYK-like tabular representation method along with partially structured samples (to indicate the shape of the derivation tree) and a genetic search technique for partitioning the set of nonterminals is discussed in [49]. This system assumes the existence of partially structured samples as well as both positive and negative samples. In [28] local search is used for better learning rule probabilities of stochastic CFG. A Genetic Algorithm (GA) with a priori distribution over the space of all possible grammars with a bias towards simpler grammars is described in [30]. This technique assumes the existence of a covering grammar which is a stochastic CFG (i.e, a CFG where each rule has an associated probability in the range [0,1]). The GA optimizes the stochastic CFGs' parameters based on language samples.

eg-GRIDS [46] uses several operators suited to construction of CFGs (such as optional omission of a nonterminal, and attempt to detect patterns like a^ib^i). MAGIc uses similar operators, and in addition uses "local search" that examines differences in parsing of samples, one successful and one unsuccessful.

8.7 Conclusion

This chapter has discussed the design phase of DSL development, which should take as an input results from domain analysis, DSLs design principles and other constraints such as development costs, DSL users, DSL life span, etc. In particular, we are interested in tools which can assist us in this difficult task. Domain analysis and the language design phase can be reduced to some degree or eliminated altogether if the new DSL is based on an already existing language. To further limit general DSL design we focused on the case when notation, at least in part, can be already provided by the domain expert. Hence, our approach assumes that some DSL programs (or excerpts) already exist. These are then input to our grammatical inference tool - MAGIc, which is able to infer the underlying context-free grammar. In this chapter we have shown that grammatical inference may be used to assist a domain expert and software language engineer in developing DSLs. However, the inferred grammars should be further analyzed, refactored, and improved by software language engineers. Hence, the proposed approach and tool can be of great help to software language engineers or domain experts to more easily discover the main language constructs and their relationships. Moreover, software language engineers can then spend more time on soft language design principles which are very hard to automate. The approach has been validated using a real DSL designed for expressing hypertrees in computer graphics.

Acknowledgments

We would like to thank Damjan Strnad and Niko Guid for providing us examples for hypertree generation. This work was supported in part by United States National Science Foundation award CCF-0811630.

References

1. Adriaans, P. W., Trautwein, M. and Vervoort, M. (2000). Towards high speed grammar induction on large text corpora, in *SOFSEM '00: Proceedings of the 27th Conference on Current Trends in Theory and Practice of Informatics* (Springer-Verlag, London, UK), ISBN 3-540-41348-0, pp. 173–186.
2. Aho, A. V., Lam, M. S., Sethi, R. and Ullman, J. D. (2006). *Compilers: Principles, Techniques, and Tools (2nd Edition)* (Addison Wesley), ISBN 0321486811.
3. Angluin, D. (1981). A note on the number of queries needed to identify regular languages, *Information and Control* **51**, 1, pp. 76–87.
4. Angluin, D. (1987). Learning regular sets from queries and counterexamples, *Information and Computation* **75**, 2, pp. 87–106.
5. Bäck, T., Fogel, D. and Michalewicz, Z. (1997). *Handbook of evolutionary computation* (Oxford University Press).
6. Bentley, J. (1986). Programming pearls: Little languages, *Communications of the ACM* **29**, 8, pp. 711–721.
7. Bergin, T. J., Jr. and Gibson, R. G., Jr. (eds.) (1996). *History of programming languages—II* (ACM, New York, NY, USA), ISBN 0-201-89502-1.
8. Blackwell, A. F. (2006). Psychological Issues in End-User Programming, in H. Lieberman, F. Paternò and V. Wulf (eds.), *End User Development* (Springer), pp. 9–30.
9. Brest, J., Greiner, S., Bošković, B., Mernik, M. and Žumer, V. (2006). Self-Adapting Control Parameters in Differential Evolution: A Comparative Study on Numerical Benchmark Problems, *IEEE Transactions on Evolutionary Computation* **10**, 6, pp. 646–657.
10. Brooks, F. P. (1996). Language design as design, in T. J. Bergin, Jr. and R. G. Gibson, Jr. (eds.), *History of Programming Languages—II* (ACM, New York, NY, USA), pp. 4–15.
11. Coplien, J. O., Hoffman, D. and Weiss, D. M. (1998). Commonality and variability in software engineering, *IEEE Software* **15**, 6, pp. 37–45.
12. Črepinšek, M., Mernik, M., Javed, F., Bryant, B. R. and Sprague, A. (2005). Extracting grammar from programs: Evolutionary approach, *ACM Sigplan Notices* **40**, 4, pp. 39–46.
13. Czarnecki, K. and Eisenecker, U. (2000). *Generative Programming: Methods, Tools, and Applications* (Addison-Wesley), ISBN 0-201-30977-7.
14. Dawes, B. (2009). Feature model diagrams in text and html, URL `http://www.boost.org/community/feature\_model\_diagrams.html`.
15. de la Higuera, C. (2000). Current trends in grammatical inference, in F. J. Ferri *et al.* (eds.), *Advances in Pattern Recognition, Joint IAPR International Workshops SSPR+SPR'2000*, *Lecture Notes in Computer Science*, Vol. 1876 (Springer), pp. 28–31.
16. Dubey, A., Jalote, P. and Aggarwal, S. K. (2008). Learning context-free grammar rules from a set of program, *IET Software* **2**, 3, pp. 223–240.
17. Falbo, R. d. A., Guizzardi, G. and Duarte, K. C. (2002). An ontological

approach to domain engineering, in *SEKE '02: Proceedings of the 14th international conference on Software engineering and knowledge engineering* (ACM, New York, NY, USA), ISBN 1-58113-556-4, pp. 351–358.

18. Frakes, W., Prieto-Diaz, R. and Fox, C. (1998). DARE: Domain analysis and reuse environment, *Annals of Software Engineering* **5**, pp. 125–141.
19. Gold, E. M. (1967). Language identification in the limit, *Information and Control* **10**, 5, pp. 447–474.
20. Gray, J., Tolvanen, J.-P., Kelly, S., Gokhale, A., Neema, S. and Sprinkle, J. (2007). Domain-specific modeling, in P. A. Fishwick (ed.), *CRC Handbook of Dynamic System Modeling*, chap. 7 (CRC Press), pp. 1–20.
21. Greenfield, J., Short, K., Cook, S. and Kent, S. (2004). *Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools* (Wiley), ISBN 0471202843.
22. Harel, D. and Rumpe, B. (2004). Meaningful modeling: What's the semantics of "semantics"? *IEEE Computer* **37**, 10, pp. 64–72.
23. Hoare, C. A. R. (1973). Hints on programming language design. Tech. Rep. CS-TR-73-403, Stanford University, Stanford, CA, USA.
24. Horowitz, E. (1984). *Fundamentals of programming languages*, 2nd edn. (Computer Science Press, Inc., New York, NY, USA), ISBN 0-88175-004-2.
25. Hunt, J. W. and McIlroy, M. D. (1976). An algorithm for differential file comparison, Tech. Rep. 41, Bell Laboratories, Murray Hill, NJ.
26. Imada, K. and Nakamura, K. (2008). Towards machine learning of grammars and compilers of programming languages, in *ECML PKDD '08: Proceedings of the European conference on Machine Learning and Knowledge Discovery in Databases - Part II* (Springer-Verlag, Berlin, Heidelberg), ISBN 978-3-540-87480-5, pp. 98–112.
27. Javed, F., Mernik, M., Bryant, B. R. and Sprague, A. (2008). An unsupervised incremental learning algorithm for domain-specific language development, *Applied Artificial Intelligence* **22**, 7, pp. 707–729.
28. Kammeyer, T. E. and Belew, R. K. (1996). Stochastic context-free grammar induction with a genetic algorithm using local search, in R. K. Belew (ed.), *Foundations of Genetic Algorithms IV* (Morgan Kaufmann), pp. 3–5.
29. Kang, K., Cohen, S., Hess, J., Novak, W. and Peterson, S. (1990). Feature-Oriented Domain Analysis (FODA) Feasibility Study, Tech. Rep. CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University.
30. Keller, B. and Lutz, R. (2005). Evolutionary induction of stochastic context free grammars, *Pattern Recognition* **38**, 9, pp. 1393–1406.
31. Klein, D. and Manning, C. D. (2005). Natural language grammar induction with a generative constituent-context model, *Pattern Recognition* **38**, 9, pp. 1407–1419.
32. Kolovos, D. S., Paige, R. F., Kelly, T. and Polack, F. A. C. (2006). Requirements for domain-specific languages, in *Proceedings of the First ECOOP Workshop on Domain-Specific Program Development, co-located with ECOOP06.*

33. Lambeau, B., Damas, C. and Dupont, P. (2008). State-merging DFA induction algorithms with mandatory merge constraints, in *ICGI '08: Proceedings of the 9th international colloquium on Grammatical Inference* (Springer-Verlag, Berlin, Heidelberg), ISBN 978-3-540-88008-0, pp. 139–153.
34. Langley, P. (1995). Order effects in incremental learning, in P. Reimann and H. Spada (eds.), *Learning in Humans and Machines: Towards an Interdisciplinary Learning Science.* (Elsevier, Amsterdam).
35. Mernik, M., Heering, J. and Sloane, A. M. (2005). When and how to develop domain-specific languages, *ACM Computing Surveys* **37**, 4, pp. 316–344.
36. Moscato, P. (1989). On evolution, search, optimization, genetic algorithms and martial arts: Towards memetic algorithms, Tech. rep., California Institute of Technology, Concurrent Computation Program 158-79.
37. Nakamura, K. (2006). Incremental learning of context free grammars by bridging rule generation and search for semi-optimum rule sets, in *ICGI '06: Proceedings of the 8th international colloquium on Grammatical Inference*, pp. 72–83.
38. Nakamura, K. and Matsumoto, M. (2005). Incremental learning of context free grammars based on bottom-up parsing and search, *Pattern Recognition* **38**, 9, pp. 1384–1392.
39. Neighbors, J. (1984). The Draco approach to constructing software from reusable components, *IEEE Transactions on Software Engineering* **10**, 5, pp. 564–574.
40. Nevill-Manning, C. G. and Witten, I. H. (1997). Identifying hierarchical structure in sequences: a linear-time algorithm, *Journal of Artificial Intelligence Research* **7**, 1, pp. 67–82.
41. Northrop, L., Feiler, P., Gabriel, R. P., Goodenough, J., Linger, R., Longstaff, T., Kazman, R., Klein, M., Schmidt, D., Sullivan, K. and Wallnau, K. (2006). Ultra-large-scale systems: The software challenge of the future, Tech. rep., Software Engineering Institute, Carnegie Mellon University, `http://www.sei.cmu.edu/uls/`.
42. Oncina, J. and Garcia, P. (1991). Inferring regular languages in polynomial update time, in N. P. de la Blanca, A. Sanfeliu and E. Vidal (eds.), *Pattern Recognition and Image Analysis*, *Series in Machine Perception and Artificial Intelligence*, Vol. 1 (World Scientific, Singapore), pp. 49–61.
43. Paakki, J. (1995). Attribute grammar paradigms—a high-level methodology in language implementation, *ACM Computing Surveys* **27**, 2, pp. 196–255.
44. Parekh, R. and Honavar, V. G. (2001). Learning DFA from simple examples, *Machine Learning* **44**, 1-2, pp. 9–35.
45. Perlin, K. and Hoffert, E. M. (1989). Hypertexture, *ACM SIGGRAPH Computer Graphics* **23**, 3, pp. 253–262.
46. Petasis, G., Paliouras, G., Spyropoulos, C. D. and Halatsis, C. (2004). eg-GRIDS: Context-free grammatical inference from positive examples using genetic search, in *ICGI '04: Proceedings of the 7th international colloquium on Grammatical Inference*, pp. 223–234.
47. Rissanen, J. (1978). Modeling by shortest data description, *Automatica* **14**, pp. 465–471.

48. Rodrigues, E. and Lopes, H. S. (2006). Genetic programming with incremental learning for grammatical inference, *HIS 2006: Proceedings of the 6th International Conference on Hybrid Intelligent Systems* , pp. 47–50.
49. Sakakibara, Y. (2005). Learning context-free grammars using tabular representations, *Pattern Recognition* **38**, 9, pp. 1372–1383.
50. Schmidt, D. C. (2006). Guest editor's introduction: Model-driven engineering, *Computer* **39**, pp. 25–31.
51. Simonyi, C., Christerson, M. and Clifford, S. (2006). Intentional software, *OOPSLA 2006: Proceedings of the 21st ACM SIGPLAN conference on Object-Oriented Programming, Systems, Languages, and Applications* **41**, 10, pp. 451–464.
52. Slonneger, K. and Kurtz, B. L. (1995). *Formal Syntax and Semantics of Programming Languages: A Laboratory Based Approach* (Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA), ISBN 0201656973.
53. Solan, Z., Horn, D., Ruppin, E. and Edelman, S. (2005). Unsupervised learning of natural languages, in *Proceedings National Academy of Sciences*, pp. 11629–11634.
54. Spinellis, D. (2001). Notable design patterns for domain-specific languages, *Journal of Systems and Software* **56**, 1, pp. 91–99.
55. Strembeck, M. and Zdun, U. (2009). An approach for the systematic development of domain-specific languages, *Software: Practice and Experience* **39**, 15, pp. 1253–1292.
56. Strnad, D. and Guid, N. (2004). Modeling trees with hypertextures, *Computer Graphics Forum* **23**, 2, pp. 173–187.
57. Strnad, D. and Guid, N. (2008). Private communication.
58. Tennent, R. D. (1977). Language design methods based on semantic principles, *Acta Informatica* **8**, pp. 97–112.
59. Thibault, S. (1998). *Domain-Specific Languages: Conception, Implementation and Application*, Phd thesis, Université de Rennes 1, France.
60. van Deursen, A. and Klint, P. (2002). Domain-specific language design requires feature descriptions, *Journal of Computing and Information Technology* **10**, 1, pp. 1–17.
61. van Deursen, A., Klint, P. and Visser, J. (2000). Domain-specific languages: An annotated bibliography, *SIGPLAN Notices* **35**, 6, pp. 26–36.
62. van Zaanen, M. (2002). Implementing alignment-based learning, in *ICGI '02: Proceedings of the 6th International Colloquium on Grammatical Inference* (Springer-Verlag, London, UK), ISBN 3-540-44239-1, pp. 312–314.
63. Vitányi, P. and Li, M. (1997). *An Introduction to Kolmogorov Complexity and Its Applications* (Springer), ISBN 987-0-387-49820-1.
64. Watt, D. A. (1990). *Programming language concepts and paradigms* (Prentice-Hall, Inc., Upper Saddle River, NJ, USA), ISBN 0-13-728874-3.
65. Weiss, D. M. and Lai, C. T. R. (1999). *Software product-line engineering: a family-based software development process* (Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA), ISBN 0-201-69438-7.
66. Wile, D. (2004). Lessons learned from real DSL experiments, *Science of Computer Programming* **51**, 3, pp. 265–290.

67. Wirth, N. (1974). On the design of programming languages, in *IFIP Congress*, pp. 386–393.
68. Wortman, D. B. (ed.) (1977). *Proceedings of an ACM conference on Language design for reliable software* (ACM, New York, NY, USA).

This page is intentionally left blank

Chapter 9

Small Size Insertion and Deletion Systems

Artiom Alhazov[1]

IEC, Department of Information Engineering,
Graduate School of Engineering, Hiroshima University,
Higashi-Hiroshima 739-8527 Japan

Alexander Krassovitskiy

[2]*Research Group on Mathematical Linguistics, Rovira i Virgili University,*
Av. Catalunya, 35, Tarragona 43002 Spain,
E-mail: `alexander.krassovitskiy@estudiants.urv.cat`

Yurii Rogozhin[2]

[1]*Institute of Mathematics and Computer Science,*
Academy of Sciences of Moldova,
Academiei 5, Chişinău MD-2028 Moldova,
E-mail: `{artiom,rogozhin}@math.md`

Sergey Verlan[1]

LACL, Département Informatique, Université Paris Est,
61 av. Général de Gaulle, 94010 Créteil, France,
E-mail: `verlan@univ-paris12.fr`

9.1 Introduction

The operations of insertion and deletion have a long history and they were first considered with a linguistic motivation [25, 10, 32]. Another inspiration for these operations comes from the fact that the insertion operation and its iterated variants are generalized versions of Kleene's operations of concatenation and closure [19], while the deletion operation generalizes the quotient operation. A study of properties of the corresponding operations may be found in [11, 12, 15]. However, insertion and deletion also have a nice biological motivation and correspond to a mismatched annealing of DNA sequences. Such operations are also present in the evolution processes in the form of point mutations as well as in RNA editing, see the discussions in [4, 5, 37] and [35]. This biological motivation of insertion-deletion operations led to their study in the framework of molecular computing, see, for example, [7, 16, 35, 38].

In general form, an insertion operation means adding a substring to a given string in a specified (left and right) context, while a deletion operation means removing a substring of a given string from a specified (left and right) context. A finite set of insertion-deletion rules, together with a set of axioms provide a language generating device: starting from the set of initial strings and iterating insertion-deletion operations as defined by the given rules one gets a language.

Insertion-deletion systems are quite powerful, leading to characterizations of recursively enumerable languages. However, there are classes of insertion-deletion systems that are decidable. In these cases it is possible to consider a graph-controlled variant of insertion-deletion systems, known under the name of insertion-deletion P systems, which permits to increase the computational power of corresponding systems.

This chapter is organized as follows. Section 9.2 introduces the definitions used in this chapter. Sections 9.3 and 9.4 present some basic results from the area of insertion-deletion systems, as well as common proof methods. Section 9.5 deals with *context-free* insertion-deletion systems and it contains, in particular, two important results of the area: Theorems 9.7 and 9.15. Section 9.6 investigates *one-sided* insertion-deletion systems, *i.e.* systems whose rules have a left (or right) context only. Section 9.7 concentrates on *pure insertion* systems, *i.e.* which do not use the deletion operation. Sections 9.8 and 9.9 present results on graph-controlled insertion-deletion systems. The final section contains some bibliographical remarks.

9.2 Definitions

We do not present here definitions concerning standard concepts of the theory of formal languages and we refer to [36] for more details.

We denote by $|w|$ the length of a word w. For a letter a and a word w we denote by $|w|_a$ the number of letters a in w. We extend this notation to $|w|_V$, where V is an alphabet, which gives the number of letters from V in w. If A is a set of words, then we put $|A| = \max_{w \in A} |w|$. The cardinality of a set A will be denoted by $card(A)$. By $alph(w)$ we denote the set of letters occurring in w.

For a word $w \in V^*$ we denote by $Perm(w)$ all words w' with the same number of letters as w: $Perm(w) = \{w' : |w'|_a = |w|_a \text{ for all } a \in V\}$, and we denote by $⧢$ the binary shuffle operation. We recall that $x ⧢ y = \{x_1y_1 \cdots x_ny_n \mid x = x_1 \cdots x_n, y = y_1 \cdots y_n, x_i, y_i \in V^*, 1 \leq i \leq n\}$. The empty string is denoted by λ.

In the sequel we will use some normal forms of type-0 grammars.

A grammar $G = (N, T, S, P)$ is said to be in *Pentonnen normal form* if it has rules of form $AB \to AC$, $A \to x$, where $A, B, C \in N$, A, B, C being different, $x \in (N \cup T)^*$ and $|x| \leq 2$. We can also assume that x is either λ or equal to uv, where $u, v \in N \cup T$ and $A \neq u$, $u \neq v$, $A \neq v$.

A grammar $G = (N, T, S, P)$ is said to be in *special Pentonnen normal form* if it has rules of form $AB \to AC$, $BA \to CA$, $A \to AB$, $A \to BA$, $A \to \delta$, where $A, B, C \in N$ are different and $\delta \in N \cup T \cup \{\lambda\}$.

A grammar $G = (N, T, S, P)$ is said to be in *Kuroda normal form* if it has rules of form $AB \to CD$, $A \to BC$, $A \to \delta$, where $A, B, C, D \in N$ and $\delta \in T \cup \{\lambda\}$.

The *Dyck language* D_n over $T_n = \{a_1, \bar{a}_1, \ldots, a_n, \bar{a}_n\}$, $n \geq 1$ is the context-free language generated by the grammar

$$G = (\{S\}, T_n, S, \{S \to \lambda, S \to SS\} \cup \{S \to a_iS\bar{a}_i \mid 1 \leq i \leq n\}).$$

Intuitively, the pairs $(a_i, \bar{a}_i), 1 \leq i \leq n$, can be viewed as parentheses, left and right, of different kinds. Then D_n consists of all strings of correctly nested parentheses. Sometimes it is convenient to define the Dyck language D over some alphabet V. In this case $n = card(V)$.

The family of matrix languages, *i.e.*, the family of languages generated by matrix grammars without appearance checking is denoted by MAT. The family of recursively enumerable languages is denoted by RE. The Parikh image of a language family F is a family of sets of vectors denoted by PsF (we assume a fixed ordering on the alphabet $T = \{a_1, \ldots, a_n\}$):

$$Ps(L) = \{(|w|_{a_1}, \dots, |w|_{a_n}) \mid w \in L\},$$
$$PsF = \{Ps(L) \mid L \in F\}.$$

A *register machine* (introduced in [29], see also [9]) is a construct

$$\mathcal{M} = (d, Q, q_0, h, P)\,,$$

where

- d is the number of registers,
- Q is a finite set of bijective labels of instructions of P,
- $q_0 \in Q$ is the initial label,
- $h \in Q$ is the halting label, and
- P is the set of instructions of the following forms:

(1) p : $(\mathrm{ADD}(k), q, s)$, with $p, q, s \in Q$, $1 \leq k \leq d$ ("increment"-instruction). Add 1 to register k and go to one of the instructions with labels q, s.
(2) p : $(\mathrm{SUB}(k), q, s)$, with $p, q, s \in Q$, $1 \leq k \leq d$ ("decrement"-instruction). Subtract 1 from the positive value of register k and go to the instruction with label q, otherwise (if it is zero) go to the instruction with label s.
(3) h : HALT (the halt instruction). Stop the computation of the machine.

For generating languages over T, we use the model of a *register machine with output tape* (introduced in [29], see also [1]), which also uses a tape operation:

(4) $p : (\mathrm{WRITE}(A), q)$, with $p, q \in Q$, $A \in T$.

The configuration of a register machine is given by the $(d + 1)$-tuple $(q, n_1, \dots, n_d)$, where $q \in Q$ and $n_i \geq 0, 1 \leq i \leq d$, describing the current label of the machine as well as the contents of all registers. A transition of the register machine consists in updating/checking the value of a register according to an instruction of one of types above and by changing the current label to another one. We say that the machine stops if it reaches the label h. A (non-deterministic) register machine $\mathcal{M}$ is said to generate a vector $(n_1, \dots, n_m)$ of natural numbers if, starting from configuration $(q_0, 0, \dots, 0)$ the machine stops in configuration $(h, n_1, \dots, n_m, 0, \dots, 0)$. The set of all vectors generated in this way by $\mathcal{M}$ is denoted by $Ps(\mathcal{M})$. It is known (e.g., see [29], [41]) that register machines generate $PsRE$. If the WRITE instruction is used, then RE can be generated.

In the case when a register machine cannot check whether a register is empty we say that it is *partially blind*; the second type of instructions is then written as $p : (\text{SUB}(k), q)$ and the transition is undefined if register k is zero.

Partially blind register machines have an implicit test for zero at the end of a (successful) computation: counters $m+1, \cdots, d$ should be empty. It is known [9] that partially blind register machines generate exactly $PsMAT$ (Parikh sets of languages of matrix grammars without appearance checking).

9.2.1 ***Insertion-Deletion Systems***

An *insertion-deletion system* is a construct $ID = (V, T, A, I, D)$, where V is an alphabet, $T \subseteq V$, A is a finite language over V, and I, D are finite sets of triples of the form (u, α, v), $\alpha \neq \lambda$, where u and v are strings over V. The elements of T are *terminal* symbols (in contrast, those of $V - T$ are called nonterminals), those of A are *axioms*, the triples in I are *insertion rules*, and those from D are *deletion rules*. An insertion rule $(u, \alpha, v) \in I$ indicates that the string α can be inserted between u and v, while a deletion rule $(u, \alpha, v) \in D$ indicates that α can be removed from the context (u, v). As stated otherwise, $(u, \alpha, v) \in I$ corresponds to the rewriting rule $uv \to u\alpha v$, and $(u, \alpha, v) \in D$ corresponds to the rewriting rule $u\alpha v \to uv$. We denote by $\Longrightarrow_{ins}$ the relation defined by an insertion rule (formally, $x \Longrightarrow_{ins} y$ iff $x = x_1uvx_2, y = x_1u\alpha vx_2$, for some $(u, \alpha, v) \in I$ and $x_1, x_2 \in V^*$) and by $\Longrightarrow_{del}$ the relation defined by a deletion rule (formally, $x \Longrightarrow_{del} y$ iff $x = x_1u\alpha vx_2, y = x_1uvx_2$, for some $(u, \alpha, v) \in D$ and $x_1, x_2 \in V^*$). We refer by $\Longrightarrow$ to any of the relations $\Longrightarrow_{ins}, \Longrightarrow_{del}$, and denote by $\Longrightarrow^*$ the reflexive and transitive closure of $\Longrightarrow$ (as usual, $\Longrightarrow^+$ is its transitive closure).

The language generated by ID is defined by

$$L(ID) = \{w \in T^* \mid x \Longrightarrow^* w, x \in A\}.$$

The complexity of an insertion-deletion system $ID = (V, T, A, I, D)$ is described by the vector $(n, m, m'; p, q, q')$ called *size*, where

$$n = \max\{|\alpha| \mid (u, \alpha, v) \in I\},\ \ p = \max\{|\alpha| \mid (u, \alpha, v) \in D\},$$

$$m = \max\{|u| \mid (u, \alpha, v) \in I\},\ \ q = \max\{|u| \mid (u, \alpha, v) \in D\},$$

$$m' = \max\{|v| \mid (u, \alpha, v) \in I\},\ \ q' = \max\{|v| \mid (u, \alpha, v) \in D\}.$$

We also denote by $INS_n^{m,m'}DEL_p^{q,q'}$ corresponding families of insertion-deletion systems. Moreover, we define the total size of the system as the sum of all numbers above: $\psi = n + m + m' + p + q + q'$.

If some of the parameters n, m, m', p, q, q' is not specified, then we write instead the symbol $*$. In particular, $INS_*^{0,0}DEL_*^{0,0}$ denotes the family of languages generated by *context-free insertion-deletion systems.* If one of numbers from the couples m, m' and/or q, q' is equal to zero (while the other is not), then we say that corresponding families have a one-sided context.

We remark that early investigations like [11, 12, 15] considered only context-free variants of insertion and deletion operations, separately. This comes from the fact that a context-free insertion (resp. deletion) may be considered like an insertion (resp. deletion) of one word into another and this can be used to define the corresponding operation on two languages.

We also remark that, historically, another complexity measure called *weight* was used for insertion-deletion systems. It corresponds to 4-tuples $(n, \bar{m}; p, \bar{q})$, where $\bar{m} = \max\{m, m'\}$ and $\bar{q} = \max\{q, q'\}$.

9.2.2 *Graph-Controlled Insertion-Deletion Systems*

Like context-free grammars, insertion-deletion systems may be extended by adding some additional controls. We discuss here the adaptation of the idea of programmed and graph-controlled grammars for insertion-deletion systems.

A *graph-controlled insertion-deletion system* is the construct $\Pi = (V, T, A, i_0, i_f, R_1, \dots, R_n)$, where

- V is a finite alphabet,
- $T \subseteq V$ is the terminal alphabet,
- $A \subseteq V^*$ is the set of axioms,
- $1 \leq i_0 \leq n$ is the *initial component*,
- $1 \leq i_f \leq n$ is the *final component*,
- R_i, for each $1 \leq i \leq n$, called *component*, is a set of insertion and deletion rules with target indicators of the following form: $(u, x, v; tar)_a$, where (u, x, v) is an insertion rule, and $(u, x, v; tar)_e$, where (u, x, v) is an deletion rule, and tar is from the set $\{1, \dots, n\}$.

If there is no confusion, we will use the term component i instead of component R_i. A configuration of Π is represented by a couple (i,w), where i is the number of the *current* component (initially i_0) and w is the current string. We also say that w is *situated* in component i. A transition $(i,w) \Rrightarrow (j,w')$ is performed as follows. First, a rule r from component R_i is non-deterministically chosen and applied; the new set from which the rule to be applied will be chosen is R_j. More formally, $(i,w) \Rrightarrow (j,w')$ if there is $r:(u,\alpha,v;j)_t \in R_i$ such that either $w \Longrightarrow_{ins} w'$ (if $t=a$) or $w \Longrightarrow_{del} w'$ (if $t=e$) by the rule (u,α,v). The result of the computation consists of all terminal strings situated in component i_f reachable from the axiom and the initial component: $L(\Pi)=\{w \in T^* \mid (i_0,w') \Rrightarrow^* (i_f,w), w' \in A\}$.

It is also possible to consider another definition of graph-controlled insertion-deletion systems, which is equivalent to the one above. We can define such systems as the tuple (V,T,A,i_0,i_f,R), where V, T, A, i_0 and i_f are defined as above, while R contains rules of form $i:(u,\alpha,v;E)_t$, where $i \in \mathit{Lab}(R)$, $\mathit{Lab}(R)$ being a set of labels associated (in a one-to-one manner) to the rules in R, $E \subseteq \mathit{Lab}(R)$ is a set of labels of rules from R and u, α, v are defined as for insertion or deletion rules and $t \in \{a,e\}$. During the derivation step, an insertion or deletion is performed, depending whether t is equal to a or to e, and the next rule to be applied is non-deterministically chosen from the set E. Formally, a configuration of Π will be represented by a couple (i,w), where i is the label of the rule to be applied and w is the current string. A transition $(i,w) \Rrightarrow (j,w')$ is performed if there is a rule $i:(u,\alpha,v;E)_t$ in R such that either $w \Longrightarrow_{ins} w'$ (if $t=a$) or $w \Longrightarrow_{del} w'$ (if $t=e$) by a rule (u,α,v) and $j \in E$. The result of the computation consists of all terminal strings reachable from the axiom and the initial label: $L(\Pi)=\{w \in T^* \mid (i_0,w') \Rrightarrow^* (i_f,w), w' \in A\}$.

Is is not difficult to see that the first definition of graph-controlled insertion-deletion systems can be easily reduced to the second one. In order to do this, it is enough for every $r:(u,\alpha,v;j)_t$ in R_i to add a rule $i:(u,\alpha,v;R_j)_t$ to R. The converse inclusion is also true and can be obtained by the subset construction.

We define the communication graph of a graph-controlled insertion-deletion system the graph with nodes $1,\dots,n$ having an edge between node i and j if there exists a rule $(u,\alpha,v;j) \in R_i$. We are particularly interested in systems whose communication graph has a tree structure.

We remark that the presentation of graph-controlled insertion-deletion given above coincides with the definition of insertion-deletion P systems [33]. Traditionally, in the literature, the term of insertion-deletion P systems is used for graph-controlled insertion-deletion systems, however, in what follows, we will use the latter term, because of a much simpler definition.

We follow [33] and denote by $ELSP_k(ins_n^{m,m'}, del_p^{q,q'})$ the family of languages $L(\Pi)$ generated by graph-controlled insertion-deletion systems with $k \geq 1$ components and insertion and deletion rules of size at most $(n, m, m'; p, q, q')$ and whose communication graph has a *tree structure*. We omit the letter E if $T = V$ and replace k by $*$ if k is not fixed. We also consider graph-controlled insertion-deletion systems where deletion rules have a priority over insertion rules, *i.e.*, no insertion rule can be applied if a deletion rule is applicable; the corresponding class is denoted as $(E)LSP_k(ins_n^{m,m'} < del_p^{q,q'})$. The letter "t" is inserted before P to denote classes whose communication graph is arbitrary, e.g., $ELStP_k(ins_n^{m,m'}, del_p^{q,q'})$.

Sometimes we are only interested in the multiplicities of each symbol in the output words, *i.e.*, in the Parikh image of the languages described above. In this case we say that a family of sets of vectors is generated and we replace L by Ps in the notation above, e.g., $PsStP_k(ins_p^{m,m'}, del_p^{q,q'})$.

9.3 Basic Simulation Principles

In this section we show some important properties of insertion-deletion systems, present some normal forms and indicate basic methods for equivalence proofs used in the rest of the sequel.

We start with the following lemma giving a normal form for insertion-deletion systems.

Lemma 9.1. *For any insertion-deletion system $ID = (V, T, A, I, D)$ of size $(n, m, m'; p, q, q')$ it is possible to construct an insertion-deletion system $ID_2 = (V \cup \{X, Y\}, T, A_2, I_2, D_2 \cup D_2')$ of the same size such that $L(ID_2) = L(ID)$. Moreover, all rules from I_2 have the form (u, α, v), where $|\alpha| = n$, $|u| = m$, $|v| = m'$, all rules from D_2 have the form (u', α, v'), where $|\alpha| = p$, $|u'| = q$, $|v'| = q'$ and $D_2' = \{(\varepsilon, X, \varepsilon), (\varepsilon, Y, \varepsilon)\}$.*

Proof. Consider

$$\begin{aligned}A_2 =&\{X^iwY^tY^j \mid w \in A, i = \max(m,q), j = \max(m',q'), t = \max(p-|w|,0)\},\\ I_2 =&\{(z_1, xY^k, z_2) \mid (a,x,b) \in I, z_1 \in \{a \sqcup\!\sqcup X^*\}, z_2 \in \{b \sqcup\!\sqcup Y^*\}\\ &\text{and } |xY^k| = n, k \geq 0, |z_1| = m, |z_2| = m'\}\cup\\ &\cup \{(z_1, Y^n, z_2) \mid z_1, z_2 \in (V \cup \{X,Y\})^*, |z_1| = m, |z_2| = m'\}\\ D_2 =&\{(z_1, d, z_2) \mid (a,x,b) \in D, z_1 \in \{a \sqcup\!\sqcup X^*\}, z_2 \in \{b \sqcup\!\sqcup Y^*\}, d \in \{x \sqcup\!\sqcup Y^*\}\\ &\text{and } |d| = p, |z_1| = q, |z_2| = q'\}.\end{aligned}$$

In fact, any rule having a left (resp. right) context of a smaller size is replaced by a group of rules, where the left (resp. right) context is a string over $V \cup \{X\}$ (resp. $V \cup \{Y\}$) of the required size. The same holds for the inserted or deleted symbol. Any axiom $w \in A$ is surrounded by X and Y ($X^iwY^tY^j$) in A_2. It is clear that if $w \in L(ID)$ then the word $X^iw'Y^j$, $w' \in \{w \sqcup\!\sqcup Y^*\}$ will be obtained in ID_2 using the corresponding rules and starting from the corresponding axiom. Now symbols X and Y can be erased by rules from D_2' which implies that $w \in L(ID_2)$. It is clear that if rules from D_2' are used before this step, then at most the same w may be obtained. Hence $L(ID) = L(ID_2)$. □

Next we prove the following lemma which shows that the deletion of terminal symbols may be excluded.

Lemma 9.2. *For any insertion-deletion system $ID = (V,T,A,I,D)$ there is a system $ID' = (V \cup V', T, A \cup A', I \cup I', D')$ such that $L(ID') = L(ID)$. Moreover, b does not contain letters from T for any rule $(a,b,c) \in D'$.*

Proof. Indeed, we can transform system ID to an equivalent system ID' as follows.

Let $V' = \{N_t \mid t \in T\}$. Consider the coding function $f : V \to V \cup V'$ defined by $f(x) = N_x$ if $x \in T$ and $f(x) = x$ otherwise. Consider also the following extension to words (where id is the identity function):

$$F(a_1 \ldots a_n) = \{g_1(a_1) \ldots g_n(a_n) \mid g_i \in \{f, id\},\ 1 \leq i \leq n\}$$

Now for any rule (a,b,c) in D (resp. in I) we introduce rules (a',b',c') in D' (resp. I'), where $a' \in F(a)$, $b' \in F(b)$ and $c' \in F(c)$. For any axiom $w \in V^*$ we add $F(w)$ to the axioms. Finally, we remove all rules $(a,b,c) \in D'$ having $|b|_T \neq 0$.

This construction ensures that the nonterminal symbol N_t acts like an alias for the symbol $t \in T$, *i.e.* for any derivation producing $w_1 t w_2$ there is another derivation producing $w_1 N_t w_2$. Hence there is no difference between erasing t or N_t. This proves the statement of the lemma. $\square$

Insertion-deletion systems represent a powerful model of computation. If the size of the system is not bounded, then an arbitrary grammar can be simulated.

Theorem 9.3. *For any type-0 grammar $G = (N, T, S, P)$ there is an insertion-deletion system $ID = (V, T, A, I, D)$ such that $L(G) = L(ID)$.*

Proof. Let $V = N \cup \{\#_i : 1 \leq i \leq |P|\} \cup \{\$\}$. Let $k_1 = \max\{|u|, u \to v \in P\}$ and $k_2 = \max\{|v|, u \to v \in P\}$. Consider $k = \max(k_1, k_2)$. The set A is defined as $A = \{\$^k S \$^k\}$.

For any rule $i : u \to v \in P$ we add insertion rules $(xu, \#_i v, y)$, $x, y \in (N \cup \{\$\})^*$, $|xu| = k$, $|y| = k$, to I and a deletion rule $(x, u\#_i, v)$, $x \in N \cup \{\$\}$ to D. Finally, a rule $(\lambda, \$, \lambda)$ is added to D.

It is not difficult to see that such system simulates G. Indeed, for any derivation $w_1 u w_2 \Longrightarrow w_1 v w_2$ in G there is a two-step derivation $\$^k w_1 u w_2 \$^k \Longrightarrow \$^k w_1 u \#_i v w_2 \$^k \Longrightarrow \$^k w_1 v w_2 \$^k$ in ID that simulates the corresponding production of G. If $w \in L(G)$ then the string $\$^k w \$^k$ will be obtained in ID. Additional symbols \$ can be deleted at this moment. So $w \in L(ID)$.

For the converse inclusion it is enough to observe that if an insertion rule $(xu, \#_i v, y)$ is used, then no more insertions inside the corresponding site xu can be done. Hence the only way to eliminate the symbol $\#_i$ is to perform the corresponding deletion. Hence the computation in ID can be rearranged in such a way that an insertion is followed by the corresponding deletion. This corresponds to a derivation step in G, which completes the proof. $\square$

As one can see from the previous theorem, the basic idea of grammar simulation by insertion-deletion systems is a construction of a set of related insertion and deletion rules that shall be used in some specified sequence, thus performing a grammar rule simulation. Usually, insertion rules introduce new nonterminal symbols in the string which can be deleted only by the corresponding deletion rules (like the symbols $\#_i$ in theorem above). If the correct sequence is not performed, then some nonterminal symbols that cannot be deleted will remain in the string. In the subsequent sections different variants of this method are shown, thereby decreasing the size of the insertion and deletion rules.

A simulation of type-0 grammars by insertion-deletion systems is the main method to prove the computational completeness of insertion-deletion systems. However, when several such results are established, it is much easier to prove the computational completeness by simulating other insertion-deletion systems. For example:

Theorem 9.4. [35] $INS_1^{1,1}DEL_2^{0,0} = RE$.

Proof. [Sketch] The proof may be done by simulating insertion-deletion systems of size $(1,1,1;1,1,1)$ which are known to be computationally complete, see [38, 39] and also Theorem 9.5. In this case it is enough to show how a deletion rule (a,b,c), $a,b,c \in V$ can be simulated using insertion and deletion rules of size $(1,1,1;2,0,0)$. Let $a \neq b \neq c$. Then a deletion rule (a,b,c) with label i may be simulated by a sequence of the following rules: $\{(a,\}_i,b),(b,]_i,c),(a,[_i,\}_i),([_i,\{_i,\}_i),([_i,K_i,\{_i,)\} \subseteq I$ and $(\lambda,\{_i\}_i,\lambda),(\lambda,K_ib,\lambda),(\lambda,[_i]_i,\lambda) \subseteq D$. The simulation is performed as follows:

$$\begin{aligned}
w_1abcw_2 &\Longrightarrow w_1a\}_ibcw_2 \Longrightarrow w_1a[_i\}_ibcw_2 \Longrightarrow w_1a[_i\}_ib]_icw_2 \Longrightarrow \\
&\Longrightarrow w_1a[_i\{_i\}_ib]_icw_2 \Longrightarrow w_1a[_iK_i\{_i\}_ib]_icw_2 \Longrightarrow \\
&\Longrightarrow w_1a[_iK_ib]_icw_2 \Longrightarrow w_1a[_i]_icw_2 \Longrightarrow w_1acw_2.
\end{aligned}$$

The idea behind the simulation is the following. Symbols $[_i$ and $]_i$ delimit the deletion site. Symbol K_i performs the deletion of b, while symbols $\}_i$ and $\{_i$ ensure that K_i is inserted only once after $[_i$ (hence only one b can be deleted). If all the above steps are not performed, then some of additional symbols will remain in the string, hence it will never become terminal. This is a common method of simulation: the working (insertion or deletion) site is delimited by special symbols in order to avoid interactions between several such sites and inside the site the sequence of insertions and deletions permits to simulate exactly one application of the corresponding rule. All additional symbols are related in such a way that the whole sequence of insertions and deletions shall be performed in order to eliminate all of them.

We remark that it would be wrong to simulate a deletion rule (a,b,c) by only rules $\{(a,[_i,b),(b,]_i,c),([_i,K_i,b)\} \subseteq I$ and $(\lambda,K_ib,\lambda),(\lambda,[_i]_i,\lambda) \subseteq D$, because it is possible to erase several symbols b, which leads to a wrong computation:

$$w_1abbcw_2 \Longrightarrow w_1a[_ibbcw_2 \Longrightarrow w_1a[_ibb]_icw_2 \Longrightarrow w_1a[_iK_ibb]_icw_2 \Longrightarrow$$
$$\Longrightarrow w_1a[_ib]_icw_2 \Longrightarrow w_1a[_iK_ib]_icw_2 \Longrightarrow w_1a[_i]_icw_2 \Longrightarrow w_1acw_2. \quad \square$$

9.4 Insertion-Deletion Systems with Rules of Small Size

This section presents some important insertion-deletion systems able to generate any RE language. The systems from this section will be used as a starting point for computational completeness proofs like the proof of Theorem 9.4.

We start with the following theorem.

Theorem 9.5. *$INS_1^{1,1}DEL_1^{1,1} = RE$.*

Proof. We give only a sketch for this proof. Full details might be found in [38, 39].

Consider a type-0 grammar $G = (N, T, S, P)$ in Pentonnen normal form. We construct a system $ID = (V, T, A, I, D)$ simulating G as follows.

The alphabet V is defined as $V = N \cup T \cup \{X, Y\} \cup \{[r], (r) \mid r \in P\}$ and $A = \{XSY\}$.

For every rule $AB \to AC$ of P we add the following insertion rules to I (below we consider that $\gamma \in N \cup T \cup \{X, Y\}$):

$$(A, [r], B) \qquad (B, (r), \gamma) \qquad ([r], C, (r))$$

and the following deletion rules to D:

$$([r], B, (r)) \qquad (A, [r], C) \qquad (C, (r), \gamma)$$

For every rule $A \to uv$ of P we add the following insertion rules to I:

$$(\gamma, [r], A) \qquad (A, (r), \gamma) \qquad ([r], u, (r)) \qquad (u, v, (r))$$

and the following deletion rules to D:

$$([r], A, (r)) \qquad (\gamma, [r], u) \qquad (v, (r), \gamma)$$

For every rule $A \to \lambda$ of P we add the deletion rule (γ_1, A, γ) to D, where $\gamma_1, \gamma \in N \cup T \cup \{X, Y\}$. Finally, we add to D rules (λ, X, λ) and (λ, Y, λ).

The simulation is performed as follows. Consider that a string w_1ABw_2, $w_1, w_2 \in (N \cup T \cup \{X, Y\})^+$ is present in ID. Then the following evolution can happen in ID (we suppose that the label of rule $AB \to AC$ of P is r):

$$w_1ABw_2 \Longrightarrow w_1A[r]Bw_2 \Longrightarrow w_1A[r]B(r)w_2 \Longrightarrow w_1A[r](r)w_2 \Longrightarrow$$
$$\Longrightarrow w_1A[r]C(r)w_2 \Longrightarrow w_1AC(r)w_2 \Longrightarrow w_1ACw_2.$$

Some of the operations above are not enforced, like the last two deletions. However, if they are not performed, the symbols A and C cannot participate in simulation of any other grammar rules, because such a simulation requires that such a symbol is surrounded by symbols from $N \cup T$. The simulation of rules of type $A \to uv$ is done in a similar way. $\square$

Theorem 9.6. $INS_2^{0,0}DEL_1^{1,1} = RE.$

Proof. We consider here only a sketch of the proof. The full proof can be found in [21].

The proof of the theorem is based on a simulation of insertion-deletion systems of size $(2, 0, 0; 3, 0, 0)$. It is known that these systems generate any recursively enumerable language, see Theorem 9.10 taken from [26]. Consider $ID = (V, T, A, I, D)$ to be such a system. Now we construct a system $ID_2 = (V_2, T, A, I_2, D_2)$ of size $(2, 0, 0; 1, 1, 1)$ that will generate same language as ID.

It is clear that in order to show the inclusion $L(ID) \subseteq L(ID_2)$ it is sufficient to show how a deletion rule $(\lambda, abc, \lambda) \in D$, with $a, b, c \in V$, may be simulated by using rules of system ID_2, *i.e.*, insertion rules of type (λ, xy, λ) and deletion rules of type (a', y', b'), with $a', b' \in V_2 \cup \{\lambda\}$, $x, y, y' \in V_2$.

For every deletion rule (λ, abc, λ) of ID we may suppose that $a \neq b \neq c$.

Consider $V_2 = V \cup \{L_i, L_i', R_i, R_i', K_i, K_i' \mid 1 \leq i \leq card(D)\}$.

Let us label all rules from D by integer numbers. Consider now a rule $i : (\lambda, abc, \lambda) \in D$, where $1 \leq i \leq card(D)$ is the label of the rule. Then the following insertion rules are introduced to I_2:

$$1 : (\lambda, L_iL_i', \lambda) \qquad 2 : (\lambda, R_i'R_i, \lambda) \qquad 3 : (\lambda, K_iK_i', \lambda)$$

and the following deletion rules are introduced to D_2 $(l, m \in V)$:

$$4 : (L_i, L_i', a) \qquad 5 : (L_i, a, b) \qquad 6 : (c, R_i', R_i)$$
$$7 : (b, c, R_i) \qquad 8 : (L_i, b, R_i) \qquad 9 : (K_i, K_i', L_i)$$
$$10 : (K_i, L_i, R_i) \qquad 11 : (K_i, R_i, m) \qquad 12 : (l, K_i, m)$$

The rule $i : (\lambda, abc, \lambda) \in D$ is simulated as follows. We first perform two insertions:

$$w_1abcw_2 \Longrightarrow^1 w_1L_iL'_iabcw_2 \Longrightarrow^2 w_1L_iL'_iabcR'_iR_iw_2.$$

and then some deletions:

$$w_1L_iL'_iabcR'_iR_iw_2 \Longrightarrow^4 w_1L_iabcR'_iR_iw_2 \Longrightarrow^6 w_1L_iabcR_iw_2 \Longrightarrow^5$$
$$\Longrightarrow^5 w_1L_ibcR_iw_2 \Longrightarrow^7 w_1L_ibR_iw_2 \Longrightarrow^8 w_1L_iR_iw_2.$$

Now we delete symbols L_iR_i using same technique as above with the help of $K_iK'_i$:

$$w_1L_iR_iw_2 \Longrightarrow^3 w_1K_iK'_iL_iR_iw_2 \Longrightarrow^9 w_1K_iL_iR_iw_2 \Longrightarrow^{10}$$
$$\Longrightarrow^{10} w_1K_iR_iw_2 \Longrightarrow^{11} w_1K_iw_2 \Longrightarrow^{12} w_1w_2.$$

Hence, $L(ID) \subseteq L(ID_2)$. In order to prove the converse inclusion, we observe that if the whole sequence of insertion and deletion rules above is not performed, then some nonterminal symbols are left and cannot be deleted anymore. Moreover, the above sequence permits the elimination of three symbols abc in a string. Indeed, the symbol L_i deletes a if and only if it was inserted at the left of a at some point. Similarly, R_i deletes c if it was inserted at the right of c at some point. Now, R_i and L_i are eliminated if and only if they meet, which means that they delete b (and of course a and c). In order to delete L_i, $K_iK'_i$ must be inserted before it. Symbol K_i deletes symbols K'_i, L_i and R_i, and is eliminated only after that. □

9.5 Context-Free Insertion-Deletion Systems

In this section we present one of the most important results in the area of insertion-deletion systems: the computational completeness of systems with context-free rules. We remark that initially insertion and deletion operations on words were considered only in a context-free manner. Hence this result answers a very old question from this area.

We give here the full proof taken from [26].

Theorem 9.7. *$INS_*^{0,0}DEL_*^{0,0} = RE$.*

Proof. Let $G = (N, T, S, P)$ be a type-0 Chomsky grammar, where N, T are disjoint alphabets, $S \in N$, and P is a finite subset of rules of the form $u \to v$ with $u, v \in (N \cup T)^*$ and u contains at least one letter from N.

We assume that all rules from P are labeled in a one-to-one manner with elements of a set M, disjoint from $N \cup T$.

We construct the following context-free insertion-deletion system $\gamma = (N \cup T \cup M, T, \{S\}, I, D)$, where

$$I = \{(\lambda, vR, \lambda) \mid R : u \to v \in P,\ R \in M,\ u, v \in (N \cup T)^*\},$$
$$D = \{(\lambda, Ru, \lambda) \mid R : u \to v \in P,\ R \in M,\ u, v \in (N \cup T)^*\}.$$

Two rules $(\lambda, vR, \lambda) \in I, (\lambda, Ru, \lambda) \in D$ as above are said to be *M-related.*

We prove the equality $L(G) = L(\gamma)$.

The inclusion $L(G) \subseteq L(\gamma)$ is obvious: each derivation step $x_1ux_2 \Longrightarrow x_1vx_2$, performed in G by means of a rule $R : u \to v$, can be simulated in γ by an insertion operation step $x_1ux_2 \Longrightarrow_{ins} x_1vRux_2$ which uses the rule $(\lambda, vR, \lambda) \in I$, followed by the deletion operation $x_1vRux_2 \Longrightarrow_{del} x_1vx_2$ which uses the rule $(\lambda, Ru, \lambda) \in D$.

Consider now the inclusion $L(\gamma) \subseteq L(G)$. The idea of the proof is to transform any terminal derivation in γ into one in which any two consecutive derivation steps (an odd one and the even one following it) simulate one production in G. Because the labels of rules from P precisely identify a pair of M-related insertion-deletion rules, and the elements of M are non-terminal symbols for γ, every terminal derivation with respect to γ must involve the same number of insertion steps and deletion steps; moreover, these steps are performed by using pairs of M-related rules from I and D.

Consider an arbitrary terminal derivation in γ,

$$\delta : S \Longrightarrow w_1 \Longrightarrow w_2 \Longrightarrow \cdots \Longrightarrow w_{2k} = w \in T^*,$$

where $k \geq 1$ is the number of the pairs of M-related insertion-deletion rules used in this derivation. Let $w_i \Longrightarrow_{ins} w_{i+1} \Longrightarrow w_{i+2}$ be a subderivation of δ such that the step $w_i \Longrightarrow_{ins} w_{i+1}$ is performed by a rule $(\lambda, vR, \lambda) \in I$ and the step $w_{i+1} \Longrightarrow w_{i+2}$ is performed by using a rule other than the M-related rule $(\lambda, Ru, \lambda) \in D$. We say that the pair of rules used in the two mentioned steps *do not match.*

Assume now that the derivation δ contains $m > 0$ non-matching pairs of rules. Let us identify a pair of M-related rules $(\lambda, vR, \lambda) \in I$ and $(\lambda, Ru, \lambda) \in D$ which are used for the same occurrence of R in δ but not in consecutive steps (that is, this pair introduces a non-matching sequence of rules in δ):

$$\delta : S \Longrightarrow^* z_1z_2 \Longrightarrow_{ins} z_1vRz_2 \Longrightarrow^+ y_1Ruy_2 \Longrightarrow_{del} y_1y_2 \Longrightarrow^* w,$$

for some $z_1, z_2, y_1, y_2 \in (N \cup T \cup M)^*$. Therefore, we have:

$$S \Longrightarrow^* z_1 z_2, \tag{9.1}$$
$$z_1 z_2 \Longrightarrow_{ins} z_1 v R z_2, \tag{9.2}$$
$$z_1 v \Longrightarrow^* y_1, \tag{9.3}$$
$$z_2 \Longrightarrow^* u y_2, \tag{9.4}$$
$$y_1 y_2 \Longrightarrow^* w. \tag{9.5}$$

Clearly, in at least one of the relations 9.3, 9.4 we have $\Longrightarrow^+$ in place of $\Longrightarrow^*$, since the sequence of rules is non-matching.

We rearrange the previous derivations as follows. From 9.1 and 9.4 we have

$$S \Longrightarrow^* z_1 z_2 \Longrightarrow^* z_1 u y_2.$$

We can now apply the insertion rule as in 9.2 and we have

$$z_1 u y_2 \Longrightarrow_{ins} z_1 v R u y_2,$$

and then the deletion rule $(\lambda, Ru, \lambda) \in D$:

$$z_1 v R u y_2 \Longrightarrow_{del} z_1 v y_2.$$

From 9.3 and 9.5 we can now obtain

$$z_1 v y_2 \Longrightarrow^* y_1 y_2 \Longrightarrow^* w.$$

Consequently, we have obtained a derivation

$$\delta' : S \Longrightarrow^* z_1 z_2 \Longrightarrow^* z_1 u y_2 \Longrightarrow_{ins} z_1 v R u y_2 \Longrightarrow_{del} z_1 v y_2 \Longrightarrow^* y_1 y_2 \Longrightarrow^* w,$$

which produces the same terminal string w and has at most $m-1$ non-matching pairs of rules.

Continuing in this way, for every terminal derivation in γ we can construct an equivalent derivation, using the same rules in a different order, and having only matching pairs of consecutive rules. Clearly, two consecutive steps of a derivation in γ which use M-related rules $(\lambda, vR, \lambda) \in I, (\lambda, Ru, \lambda) \in D$, correspond to a derivation step in G which uses the rule $R : u \to v$. This implies the inclusion $L(\gamma) \subseteq L(G)$. □

The system constructed in the previous proof has only one axiom, hence this complexity parameter has an optimal value.

The context control of a type-0 grammar does not really disappear in the corresponding insertion-deletion system (as constructed in Theorem 9.7 above). It rather changes its form, becoming a rigid synchronization of insertions and deletions. In other terms, if a word u represents the context of a word v in a "context-sensitive production" $R: \ u \to v$, then in the corresponding insertion-deletion system the word v will also be conditioned by the later occurrence of u in a successful derivation (hence u is yet again the context of v). This condition is enforced by the newly introduced symbol R which acts as a "remote context binder". The fact that the context u "seems" to appear after the context-controlled v is of no importance, reflecting the reversal of generative process of the grammar.

We illustrate the construction from the proof with a simple example: consider a context-sensitive grammar $G = (\{S, X, Y\}, \{a, b, c\}, S, P)$ with the set of productions

$$\begin{aligned} P = \{R_1 : S \to aSX,\ R_2 : S \to aY,\ R_3 : YX \to bYc, \\ R_4 : cX \to Xc,\ R_5 : Y \to bc\}. \end{aligned}$$

It is easy to see that it generates the non-context-free language $L(G) = \{a^i b^i c^i \mid i \geq 1\}$.

The obtained system is

$$\begin{aligned} \gamma &= (V, \{a, b, c\}, \{S\}, I, D), \text{ where} \\ V &= \{S, X, Y, a, b, c, R_1, R_2, R_3, R_4, R_5\}, \\ I &= \{(\lambda, aSXR_1, \lambda),\ (\lambda, aYR_2, \lambda),\ (\lambda, bYcR_3, \lambda), \\ &\qquad (\lambda, XcR_4, \lambda),\ (\lambda, bcR_5)\}, \\ D &= \{(\lambda, R_1S, \lambda),\ (\lambda, R_2S, \lambda),\ (\lambda, R_3YX, \lambda), \\ &\qquad (\lambda, R_4cX, \lambda),\ (\lambda, R_5Y, \lambda)\}. \end{aligned}$$

Consider a derivation for the word $a^3b^3c^3$ in grammar G:

$$\begin{aligned} S \Longrightarrow aSX \Longrightarrow aaSXX \Longrightarrow aaaYXX \Longrightarrow aaabYcX \Longrightarrow \\ \Longrightarrow aaabYXc \Longrightarrow aaabbYcc \Longrightarrow aaabbbccc. \end{aligned}$$

One of the corresponding derivations in γ is the following:

$$\begin{aligned} & S \Longrightarrow_{ins} aSXR_1S \Longrightarrow_{ins} aaSXR_1SXR_1S \Longrightarrow_{ins} \\ & \Longrightarrow_{ins} aaaYR_2SXR_1SXR_1S \Longrightarrow_{del} aaaYR_2SXXR_1S \Longrightarrow_{del} \\ & \Longrightarrow_{del} aaaYXXR_1S \Longrightarrow_{ins} aaabYcR_3YXXR_1S \Longrightarrow_{del} \\ & \Longrightarrow_{del} aaabYcXR_1S \Longrightarrow_{ins} aaabYXcR_4cXR_1S \Longrightarrow_{ins} \\ & \Longrightarrow_{ins} aaabbYcR_3YXcR_4cXR_1S \Longrightarrow_{del} aaabbYcR_3YXcR_1S \Longrightarrow_{del} \\ & \Longrightarrow_{del} aaabbYccR_1S \Longrightarrow_{ins} aaabbbcR_5YccR_1S \Longrightarrow_{del} \\ & \Longrightarrow_{del} aaabbbcR_5Ycc \Longrightarrow_{del} aaabbbccc. \end{aligned}$$

In the proof of Theorem 9.7, the length of inserted or deleted strings is not bounded, but a bound can be easily found by controlling the length of strings appearing in the rules of the underlying type-0 grammar:

Theorem 9.8. $INS_3^{0,0}DEL_3^{0,0} = RE.$

Proof. Let $G = (N, T, S, P)$ be type-0 Chomsky grammar in Kuroda normal form. Then, the rules of the context-free insertion-deletion system constructed in the proof of Theorem 9.7 are of the form $(\lambda,\ \alpha,\ \lambda)$ with $|\alpha| \leq 3$, hence $RE \subseteq INS_3^0 DEL_3^0$. □

The total size of the system provided by the proof of Theorem 9.7 is 6. We can improve this result by one, decreasing by one either the length of the inserted strings or the length of the deleted strings. We give below sketches of proofs for both cases. These proofs are made by a direct simulation of systems of size $(3, 0, 0; 3, 0, 0)$. A different approach is given in [26], where a grammar in Kuroda normal form is simulated.

Theorem 9.9. $INS_3^{0,0}DEL_2^{0,0} = RE.$

Proof. [Sketch] A deletion rule $i : (\lambda, abc, \lambda)$ can be simulated by an insertion rule $(\lambda, C_iB_iA_i, \lambda)$ and three deletion rules (λ, A_ia, λ), (λ, B_ib, λ) and (λ, C_ic, λ). The simulation works as follows.

$$w_1abcw_2 \Longrightarrow w_1C_iB_iA_iabcw_2 \Longrightarrow w_1C_iB_ibcw_2 \Longrightarrow w_1C_icw_2 \Longrightarrow w_1w_2.$$

The proof of the validity of this simulation may be obtained in a similar way to Theorem 9.7. We leave the details as a task for the reader and only note that the rules above have the weight as requested in the statement of the theorem. □

A counterpart of this result is also true: there is a trade-off between the lengths of inserted and deleted strings.

Theorem 9.10. $INS_2^{0,0}DEL_3^{0,0} = RE.$

Proof. [Sketch] An insertion rule $i : (\lambda, abc, \lambda)$ can be simulated by three insertion rules (λ, aA_i, λ), (λ, bB_i, λ), (λ, cC_i, λ) and a deletion rule $(\lambda, C_iB_iA_i, \lambda)$. The simulation works as follows.

$$w_1w_2 \Longrightarrow w_1aA_iw_2 \Longrightarrow w_1abB_iA_iw_2 \Longrightarrow w_1abcC_iB_iA_iw_2 \Longrightarrow w_1abcw_2.$$

As above, we leave the details of the proof as a task for the reader. □

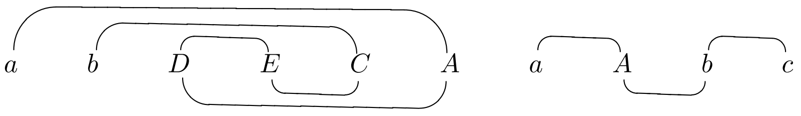
9.5.1 *Non-completeness Results*

We show below that the obtained complexity parameters for context-free insertion-deletion systems are optimal. If one of the parameters is further decreased, then the language generated by such systems is included in the family of context-free languages.

The following lemma shows that the nonterminal alphabet is not relevant if one starts from an empty word.

Lemma 9.11. *Let $ID = (V, T, A, I, D)$ be a context-free insertion-deletion system of size $(2,0,0;2,0,0)$. Suppose also that $A = \{\lambda\}$. Then there is a system $ID_2 = (T, T, \{\lambda\}, I_2, D_2)$ of size $(2,0,0;2,0,0)$ such that $L(ID) = L(ID_2)$.*

Proof. Consider a derivation of $w \in T^*$. Let us mark the corresponding insertion pairs by an overline and the corresponding deletion pairs by an underline. For example, suppose that we insert aA, after that bC in position 1, DE in position 2, aA in position 6 and bc in position 8. After that suppose that we delete EC, DA and Ab. Then the corresponding marking will be as follows (the resulting word is $w = abac$):



We may interpret symbols as labeled graph nodes and lines as edges. In this case we obtain a graph. It is easy to observe that this graph consists of a set of disjoint linear paths and/or cycles. Indeed, for each node, at most two edges corresponding to an insertion and a deletion may be drawn. Let us also label edges corresponding to insertions by i and edges corresponding to deletions by d. If we take the example above, we obtain:

$$a \overset{i}{\text{———}} A \overset{d}{\text{———}} D \overset{i}{\text{———}} E \overset{d}{\text{———}} C \overset{i}{\text{———}} b$$

$$a \overset{i}{\text{———}} A \overset{d}{\text{———}} b \overset{i}{\text{———}} c$$

We may suppose that the first and the last edge of a path are marked with i. If this is not the case, we add an additional node labeled by λ and we connect this node with the last node by a path labeled by i. In particular, a path containing only one letter a (corresponding to an insertion of a) will be written as $\lambda \overset{i}{\text{—}} a$. Hence, each path consists of sequences of one insertion followed by one deletion.

We observe that if we consider a derivation of $w \in T^*$ then there may only be paths of the following 4 types:

(1) Paths that start with a letter $a \in T$ and that end with a letter $b \in T$.
(2) Paths that have at one end a terminal letter a and at the other end λ.
(3) Paths that have λ at both ends.
(4) Cycles.

We remark that in Case 1 the path leads to the word ab (*i.e.*, contributes to the production of the subword ab of w), in the second case the path produces the letter a and in the last two cases the path generates the empty word.

More exactly, let p be a path. We denote by $yield(p)$ the word produced by p:

$$yield(p) = \begin{cases} ab, \text{ if } p = a \overset{i}{-} \cdots \overset{i}{-} b, \ a, b \in T \cup \{\lambda\}. \\ \lambda, \ \text{ if } p \text{ is a cycle.} \end{cases}$$

Let $xy \neq \lambda$, $x, y \in V \cup \{\lambda\}$ be a word. We denote by $yield(xy)$ the set of all words that may be produced by a path containing xy:

$yield(xy) = \{yield(p), \text{ where } x \overset{i}{-} y \in p\}$.

We remark that $|yield(xy)| \leq |xy|$.

Without loss of generality, we may suppose that there are no paths of type 3 and 4, because by eliminating the corresponding insertions and deletions we obtain the same word.

Suppose that we have a path marked by over- and underlines as above. We shall understand by an interior of the path the set of all positions that are underlined. In the example above, all positions between D and the first A form the interior of the path. It is clear that no other path (of type 1 and 2) may be situated in the interior of some path, because in this case the corresponding deletion cannot be performed. Consequently, all paths are independent of each other, and we may group rules corresponding to each path and compute paths one after another. Moreover, each path contributes to at most two terminal symbols of the resulting word. Therefore, the computation consists of insertion of terminal symbols corresponding to paths ends as well as of deletion of terminal symbols.

Now, in order to prove the lemma, it is enough to show that me may precompute all possible paths. This may be done by using the following observation. We may assume that each path p has the following property: if $A \overset{i}{-} B$ belongs to p, then p does not contain an insertion that has

A in the left-hand side $(A \overset{i}{-} X)$ or B in the right-hand side $(Y \overset{i}{-} B)$. This assertion is obvious, because if p contains such a pair, for example $p = \cdots \overset{d}{-} A \overset{i}{-} X \overset{d}{-} \cdots \overset{d}{-} A \overset{i}{-} B \cdots$, then we may eliminate the subpath between two A's by obtaining an equivalent path (that leads to the same ends) $p' = \cdots \overset{d}{-} A \overset{i}{-} B \cdots$. Hence, the length of each path is bounded by $2 \cdot card(V)$, and we may precompute all possible paths.

Finally, let I' be the subset of I that contains only terminal insertion rules, *i.e.*, any rule of I' is of form (λ, ab, λ) or (λ, a, λ), where $a, b \in T$. Analogously, let D' be the subset of terminal deletion rules of D. Let us also precompute all possible paths that end with a terminal letter or λ (of type 1 and 2). Let us denote by I_1 the set of rules inserting the words produced by these paths. More exactly, we define $I_1 = \{(\lambda, yield(xy), \lambda) \mid (\lambda, xy, \lambda) \in I\}$. It is clear that the system $ID_2 = (T, T, \{\lambda\}, I_2, D_2)$, where $I_2 = I' \cup I_1$ and $D_2 = D'$ generates the same language as ID. It is also clear that the converse inclusion holds. Hence, our assertion is proved. □

In a similar manner it can be proved that the nonterminal alphabet is not relevant even in the general case. See [40] for details.

Lemma 9.12. *Let $ID = (V, T, A, I, D)$ be a context-free insertion-deletion system of size $(2, 0, 0; 2, 0, 0)$. Then there is a system $ID_2 = (T, T, A_2, I_2, D_2)$ of size $(2, 0, 0; 2, 0, 0)$ such that $L(ID) = L(ID_2)$.*

The following lemma (see [40]) shows that we can eliminate the deletion operation. This can be also done by first eliminating terminal deletions according to Lemma 9.2, and after that applying the construction from the previous two lemmas.

Lemma 9.13. *Let $ID = (T, T, A, I, D)$ be a context-free insertion-deletion system of size $(2, 0, 0; 2, 0, 0)$. Then there is a system $ID_2 = (T, T, A_2, I_2, \emptyset)$ of size $(2, 0, 0; 0, 0, 0)$ such that $L(ID) = L(ID_2)$.*

Next theorem shows that insertion-deletion systems of size $(2, 0, 0; 0, 0, 0)$ can be described by a context-free grammar. This is a particular case of a more general result for systems of size $(*, 1, 1; 0, 0, 0)$ given in [35].

Lemma 9.14. *Let $ID = (T, T, A, I, \emptyset)$ be a context-free insertion-deletion system of size $(2, 0, 0; 0, 0, 0)$. Then there is a context-free grammar $G = (N, T, Z, P)$ such that $L(ID) = L(G)$.*

Proof. We construct G as follows. Consider $N = \{Z, S\}$ and let T be the terminal alphabet of ID. Define $P = P_A \cup P_I \cup \{S \to \lambda\}$, where

$$P_A = \{Z \to Sa_1Sa_2S \dots Sa_nS \mid a_1a_2 \dots a_n \in A\},$$

$$P_I = \{S \to SaSbS \mid (\lambda, ab, \lambda) \in I\} \cup \{S \to SaS \mid (\lambda, a, \lambda) \in I\}.$$

It is clear that $L(G) = L(ID)$. Indeed, symbol S marks all possible insertion positions and permits the simulation of insertion rules as well. □

Consequently, we obtain:

Theorem 9.15. *$INS_2^{0,0}DEL_2^{0,0} = INS_2^{0,0}DEL_0^{0,0} \subset CF$.*

Proof. The strictness of the inclusion follows from the fact that the language $L = \{a^*b^*\}$ cannot be generated by a context-free insertion-deletion system of size $(2, 0, 0; 2, 0, 0)$. Indeed, consider an arbitrary system $ID = (T, T, A, I, \emptyset)$. It is easy to observe that for each word w that belong to $L(ID)$ words $\{x^*wx^* \mid (\lambda, x, \lambda) \in I\}$ belong to $L(ID)$. Therefore, if we suppose that $L(ID)$ is not finite, then $I \neq \emptyset$, and then for any word $w \in L(ID)$, there are words $\{x^*wx^* \mid (\lambda, x, \lambda) \in I\}$ in $L(ID)$. It is easy to see that L does not have such a property. □

Theorem 9.16. *$INS_2^{0,0}DEL_2^{0,0}$ is incomparable with REG.*

Proof. From the previous theorem we obtain that $REG \setminus INS_2^{0,0}DEL_2^{0,0} \neq \emptyset$. It is also clear that the Dyck language D_n may be generated by a context-free insertion system having insertion rules $(\lambda, a_i\bar{a}_i, \lambda)$, $1 \le i \le n$. Hence, the assertion is proved. □

From Lemma 9.14 and Theorem 9.15 is is clear that languages generated by insertion-deletion systems of size $(2, 0, 0; 2, 0, 0)$ have a particular structure (below, we denote by $\prod$ the concatenation operation).

Theorem 9.17. *A language L belongs to $INS_2^{0,0}DEL_2^{0,0}$ if and only if it can be represented in the form*

$$L = h\left(T'^* \sqcup\!\sqcup \bigcup_{w=a_1\dots a_n \in A} \prod_{i=1}^{|w|} Da_iD\right),$$

where $A \subseteq T^*$ is a finite set of words, T is an alphabet, D is the Dyck language over an alphabet $T'' \subseteq T$, h is a coding and $T' \subseteq T$.

In a similar way next two results can be obtained. See [40] for more details.

Theorem 9.18. *$INS_m^{0,0}DEL_1^{0,0} = INS_m^{0,0}DEL_0^{0,0} \subset CF$ for any $m > 0$.*

Theorem 9.19. *$INS_1^{0,0}DEL_p^{0,0} \subset REG$ for any $p > 0$.*

9.6 One-Sided Contextual Insertion-Deletion Systems

In this section we present results about insertion-deletion systems with one-sided context, *i.e.*, of size $(n, m, m'; p, q, q')$ where either $m + m' > 0$ and $m \cdot m' = 0$ or $q + q' > 0$ and $q \cdot q' = 0$, *i.e.*, one of numbers in some couple is equal to zero.

The proofs are based on simulation of insertion-deletion systems from Sections 9.4 and 9.5 which are known to generate all RE languages. The proof technique is very similar to the one from Theorem 9.4.

We give the sketch of proof for the following theorem.

Theorem 9.20. *$INS_1^{1,2}DEL_1^{1,0} = RE$.*

Proof. [Sketch] The proof is based on the simulation of insertion-deletion systems of size $(1, 1, 1; 1, 1, 1)$. By Lemma 9.1 it is sufficient to show how a deletion rule (a, x, b), with $a, b, x \in V$, may be simulated by using rules of the target system, *i.e.*, insertion rules of type $(a', x', b'c')$ and deletion rules of type (a'', y, λ).

According to Lemma 9.1, we may assume that $ab \neq \lambda$. Moreover, we may assume that the system has no insertion rules of the form $(a, b, b), a, b \in V$. If this is the case then we replace every such rule by two insertion rules (a, X, b), (a, b, X), and one deletion rule (b, X, b), where X is a new nonterminal.

A deletion rule $i : (a, x, b)$, where i is the label of the rule, is simulated by two insertion rules (x, X_i, b), (a, D_i, xX_i) and three deletion rules (D_i, x, λ), (D_i, X_i, λ), (a, D_i, λ).

Symbols D_i and X_i act like left and right parentheses that surround x before deleting it. The simulation is performed as follows. First, two insertions are performed:

$$w_1axbw_2 \Longrightarrow_{ins} w_1axX_ibw_2 \Longrightarrow_{ins} w_1aD_ixX_ibw_2,$$

and then x is deleted:

$$w_1aD_ixX_ibw_2 \Longrightarrow_{del} w_1aD_iX_ibw_2.$$

At this moment symbols X_i and D_i are deleted:

$$w_1aD_iX_ibw_2 \Longrightarrow_{del} w_1aD_iw_2 \Longrightarrow_{del} w_1abw_2.$$

Hence, every derivation in an insertion-deletion system of size $(1, 1, 1; 1, 1, 1)$ can be carried out in a system of size $(1, 1, 2; 1, 1, 0)$. On the other hand, we observe that once being inserted, the nonterminals X_i, D_i can be erased only by the rules shown above. Moreover, if they are not deleted, then no symbol can be inserted at the right of a or at the left of b. The rule (D_i, x, λ) can delete at most one x as the pair D_ix is followed by X_ib and $b \neq x$. Thus, there is a one-to-one correspondence between the original and the new systems, which implies that the theorem statement holds. □

The following theorem shows the trade-off between the left context of the insertion and the size of the inserted string.

Theorem 9.21. $INS_2^{0,2}DEL_1^{1,0} = RE$.

Proof. [Sketch] We prove the theorem by simulating systems from the previous theorem. An insertion rule $i : (a, x, bc)$ can be simulated by an insertion rule (λ, Y_ix, bc) and a deletion rule (a, Y_i, λ).

We remark that since the left context is empty, a rule (λ, xy, bc) may be applied any number of times, hence the language $w_1(xy)^+bcw_2$ (we can also write it as $w_1(xy)^*xybcw_2$ or $w_1xy(xy)^*bcw_2$) may be obtained from w_1bcw_2. This behavior shall be taken into account.

The rule $i : (a, x, bc)$ is simulated as follows. We first perform an insertion:

$$w_1abcw_2 \Longrightarrow_{ins}^{+} w_1aY_ix(Y_ix)^*bcw_2.$$

After that we delete Y_i:

$$w_1aY_ix(Y_ix)^*bcw_2 \Longrightarrow_{del} w_1ax(Y_ix)^*bcw_2.$$

If only one insertion is performed during the insertion step, then we obtain the string w_1axbcw_2. Hence, $L(ID) \subseteq L(ID_2)$.

For the converse inclusion see [28]. □

The next result decreases the size of the insertion context at the price of increasing the size of the deletion strings.

Theorem 9.22. [28] $INS_2^{0,1}DEL_2^{0,0} = RE$.

Proof. The proof of the theorem is based on a simulation of insertion-deletion systems of size $(1, 1, 1; 2, 0, 0)$. Consider $ID = (V, T, A, I, D)$ to be such a system. We now construct system $ID_2 = (V_2, T, A, I_2, D_2)$ of size $(2, 0, 1; 2, 0, 0)$ that will generate same language as ID.

Like in the previous theorems, we show that any rule $(a, x, b) \in I$ with $a, x, b \in V$ may be simulated by rules of system ID_2, *i.e.*, insertion rules of type $(\lambda, a'b', c')$ and deletion rules of type $(\lambda, a'b', \lambda)$, with $a', b', c' \in V_2 \cup \{\lambda\}$, $0 < |a'b'| \leq 2$.

Consider $V_2 = V \cup \{X_i^1, X_i^2, X_i^3, Y_i^2, Y_i^3, K_i^1, K_i^2 \mid 1 \leq i \leq card(I)\}$.

Let us label all rules from I by integer numbers. Consider now a rule $i : (a, x, b) \in I$, where $1 \leq i \leq card(I)$ is the label of the rule. We introduce the following insertion rules in I_2:

$$(\lambda, X_i^1, b), \tag{9.6}$$
$$(\lambda, X_i^2 Y_i^2, \lambda), \tag{9.7}$$
$$(\lambda, X_i^3 Y_i^3, \lambda), \tag{9.8}$$
$$(\lambda, aK_i^1, \lambda), \tag{9.9}$$
$$(\lambda, xK_i^2, K_i^1), \tag{9.10}$$

and the following deletion rules in D_2:

$$(\lambda, Y_i^2 a, \lambda), \tag{9.11}$$
$$(\lambda, X_i^2 X_i^3, \lambda), \tag{9.12}$$
$$(\lambda, K_i^1 Y_i^3, \lambda), \tag{9.13}$$
$$(\lambda, K_i^2 X_i^1, \lambda). \tag{9.14}$$

We say that these rules are i-related.

Like in the previous theorem, since the left context is empty, an insertion rule (λ, xy, b) may be applied any number of times, hence the language $w_1(xy)^+ bw_2$ (we can also write it as $w_1(xy)^* xybw_2$ or $w_1 xy(xy)^* bw_2$) may be obtained from $w_1 bw_2$.

The simulation of the rule $i : (a, x, b) \in I$ is done in several stages. First the site ab is decorated as follows: $X_i^2 Y_i^2 a X_i^3 Y_i^3 X_i^1 b$ (Y_i^2 is before a, X_i^3 and Y_i^3 mark the interior and X_i^1 marks the position before b). After that ax is inserted inside $X_i^3 Y_i^3$ and finally, the initial a is deleted. More exactly, we first perform insertions of X_i^1 and $X_i^2 Y_i^2$ (in any order):

$$w_1 abw_2 \Longrightarrow_{ins}^{+} w_1 a(X_i^1)^+ bw_2 \Longrightarrow_{ins}^{+} w_1(X_i^2 Y_i^2)^+ a(X_i^1)^+ bw_2.$$

After that we insert $X_i^3 Y_i^3$ and aK_i^1:

$$\begin{aligned} &w_1(X_i^2 Y_i^2)^+ a(X_i^1)^+ bw_2 \Longrightarrow_{ins}^{+} \\ &\qquad w_1(X_i^2 Y_i^2)^+ a((X_i^3 Y_i^3)^+ X_i^1)^+ bw_2 \Longrightarrow_{ins}^{+} \\ &\qquad\qquad w_1(X_i^2 Y_i^2)^+ a((X_i^3 (aK_i^1)^+ Y_i^3)^+ X_i^1)^+ bw_2. \end{aligned}$$

At last we insert xK_i^2:

$$\begin{aligned} &w_1(X_i^2 Y_i^2)^+ a((X_i^3 (aK_i^1)^+ Y_i^3)^+ X_i^1)^+ bw_2 \Longrightarrow_{ins}^{+} \\ &\qquad w_1(X_i^2 Y_i^2)^+ a((X_i^3 (a(xK_i^2)^+ K_i^1)^+ Y_i^3)^+ X_i^1)^+ bw_2. \end{aligned}$$

We can delete Y_i^2a at this stage of computation or earlier (it does not matter):

$$w_1(X_i^2Y_i^2)^+a((X_i^3(a(xK_i^2)^+K_i^1)^+Y_i^3)^+X_i^1)^+bw_2 \Longrightarrow_{del} \\ w_1(X_i^2Y_i^2)^*X_i^2((X_i^3(a(xK_i^2)^+K_i^1)^+Y_i^3)^+X_i^1)^+bw_2.$$

After that deletion rules $(\lambda, X_i^2X_i^3, \lambda)$, $(\lambda, K_i^1Y_i^3, \lambda)$ and $(\lambda, K_i^2X_i^1, \lambda)$ may be applied.

$$w_1(X_i^2Y_i^2)^*X_i^2((X_i^3(a(xK_i^2)^+K_i^1)^+Y_i^3)^+X_i^1)^+bw_2 \Longrightarrow_{del} \\ w_1(X_i^2Y_i^2)^*(X_i^3(a(xK_i^2)^+K_i^1)^+Y_i^3)^*(a(xK_i^2)^+K_i^1)^*ax(X_i^1)^*bw_2.$$

If only one insertion is performed during each insertion step, then we obtain the string w_1axbw_2. Hence, $L(ID) \subseteq L(ID_2)$.

Now we prove that no other words may be obtained using rules above. Indeed, by construction, any insertion rule inserts at least one symbol from $V_2 \setminus V$. So, in order to eliminate it, a corresponding deletion rule is needed. Moreover, inserted symbols may be divided in two categories that group symbols with respect to the deletion. The first category contains X_i^2, Y_i^2, X_i^3, Y_i^3 and K_i^1. It is clear that if any of these symbols is inserted into the string, then all other symbols must be also inserted, otherwise it is not possible to eliminate them. The second group contains symbols X_i^1 and K_i^2. Now let us present some invariants which appear if we want to obtain a terminal string. Suppose that there is $(a, x, b) \in I$ and w_1abw_2 is a word obtained in some step of a derivation in ID. We can deduce the following.

- One of the rules (9.9), (9.7) or (9.8) must be applied, otherwise the simulation cannot start.
- In order to eliminate the introduced symbols, rules (9.13), (9.11) or (9.12) must be applied.
- Rule (9.11) may be applied only if $X_i^2Y_i^2$ is followed by symbol a: $(\dots X_i^2Y_i^2a\dots)$
- Rule (9.12) may be applied only if symbol X_i^3 was preceded by the string $X_i^2Y_i^2a$: $(\dots X_i^2Y_i^2aX_i^3\dots Y_i^3\dots)$.
- Rule (9.13) may be applied only if symbol Y_i^3 is preceded by K_i^1: $(\dots X_i^2Y_i^2aX_i^3\dots aK_i^1Y_i^3\dots)$.

Hence, once one of above rules is applied, all other insertion and deletion rules above must be also applied, otherwise some non-terminal symbols are not eliminated. We also remark that if at this moment the three deletion rules (9.11), (9.12) and (9.13) are performed, then string w_1abw_2 is obtained, *i.e.*, no change was made with respect to the initial string (one a was deleted together with Y_i^2, but at the same time inserted together with K_i^1). To conclude, any of insertions (9.9), (9.7) or (9.8) introduces at least one non-terminal symbol, and in order to eliminate it a specific sequence of above rules shall be used. Moreover, this sequence does not make any changes to the string. In a more general case, it was proved in Lemma (9.11) (see also [40]) that any sequence of context-free insertions and deletions of length at most 2 contributes to at most two symbols of the final terminal word. In our case, the sequence inserts only one terminal a but it also needs to delete an a in order to eliminate all non-terminals.

We are interested in the particular moment when the insertion of aK_i^1 is performed. Now, rule (9.10) inserts the string xK_i^2 between a and K_i^1. After that, the above sequence of insertion and deletion rules is performed and a string $w_1axK_i^2bw_2$ is obtained. At this moment, symbol x is inserted and we also know that its left neighbor is a. Symbol K_i^2 may be eliminated if and only if it is adjacent to X_1 which is always inserted before symbol b. Hence, after eliminating all additional symbols we either obtain the same word (xK_i^2 was not inserted), or insert x between a and b which simulates the corresponding rule of ID. This concludes the proof. $\square$

Now we present counterparts of the three theorems above.

Theorem 9.23. *$INS_1^{1,0}DEL_1^{1,2} = RE$.*

Proof. The proof of the theorem is based on a simulation of insertion-deletion systems of size $(1,1,1;1,1,1)$.

Using Lemma 9.1 it is sufficient to show how an insertion rule $(a,x,b) \in I$, with $a,b,x \in V$, may be simulated by using insertion rules of type (a',x',λ) and deletion rules of type $(a',y',b'c')$, with $a',b',c' \in V_2 \cup \{\lambda\}$, $x',y' \in V_2$.

For every rule $(a,x,b) \in I$ we may suppose that $x \neq b$. Indeed, if this is not the case, then this rule can be replaced by two insertion rules (a,B,b), (a,b,B) and one deletion rule (b,B,b).

Consider a rule $i:(a,x,b)$, where i is the label of the rule. We simulate this rule by following insertion rules:

$$(a,A_i,\lambda), \qquad (A_i,x,\lambda), \qquad (A_i,B_i,\lambda),$$

and the following deletion rules:

$$(x,B_i,b), \qquad (a,A_i,xB_i).$$

Like in the previous proofs, A_i and B_i act like parentheses that surround the insertion site of x. The rule $i : (a, x, b)$ is simulated as follows. We first perform insertions of A_i and x:

$$w_1abw_2 \Longrightarrow w_1a(A_i)^+bw_2 \Longrightarrow w_1a(A_iB_i^+)^+bw_2 \Longrightarrow w_1a(A_i(x+B_i)^+)^+bw_2,$$

and then the deletions (they are applicable to the string $w_1aA_ixB_ibw_2$)

$$w_1aA_ixB_ibw_2 \Longrightarrow w_1axB_ibw_2 \Longrightarrow w_1axbw_2.$$

Now in order to prove the converse inclusion, we observe that after performing insertion of nonterminal symbols A_i and B_i, the only way to remove these symbols is to erase them with the introduced deletion rule. This means that x is inserted between A_i and B_i, B_i is inserted immediately to the left of b, A_i first inserts one B_i and after that one symbol x. To conclude the proof we remark that if more than one A_i, B_i or x is inserted, then it is impossible to eliminate the corresponding symbol. □

Theorem 9.24. $INS_1^{1,0}DEL_2^{0,2} = RE$.

Proof. [Sketch] The proof of the theorem is based on a simulation of insertion-deletion systems of size $(1, 1, 0; 1, 1, 2)$ from Theorem 9.23 above. A deletion rule $i : (a, x, bc)$ is simulated by an insertion rule (a, A_i, λ) and a deletion rule (λ, A_ix, bc). We also suppose that we don't have $x = b = c$. The simulation is performed as follows. We first perform insertions of A_i:

$$w_1axbcw_2 \Longrightarrow^+ w_1a(A_i)^+xbcw_2,$$

and after that one deletion (it is applicable to the string $w_1aA_ixbcw_2$)

$$w_1aA_ixbcw_2 \Longrightarrow w_1abcw_2.$$

The converse inclusion follows from the fact that once inserted, symbol A_i must be deleted by the corresponding rule, therefore the correct simulation is performed. □

The following theorem is the counterpart of Theorem 9.22.

Theorem 9.25. $INS_2^{0,0}DEL_2^{0,1} = RE$.

Proof. [Sketch] The proof of the theorem is based on a simulation of insertion-deletion systems of size $(2,0,0;3,0,0)$.

A deletion rule $i : (\lambda, abc, \lambda)$, where $i)$ is the label of the rule, is simulated by the following group of insertion rules:

$$1 : (\lambda, A_i^{(1)}B_i^{(1)}, \lambda), \quad 2 : (\lambda, A_i^{(2)}B_i^{(2)}, \lambda), \quad 3 : (\lambda, A_i^{(3)}B_i^{(3)}, \lambda),$$
$$4 : (\lambda, A_i^{(4)}B_i^{(4)}, \lambda), \quad 5 : (\lambda, A_i^{(5)}B_i^{(5)}, \lambda),$$

and the following group of deletion rules:

$$6 : (\lambda, aA_i^{(1)}, B_i^{(1)}), \quad 7 : (\lambda, bA_i^{(2)}, B_i^{(2)}), \quad 8 : (\lambda, cA_i^{(3)}, B_i^{(3)}),$$
$$9 : (\lambda, B_i^{(1)}B_i^{(2)}, A_i^{(5)}), \quad 10 : (\lambda, B_i^{(5)}B_i^{(3)}, A_i^{(4)}), \quad 11 : (\lambda, A_i^{(5)}A_i^{(4)}, B_i^{(4)}),$$
$$12 : (\lambda, B_i^{(4)}, \lambda).$$

The simulation is performed as follows. Symbols $A^{(j)_i}$ and $B^{(j)_i}$ act like parentheses and we use them to surround letters b and c in a particular order. This permits to further perform a sequence of deletions that will eliminate all three letters. In more details, we first perform insertions of $A_i^{(j)}B_i^{(j)}$, $j \in \{1,2,3,4,5\}$ using rules (1) – (5):

$$w_1abcw_2 \Longrightarrow^+ w_1aA_i^{(1)}B_i^{(1)}bA_i^{(2)}B_i^{(2)}A_i^{(5)}B_i^{(5)}cA_i^{(3)}B_i^{(3)}A_i^{(4)}B_i^{(4)}w_2.$$

After that deletion rules (6) – (8) are applied:

$$w_1aA_i^{(1)}B_i^{(1)}bA_i^{(2)}B_i^{(2)}A_i^{(5)}B_i^{(5)}cA_i^{(3)}B_i^{(3)}A_i^{(4)}B_i^{(4)}w_2 \Longrightarrow^+$$
$$w_1B_i^{(1)}B_i^{(2)}A_i^{(5)}B_i^{(5)}B_i^{(3)}A_i^{(4)}B_i^{(4)}w_2$$

Now the remaining introduced symbols are removed:

$$w_1B_i^{(1)}B_i^{(2)}A_i^{(5)}B_i^{(5)}B_i^{(3)}A_i^{(4)}B_i^{(4)}w_2 \Longrightarrow^{9,10}$$
$$w_1A_i^{(5)}A_i^{(4)}B_i^{(4)}w_2 \Longrightarrow^{11} w_1B_i^{(4)}w_2 \Longrightarrow^{12} w_1w_2$$

Thus, we obtain the string w_1w_2, so we model rule $i : (\lambda, abc, \lambda) \in D$ correctly. The converse inclusion is shown in [21]. □

The symmetric variant of these results is a corollary to the above theorems.

Corollary 9.26. $INS_1^{2,1}DEL_1^{0,1} = INS_2^{2,0}DEL_1^{0,1} = INS_2^{1,0}DEL_2^{0,0} = RE.$
$INS_1^{0,1}DEL_1^{2,1} = INS_1^{0,1}DEL_2^{2,0} = INS_2^{0,0}DEL_2^{1,0} = RE.$

Another interesting result is shown in [22].

Theorem 9.27. $INS_1^{2,0}DEL_1^{0,2} = INS_1^{0,2}DEL_1^{2,0} = RE.$

Proof. [Sketch] The proof of the theorem is based on a simulation of insertion-deletion systems of size $(1, 1, 0; 1, 1, 2)$.

Using Lemma 9.1 we can assume that with the exception of two deletion rules of the form (λ, X, λ) and (λ, Y, λ) all deletion rules have always two symbol in the right context, and one symbol in the left context.

For every deletion rule (a, x, bc) we may suppose that $a \neq x$. If this is not the case, then this rule may be replaced by two deletion rules (B, a, bc), (a, B, bc) and one insertion rule (a, B, λ), where B is a new nonterminal.

Consider a rule $i : (a, x, bc)$, where i is the label of the rule. This rule is simulated by an insertion rule (ax, B_i, λ) and two deletion rules (λ, x, B_ib) and (λ, B_i, bc) as follows. We first perform insertions of B_i:

$$w_1axbcw_2 \Longrightarrow^+ w_1ax(B_i)^+bcw_2.$$

After that we delete of x and B_i (applicable to $w_1axB_ibcw_2$)

$$w_1axB_ibw_2 \Longrightarrow w_1aB_ibcw_2 \Longrightarrow w_1abcw_2.$$

In a way similar to the previous proofs, it is possible to show that the inserted nonterminal symbols can be deleted only if a correct simulation of the deletion rule (a, x, bc) is performed. □

9.6.1 *Non-completeness Results*

In what follows we show that there are classes of one-sided insertion-deletion systems that are not computationally complete. Most of the results in this subsection are from [24] and [21]. We start with the following result.

Theorem 9.28. [21] $REG \setminus INS_1^{1,0}DEL_1^{1,1} \neq \emptyset$.

Proof. Consider the regular language $L = \{(ba)^+\}$. We claim that there is no insertion-deletion system ID of size (1,1,0;1,1,1) such that $L(ID) = L$.

We shall prove the above statement by contradiction. Suppose there is such system $ID = (V, \{a, b\}, A, I, D)$ and $L(ID) = L$. From Lemma 9.2 we can suppose that ID does not delete terminal symbols.

Consider a terminal derivation in ID: $w \Longrightarrow^+ w_f$, where $w \in A$ and $w_f \in (ba)^+$. Now consider an arbitrary ba block of w_f ($w_f = \alpha ba\beta$, $\alpha, \beta \in (ba)^*$) and take its letter a. Since there are no terminal deletion rules in ID, this letter is either inserted by an insertion rule, or it was a part of an axiom. We may omit the latter case by taking a derivation that produces a string that is long enough. We may also omit the case when this letter a was inserted by a rule $(\lambda, a, \lambda) \in I$, because in this case a may be inserted at any place in the final string, in particular a string $\alpha baa\beta$ might be obtained. Now suppose that this letter was inserted using a rule $(z, a, \lambda) \in I$, $z \in V$:

$$w \Longrightarrow^* w_1 z w_2 \Longrightarrow w_1 z a w_2 \Longrightarrow^* \alpha ba\beta = w_f. \tag{9.15}$$

This means that

$$\begin{aligned} w_1 z &\Longrightarrow^* \alpha b \\ w_2 &\Longrightarrow^* \beta \end{aligned} \tag{9.16}$$

We remark that symbol a might be inserted twice:

$$w \Longrightarrow^* w_1 z w_2 \Longrightarrow w_1 z a w_2 \Longrightarrow w_1 z a a w_2. \tag{9.17}$$

From (9.17) and (9.16) we obtain

$$w \Longrightarrow^* w_1 z a a w_2 \Longrightarrow^* \alpha baa\beta$$

which is a contradiction. □

A counterpart of this result is also true.

Theorem 9.29. [28] $CF \setminus INS_1^{1,1}DEL_1^{1,0} \neq \emptyset$.

In this case the language $L = \{a^n b^n \mid n \geq 0\}$ cannot be generated. In a way similar to Theorem 9.28 it is possible to show that the language $(ba)^+$ cannot be generated by systems of size $(1, 1, 0; 2, 0, 0)$ and $(2, 0, 0; 1, 1, 0)$.

Theorem 9.30. [23] $REG \setminus INS_1^{1,0}DEL_2^{0,0} \neq \emptyset$.

Theorem 9.31. [23] $REG \setminus INS_2^{0,0}DEL_1^{1,0} \neq \emptyset$.

Now we shall concentrate on systems of size $(1, 1, 0; 1, 1, 0)$. We show that the language generated by such insertion-deletion systems is a particular subclass of the family of context-free languages.

We start our investigations by systems that do not contain deletion rules. In the book [35] it is already shown that the family $INS_n^{1,1}DEL_0^{0,0}$, $n \geq 1$, is a subset of the family of context-free languages. Below we show that even a smaller subclass, $INS_1^{1,0}DEL_0^{0,0}$, having one-sided insertion rules contains non-regular context-free languages.

Theorem 9.32. $INS_1^{1,0}DEL_0^{0,0} \cap (CF \setminus REG) \neq \emptyset$.

Proof. In order to fulfill the assertion of the theorem it suffices to show that a non-regular context-free language can be generated by an insertion-deletion system of size $INS_1^{1,0}DEL_0^{0,0}$.

Consider a system $ID = (T, T, \{a\}, I, \emptyset)$, where $T = \{a, b, c, d\}$ and I is defined as follows: $I = \{(a, b, \lambda), (b, c, \lambda), (c, d, \lambda), (d, a, \lambda)\}$.

Let L be the language generated by ID ($L = L(ID)$). It is clear that L can defined by the following formulas:

$$L = L_1 \qquad L_1 = aL_2^* \qquad L_2 = bL_3^* \qquad L_3 = cL_4^* \qquad L_4 = dL_1^*$$

By substituting L_i, for $2 \le i \le 4$ into the description of L_{i-1} we obtain:

$$L_1 = a(b(c(dL_1^*)^*)^*)^*$$

Let $R = \{(abcd)^*(dcb)^*\}$. Consider the language $L'' = L \cap R$. Consider the word $w = abcddbc$ from R. This word is generated in L as follows (we underline the inserted symbol):

$$a \Longrightarrow a\underline{b} \Longrightarrow a\underline{b}b \Longrightarrow ab\underline{c}b \Longrightarrow ab\underline{c}cb \Longrightarrow abc\underline{d}cb \Longrightarrow abc\underline{d}dcb$$

We observe that the generation of the second part of w, the subword dcb, is related to the generation of its first part $abcd$, because every letter is inserted two times: first for the second part and after that for the first part. It is also clear that this is the only way to generate the subword dcb. Moreover, it can be easily seen that such a generation leads to a one-to-one correspondence between $abcd$ and dcb. Now, taking w it is possible to insert a after the first letter d and to continue in a similar manner as before and so on, which gives $w_n = (abcd)^n(dcb)^n$, $n \ge 1$. It is also possible to obtain more copies of $abcd$ by performing insertions of four corresponding letters after d, c, b or a in the first part of w_n. Hence, we finally obtain $L'' = \{(abcd)^i(dcb)^j, j \le i\}$, which is a non-regular context-free language (by the inverse morphism $\{abcd \to x, dcb \to y\}$ it becomes the well known language $\{x^iy^j, 1 \le j \le i\}$). Since the intersection of two regular languages would be regular, we obtain that L is a non-regular context-free language, which concludes the proof. $\square$

The lemma below shows that in the case of the family $INS_1^{1,0}DEL_0^{0,0}$ the corresponding context-free grammar has a very special form.

Lemma 9.33. *For any $L \in INS_1^{1,0}DEL_0^{0,0}$ it is possible to construct a context-free grammar $G = (\{S\} \cup \{S_a \mid a \in T\} \cup T, T, S, P)$, generating L and having rules of the following form:*

$$\begin{array}{ll} S \to w & w \in (\{aS_a \mid a \in T\})^* \\ S_a \to S_abS_bS_a \mid \lambda & a, b \in T \end{array}$$

Proof. Let $L = L(ID)$, where $ID = (T, T, A, I, \emptyset)$ has size $(1, 1, 0; 0, 0, 0)$. We construct the grammar $G = (V, T, S, P)$ as follows:

Take the alphabet $V = T \cup \{S_x \mid x \in T\}$. The set of productions is defined as follows.

For any $(a, b, \lambda) \in I$ we add the following productions to P:

$$S_a \to S_a b S_b S_a \mid \lambda \tag{9.18}$$

For any $a_1 \cdots a_n \in A$ we add the following productions to P:

$$S \to a_1 S_{a_1} \cdots a_n S_{a_n} \tag{9.19}$$

It is easy to observe that $L(G) = L(ID)$. Indeed, after each letter a the grammar G inserts the symbol S_a which may insert (in any order and in any combination) all possible symbols coming after the letter a. Symbol S_a corresponds to a placeholder, indicating that at that place a letter can be inserted by a. □

We remark that it is possible to extend the previous lemma to systems inserting a regular language instead of a symbol.

Now we will describe the family $INS_1^{1,0}DEL_1^{1,0}$. The starting point is the construction given in Lemma 9.33, however it is important to show that all possible deletions may be precomputed. We start with the following definitions.

Definition 9.34. For any set of insertion rules I and for a letter a we define $I_a = \{x \mid (a, x, \lambda) \in I\}$, i.e. the set of all letters that can be inserted next to a.

Definition 9.35. We say that a word w is generated by letter a if there exists a derivation $a \Longrightarrow^* aw$.

Definition 9.36. For an insertion-deletion system $ID = (V, T, A, I, D)$ we denote by $L_{ID}(a)$ the language generated by the system $ID_a = (V, T, \{a\}, I, D)$.

We remark that any word in an insertion-deletion system of size $(1, 1, 0; 0, 0, 0)$ will have a particular structure: for any word $w = w'aw''$ of $L(ID)$ the set of words $\{w'a(L_{ID}(b))^* w''\}$, $b \in I_a$ will be also part of $L(ID)$. Hence, a letter a will be followed by a repetition of blocks $L_{ID}(b)$, $b \in I_a$.

The next lemma shows that in order to compute the effect of deletion rules for a system of size $(1, 1, 0; 1, 1, 0)$ it is enough to take only blocks containing non-repeating letters, thus giving a limit on the width of a derivation tree that should be examined in order to compute the effect of deletion rules.

Lemma 9.37. *Consider an insertion-deletion system* $ID = (V, T, A, I, D)$ *of size* $(1, 1, 0; 1, 1, 0)$. *Take a letter* $a \in V$ *and a letter* $b \in I_a$. *Consider a derivation of a word* w *in* ID:

$$w' \Longrightarrow^* z'az'' \Longrightarrow^* z'\ a\ u_1b\ u_2b\ \cdots\ u_nb\ u_{n+1}\ z'' \Longrightarrow^* \\ \Longrightarrow^* z'\ a\ u_1by_1\ u_2by_2\ \cdots\ u_nby_n\ u_{n+1}\ z'' \Longrightarrow^* w,$$

where $w' \in A$, $z', z'', w, y_j, u_j \in V^*$, $|u_j|_b = 0$, $1 \leq j \leq n + 1$, *and* u_j *is generated by* a *and* y_j *is generated by* b.

If during the derivation the symbol b *from the block* u_iby_i, $i \geq 2$ *is deleted by some symbol* $d \in V$ *belonging to the word* $u_1by_1 \cdots u_i$, *then we may suppose that this* d *belongs to* u_i $(u_i = u_i'du_i'')$, i.e., *b can be deleted only by a symbol from the same block.*

Proof. We will show that for any derivation that does not fulfill the above property it is possible to construct an equivalent derivation which will satisfy the conditions above.

Let d not be a part of u_i. Then there are several possible cases for the position of d:

(1) d belongs to by_{i-1},
(2) d belongs to u_{i-1},
(3) d belongs to u_kby_k, $k < i - 1$,

Consider the first case. Let $by_{i-1} = xdx'$. This implies that $b \Longrightarrow^* xd$, $dx' \Longrightarrow^* d$ and $du_ib \Longrightarrow^* d$. Then we can rearrange the derivation as follows (below we denote by v the word $u_{i+1}by_{i+1}\ \cdots\ u_nby_n\ u_{n+1}$ and we underline the inserted part):

$$z'\ a\ z'' \Longrightarrow^* z'\ a\ \underline{v}\ z'' \Longrightarrow z'\ a\ \underline{b}\ v\ z'' \Longrightarrow^* \\ z'\ a\ b\ \underline{y_i}\ v\ z'' \Longrightarrow^* z'\ a\ b\underline{xd}\ y_i\ v\ z'' \Longrightarrow^* \\ z'\ a\ \underline{u_{i-1}}bxd\ y_i\ v\ z'' \Longrightarrow^* z'\ \underline{u_1by_1\ \cdots u_{i-2}by_{i-2}}u_{i-1}bxd\ y_i\ v\ z''.$$

The derivation above shows that it is possible to generate directly $bxdy_i$ without generating $x'u_ib$.

Now consider the second case. Let $u_{i-1} = xdx'$. Then $a \Longrightarrow^* xd$, $dx'by_{i-1} \Longrightarrow^* d$ and $du_i \Longrightarrow^* d$. Then we can rearrange the derivation as follows (below we denote by v the word $u_{i+1}by_{i+1} \cdots u_nby_n\, u_{n+1}$ and we underline the inserted part):

$$z'\ a\ z'' \Longrightarrow^* z'\ a\ \underline{v}\ z'' \Longrightarrow z'\ a\ \underline{b}\ v\ z'' \Longrightarrow^* z'\ a\ b\underline{y_i}\ v\ z'' \Longrightarrow^*$$
$$z'\ a\ \underline{xd}\, by_i\ v\ z'' \Longrightarrow z'\ a\ xd\ y_i\ v\ z'' \Longrightarrow^* z'\ a\ \underline{u_1by_1\ \cdots u_{i-2}by_{i-2}}\ xd\ y_i\ v\ z''.$$

The above derivation satisfies the condition of the lemma, because d belongs to the same block as b.

The third case can be reduced to the second one by observing that in this case we do not need to generate the subsequence $u_{k+1}by_{k+1} \cdots u_{i-1}by_{i-1}$ from a, because it is erased anyway. □

Remark 9.38. We remark that the in the case of the first block u_1by_1, symbol b may be deleted by a symbol d from u_1; in this case we can extend Lemma 9.37 to $i = 1$. However, d can be also from $z'a$ and this case is investigated in Lemma 9.41.

Definition 9.39. For a word $w \in L(ID)$ ($u \Longrightarrow^* w$, $u \in A$) we construct the derivation tree of w iteratively as follows:

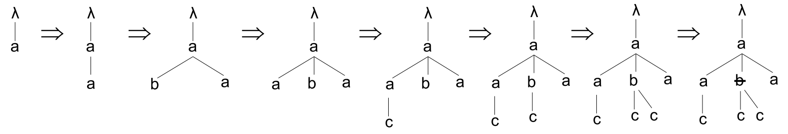
- Initially the tree has a root labeled by λ with children $a_1, \cdots, a_n$, where $u = a_1 \cdots a_n$. If $n = 1$, we can consider that the tree is rooted by a_1.
- For a transition $w'aw'' \Longrightarrow_{ins} w'abw''$ we consider the node corresponding to the letter a above and add as a left child a node labeled by symbol b.
- For a transition $w'abw'' \Longrightarrow_{del} w'aw''$ we consider the node corresponding to the letter b above and strike it out. In the future, this node is not considered anymore – it is treated like it is replaced it by its children (the corresponding links from the parent of b to all children should be added).

Having a derivation tree T for w, one can read w by concatenating labels of vertices from the preordering of T by a depth-first search. Hence, the root corresponds to the first letter of w and the rightmost label of the tree corresponds to the last letter of w. It is clear that there is a one-to-one correspondence between a derivation tree for an insertion-deletion system from the family $INS_1^{1,0}DEL_0^{0,0}$ and the derivation tree for the corresponding context-free grammar constructed as in Lemma 9.33.

Example 9.40. Let $ID = (\{a,b,c\}, \{a,b,c\}, \{a\}, I, D)$ with $I = \{(a,b,\lambda), (a,a,\lambda), (b,c,\lambda), (a,c,\lambda)\}$ and $D = \{(c,b,\lambda)\}$. We can derive $w = aaccca$ as follows:

$$a \Longrightarrow aa \Longrightarrow aba \Longrightarrow aaba \Longrightarrow aacba \Longrightarrow aacbca \Longrightarrow aacbcca \Longrightarrow aaccca$$

This corresponds to the following sequence of trees leading to the derivation tree of w.



The next lemma gives a bound on the depth of the derivation tree that has to be examined in order to compute the effect of deletion rules.

Lemma 9.41. *Consider an insertion-deletion system* $ID = (V,T,A,I,D)$ *of size* $(1,1,0;1,1,0)$. *Consider a word* $w \in L(ID)$ *and the corresponding derivation tree* T. *Now consider that during the construction of* T *a deletion rule* (c,x,λ) *will be applied. Denote the tree at this moment by* T'. *Denote by* b *the first common ancestor of deleting* c *and deleted* x *in* T' *and by* π *the path between* b *and the deleting* c *(including ends).*

If π *contains multiple occurrences of* c*, then the derivation of* w *may be rearranged such that the deletion of* x *is performed by the first occurrence of* c *in* π.

Proof. We remark that the above situation implies that x is a child of b and π is the rightmost path in the part of the tree rooted by b and ending before x, see Figure 9.1 (a). Now if π contains several occurrences of c, then we can rearrange the derivation of w as follows:

(1) Use same rules until the beginning of derivation of the first element from π.
(2) Derive π until the first c and the whole first subtree, see Figure 9.1(b).
(3) Delete x by this c, see Figure 9.1(c).
(4) Continue the derivation of π from the first c.

The obtained result is given on Figure 9.1(d), which is exactly what should have been obtained by the application of the deletion rule (c,x,λ) on the initial tree. □

Now we are ready to prove the main theorem of this section.

Theorem 9.42. $INS_1^{1,0}DEL_1^{1,0} \subset CF$.

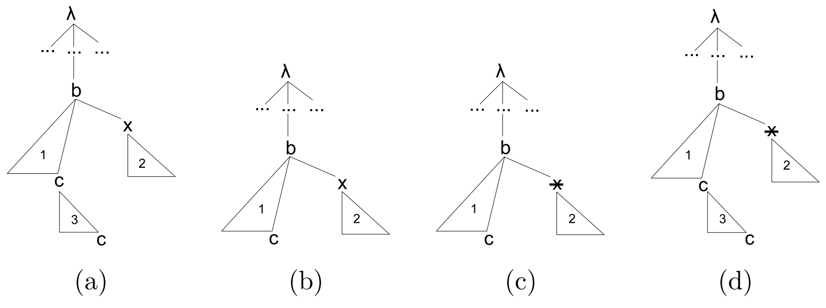


Fig. 9.1 The situation from Lemma 9.41: (a) the derivation tree before the application of the deletion rule (c, x, λ), (b) and (c) intermediate steps, (d) after the application.

Proof. Consider an insertion-deletion system $ID = (V, T, A, I, D)$ of size $(1, 1, 0; 1, 1, 0)$. By Lemmas 9.37 and 9.41 it is possible to restrict the application of deletion rules to all possible derivation subtrees that do not have repetition of letters in width (for any node, all its children are different) and height (any path from the root does not contain repetitions of letters). Since the number of such subtrees is finite, one can precompute all possible applications of deletion rules in them.

Consider a system $ID_1 = (V, T, A, I, \emptyset)$ and construct for it a context-free grammar $G_1 = (N, T, S, P)$ as in Lemma 9.33. Let $G_a = (N, T, S^a, P \cup \{S^a \to aS_a\})$. Now consider any restricted (in width and height) subtree τ rooted by $a \in V \cup \{\lambda\}$. Let w be the word corresponding to τ. Consider the derivation tree τ' of G_a corresponding to τ and eliminate the nodes labeled by λ and edges leading to these nodes. Denote the obtained tree by τ''. Let w'' be the sentence corresponding to τ''. It is clear that w'' is a marked variant of w, the marks S_x, $x \in V$, correspond to places where $I(x)^*$ can be inserted.

Now it is possible to compute the effect of deletion rules on w'' as follows.

$$D_0^\tau = \{z \mid w'' = az\}$$
$$D_{i+1}^\tau = \left\{uxS_tv \;\middle|\; uxzyS_tv \in D_i^\tau,\ (x, y, \lambda) \in D,\ t \in V,\ z \in \{S_a \mid a \in V\}^*\right\}$$
$$D^\tau = \bigcup_{i \geq 0} D_i^\tau$$

The above process is finite and D^τ contains strings corresponding all possible deletions that can be performed in τ. We define the set P_2 as follows:

$$P_2 = \{S_a \to w \mid a \in V,\ w \in D^\tau,\ \tau \text{ has the root } a\}.$$

Now consider the grammar $G = (N, T, S, P \cup P_2)$. From Lemma 9.37 and 9.41 and the construction above it is clear that G simulates ID, as productions from P permit to simulate insertion rules from I and those from P_2 permit to simulate deletion rules from D. The strictness of the inclusion follows from Theorem 9.28 (see also [21]), where it is shown that the language $(ba)^+$ cannot be generated by such systems. $\square$

Example 9.43. Let $ID = (T, T, \{a\}, I, D)$ with $T = \{a, b, c, d, d', e, e', f\}$, $I = \{(a, b, \lambda), (a, d, \lambda), (a, f, \lambda), (b, c, \lambda), (d, e, \lambda), (d, d', \lambda), (e, e', \lambda)\}$ and $D = \{(c, d, \lambda), (c, e, \lambda)\}$. Consider the tree τ corresponding to the word $abcdee'd'f$ and the derivation tree τ'' of G_1 corresponding to τ (see Figure 9.2).

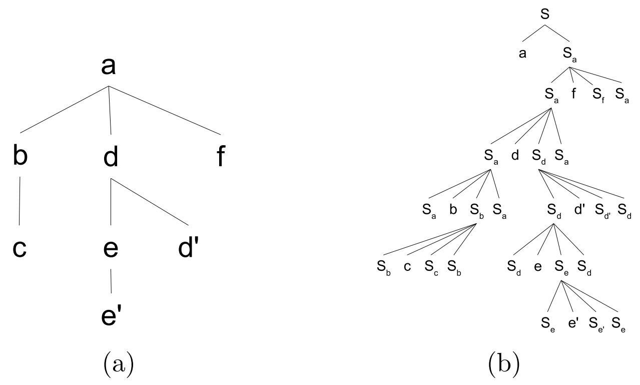


Fig. 9.2 The trees for the derivation of $abcdee'd'f$ from Example 9.43. The derivation in ID (a) and in G_1 (b).

Then we compute w'' and the application of rules from D to w'':

$$D_0^\tau = \{z \mid w'' = az\} = \{S_a b S_b c S_c S_b S_a d S_d e S_e e' S_{e'} S_e S_d d' S_{d'} S_d S_a f S_f S_a\}$$
$$D_1^\tau = \{S_a b S_b c S_d e S_e e' S_{e'} S_e S_d d' S_{d'} S_d S_a f S_f S_a\}$$
$$D_2^\tau = \{S_a b S_b c S_e e' S_{e'} S_e S_d d' S_{d'} S_d S_a f S_f S_a\}$$
$$D^\tau = D_2^\tau \cup D_1^\tau \cup D_0^\tau$$

Hence the following rules shall be added to the grammar G:

$$S_a \to S_a b S_b c S_d e S_e e' S_{e'} S_e S_d d' S_{d'} S_d S_a f S_f S_a$$
$$S_a \to S_a b S_b c S_e e' S_{e'} S_e S_d d' S_{d'} S_d S_a f S_f S_a$$

9.7 Pure Insertion Systems

In this section we consider systems which only use the operation of insertion, *i.e.*, the set of deletions is empty. In this section, we use the notation $INS_n^{m,m'}$ (*i.e.* we omit the DEL part) in order to denote families of languages generated by insertion-only systems. It is known that the classes of languages obtained by systems using only insertions (we call them *insertion languages*) are incomparable with many known language classes. As an example, consider a linear language $\{a^nba^n \mid n \geq 1\}$. This language cannot be generated by any insertion system (see Theorem 6.6 in [35]).

In order to overcome this obstacle, one can use some codings to "interpret" the generated strings. In the literature several types of codings were considered. It is possible to consider the following languages as a result for an insertion system I:

(1) $h(L(I) \cap R)$, where h is a morphism and R is a special language (as considered in [31, 34]), or
(2) $\varphi(h^{-1}(L(I)))$, where h is a morphism and φ is a weak coding (considered in [27, 35, 17]).

We mention that both types of codings are rather simple and can be simulated by a finite state transducer, provided that R is regular. In some cases we consider R to be the Dyck language. Thus, a formula of the same structure as the one in the Chomsky-Schützenberger theorem for context-free languages is obtained. Using this method, we present several characterizations of language classes from the Chomsky hierarchy in terms of insertion systems.

First, we recall several results on the computational power and decidability of small size insertion systems.

Theorem 9.44. [35] *The following statements hold.*

(1) $FIN \subset INS_*^{0,0} \subset INS_*^{1,1} \cdots \subset INS_*^{*,*} \subset CS$;
(2) $INS_*^{k,l} \subset INS_*^{k',l'}$, *where* $0 \leq k \leq k', 0 \leq l \leq l'$ *and either* $k < k'$ *or* $l < l'$;
(3) $INS_2^{2,2}$ *contains non-semilinear languages.*
(4) All families $INS_*^{n,m}, n, m \geq 0$, *are anti-AFLs.*
(5) REG *is incomparable with* $INS_*^{n,m}$ *for all* $n, m \geq 0$.
(6) CF, LIN *are incomparable with* $INS_*^{n,m}$ *for all* $n, m \geq 2$;
(7) $LIN - INS_*^{*,*} \neq \emptyset$;
(8) Each regular language is the coding of a language in $INS_*^{1,1}$.

It is possible to obtain the following representation of a regular language by using insertion systems and star languages. We recall that the family $STAR = \{A^* \mid A \in FIN\}$ of *star* languages is a subfamily of regular languages.

Theorem 9.45. [34] *Any regular language L can be represented in the form $L = h(L' \cap R)$, where h is a weak coding, $L' \in INS_2^{0,0}$, and R is a star language.*

Let W represent the family of weak codings. We mention that the inclusion $REG \subset W(INS_2^{0,0} \cap STAR)$ is proper, because the Dyck language is in $INS_2^{0,0}$.

Next, we consider several characterizations of context-free languages by means of insertion systems. The following theorem is from [35].

Theorem 9.46. $INS_*^{1,1} \subseteq CF$.

Proof. [Sketch] For an insertion system $\Pi = (T, T, A, I, \emptyset)$ consider a context-free grammar $G = (D, T, S, P)$ with nonterminal alphabet $D = \{D_{a,b} \mid a, b \in T \cup \{\lambda\}\}$ and the set of productions $P = P_1 \cup P_2 \cup P_3$, where

$$\begin{aligned} P_1 = \quad & \{S \to \delta(\lambda, w, \lambda) \mid w \in A\}, \\ P_2 = \quad & \{D_{a,b} \to a \mid D_{a,b} \in N, a, b \in T\}, \\ P_3 = \quad & \{D_{\overline{a_1},\overline{a_2}} \to \delta(\overline{a_1}, w, \overline{a_2}) \mid (a_1, w, a_2) \in I, \\ & \text{for } l = 1, 2 \;\; \overline{a_l} = a_l, \text{ if } a_l \neq \lambda \text{ and } \overline{a_l} \in V \cup \{\lambda\}, \text{ if } a_l = \lambda\}, \end{aligned}$$

where for every $a_1, a_2 \in T \cup \{\lambda\}, w \in V^*$ we denote by $\delta(a_1, w, a_2)$ the following

$$\delta(a_1, w, a_2) = \begin{cases} D_{a_1,a_2}, & \text{if } w = \lambda \\ D_{a_1,b_1} D_{b_1,b_2} \ldots D_{b_{k-1},b_k} D_{b_k,a_2}, & \text{if } w = b_1 \ldots b_k. \end{cases}$$

The rule $(a_1, b_1 \ldots b_k, a_2) \in I, a_1, a_2 \in T, b_1 \ldots b_k \in T^k$ can be simulated by the grammar iff the corresponding sentential form contains D_{a_1,a_2}. It is clear that nonterminals in D preserve one symbol left and right contexts. Hence, there is the following derivation $wD_{a_1,a_2}w' \Longrightarrow wD_{a_1,b_1}D_{b_1,b_2} \ldots D_{b_{k-1},b_k}D_{b_k,a_2}w'$. Clearly, the resulted string still preserves one symbol context. The simulation of rules that have no contexts is performed by the productions from P_3 with arbitrary contexts: $\overline{a_l} \in V \cup \{\lambda\}, l = 1, 2$.

The simulation starts by the production

$$S \to D_{\lambda,b_1} D_{b_1,b_2} \ldots D_{b_{k-1},b_k} D_{b_k,\lambda} \in P_1,$$

corresponding to the choice of an axiom from A. A terminal string is obtained by applying terminal rules P_2 at the end of derivation. Since there is a one-to-one correspondence between derivations in G and Π we obtain $L(\Pi) = L(G)$. Hence, $INS_*^{1,1} \subseteq CF$. $\square$

Now we give a characterization of context-free languages by the means of insertion systems that use one symbol contexts.

Theorem 9.47. [20] *A language L is context-free if and only if it can be represented in the form $L = \varphi(h^{-1}(L'))$ where $L' \in INS_3^{1,1}$, φ is a weak coding and h is a morphism.*

Proof. [Sketch] Taking into account Theorem 9.46 and the closure property of context-free languages with respect to inverse morphisms and weak codings, it is enough to show that for any context-free language L there is an insertion system Π having at most one-symbol contexts, such that $L = \varphi(h^{-1}(L(\Pi)))$, where h is a morphism, and φ is a weak coding.

Let $G = (N, T, S, P)$ be a context-free grammar in Chomsky normal form such that $L = L(G)$. Consider the following insertion system $\Pi = (V, V, I, \emptyset, \{S\})$, where $V = T \cup N \cup \{\#\}$, $I = \{(A, \#\gamma, \alpha) \mid \alpha \in T \cup V, A \to \gamma \in P, \gamma \in T \cup V^2\}$; the morphism h and the weak coding are defined as follows

$$h(a) = \begin{cases} a\#, & \text{if } a \in V\backslash(T \cup \{\#\}), \\ a & \text{if } a \in T, \end{cases} \qquad \varphi(a) = \begin{cases} a, & \text{if } a \in T, \\ \lambda & \text{if } a \in V\backslash T. \end{cases}$$

We claim that $L(\Pi) = L(G)$. Indeed, each rule $(A, \#\gamma, \alpha) \in R$ can be applied to the sentential form $wA\alpha w'$ iff A is unmarked (is not rewritten). Hence, a production $A \to \gamma \in P$ can be simulated by the corresponding rule in Π.

When every nonterminal is marked and no rules can be applied, the inverse morphism h^{-1} is applied to the output word. Indeed, if the system produces a word having some unmarked nonterminal then h^{-1} is not defined. At this point h^{-1} removes all marking symbols, and φ removes all nonterminal symbols. Hence, by applying the counterpart rules we get equivalent derivations and, hence, $L(\Pi) = L(G)$. $\square$

We present now several characterizations of recursively enumerable languages by the means of insertion systems. We recall a recent characterization of the family $INS_3^{3,3}$.

Theorem 9.48. [17, 30] *Each language $L \in RE$ can be written as $L = \varphi(h^{-1}(L'))$, where φ is a weak coding, h is a morphism, and $L' \in INS_3^{3,3}$.*

Proof. [Sketch] The idea of the proof is to apply "mark and migrate" technique in order to simulate a type-0 grammar. According to this technique, symbols that have been rewritten are marked. In the following a special symbol # called *marking symbol* will be used. We say that a letter a is *marked* in a sentential form waw' if it is followed by #, *i.e.*, $|w'| > 0$, and # is the prefix of w'. For example, in order to simulate a context-free production $A \to BC$, the string $\#BC$ is inserted immediately at the right of A, assuming that A was not marked before. As soon as the derivation of the simulated sentential form is completed, every nonterminal A is marked, and the inverse morphism is applied to the pairs $A\#$.

In order to simulate context-sensitive productions of the form $AB \to CD$, the *migration* of symbols is applied. This means that if a pair AB that should be used by the production is separated by one or more marked symbols, then copies of symbol A are inserted to the right, using the marked symbols as contexts. In this way, the symbol A can migrate to the right and become adjacent to B. When only the terminal symbols are unmarked in the resulted sentential form, the inverse morphism h^{-1} and the weak coding may be applied in order to eliminate marking symbols and nonterminals. More specifically, let $G = (N, T, S, P)$ be a grammar in Pentonnen normal form. Consider an insertion system $\Pi = (V, V, I_1 \cup I_2, \emptyset, \{\$S\$\$\$\})$, where $V = T \cup N \cup F \cup \overline{F} \cup \{\#, \overline{\#}, \$\}$, $F = \{F_A, \mid A \in N\}$. The sets of rules are

$$\begin{aligned} I_1 =& \{r_i.1.1 : (AB, \#C, \alpha) \mid i : AB \to AC \in P, \alpha \in V \backslash \{\#, \overline{\#}\}\} \cup \\ & \{r_i.1.2 : (A, \#BC, \alpha) \mid i : A \to BC \in P, \alpha \in V \backslash \{\#, \overline{\#}\}\} \cup \\ & \{r_i.1.3 : (A, \#\delta, \alpha) \mid i : A \to \delta \in P, \alpha \in V \backslash \{\#, \overline{\#}\}\}; \end{aligned}$$

$$\begin{aligned} I_2 =& \{r_A.2.1 : (AY\#, F_A, \alpha) \mid \alpha \in (N \cup \{\$\})V^2 \cup \overline{\#}V\#, Y \in N \cup F, A \in N\} \cup \\ & \{r_A.2.2 : (\alpha' A, \#\overline{\#}, Y\#F_A) \mid \alpha' \in V^2 \setminus \{A\overline{\#}\}, Y \in (N \cup F), A \in N\} \cup \\ & \{r_A.2.3 : (\overline{\#}Y\#, \overline{\#}, F_A) \mid Y \in N \cup F, A \in N\} \cup \\ & \{r_A.2.4 : (\overline{\#}F_A, \#A, \alpha) \mid \alpha \in V \setminus \{\#\}, A \in N\} \cup \\ & \left\{ r_A.2.5 : (A\overline{\#}, A, \alpha) \;\middle|\; \begin{array}{c} A \in N, \\ \alpha \in (N \backslash \{A\})(V \backslash \{\overline{\#}\})V \cup (N \cup F)\#(N \cup \{\overline{\#}\}) \end{array} \right\} \cup \\ & \{r_A.2.6 : (A, \#, \overline{\#}AY) \mid A \in N, Y \in \{V \backslash \{\#\}\}\}. \end{aligned}$$

The weak coding φ and the morphism h are defined as follows:

$$h(\alpha) = \begin{cases} \alpha\#, & \text{if } \alpha \in N \cup F, \\ \alpha, & \text{if } \alpha \in T \cup \{\overline{\#}, \$\}; \end{cases} \qquad \varphi(\alpha) = \begin{cases} \alpha, & \text{if } \alpha \in T, \\ \lambda, & \text{if } \alpha \in V \setminus T. \end{cases}$$

The set of rules I_1 simulates the rewriting rules, whereas the set I_2 simulates the migration of a symbol $A \in N$ to the right through the marked symbols. The migration works as following:

$$w\alpha' AY\#\alpha w' \Longrightarrow w\alpha' AY\#F_A\alpha w' \Longrightarrow w\alpha' A\#\overline{\#}Y\#F_A\alpha w' \Longrightarrow$$
$$w\alpha' A\#\overline{\#}Y\#\overline{\#}F_A\alpha w' \Longrightarrow w\alpha' A\#\overline{\#}Y\#\overline{\#}F_A\#A\alpha w'.$$

At the end of this sequence of steps, every symbol between α' and α' is marked, except A. One can verify that the only possible sequence of rules that may be applied is $r_A.2.1 - r_A.2.4$. It is easy to see that rules $r_A.2.5, r_A.2.6$ correspond to the migration of symbol A to the right of $\overline{\#}$. At the end of the derivation, a terminal string is obtained by applying h^{-1} and φ. Since there is one-to-one correspondence between derivations in G and Π, we obtain $\varphi(h^{-1}(L(\Pi)) = L(G)$. Hence, $W(V(INS_3^{3,3})) = RE$, where W and V are the family of weak codings and the family of inverse morphisms, respectively. $\square$

Let us note that in the theorem above, one can perform the migration of all terminal symbols at the end of the derivation (by a construction similar to the ones for the nonterminals) to the right hand side of the sentential form. So, all marked nonterminals appear as a prefix of the computed sentential form. Taking into account this remark and that the migration of symbols can be performed in both directions (by the mirrored rules), the following characterization of RE languages holds.

Corollary 9.49. [17] *Every language $L \in RE$ can be represented in either of the forms $L = L' \setminus R$, $L = L'/R'$, where $L' \in INS_3^{3,3}$, R, R' are regular languages, and $\backslash R, /R'$ denote the left and right quotient with R, R' respectively.*

In a similar way it is possible to obtain a characterization of RE by replacing the inverse morphism h^{-1} by the intersection with a regular language. It is shown in [31] that in order to obtain this characterization it is enough to use strictly $k-$testable languages (denoted by $LOC(k)$), which is a strictly subset of the family of regular languages, for $k \geq 2$. We recall that a language L is a strictly $k-$testable language over T if there are finite sets $Pref, Suf, Int \subseteq T^k$, and for every w, $w \in L$ if and only if

- the prefix of w of length k belongs to $Pref$,

- the suffix of w of length k belongs to Suf, and
- every proper subsequence of w of length k belongs to Int.

It is clear that an intersection with $LOC(2)$ permits the filtration of sentential forms that have following properties:

- every nonterminal is followed by a marking symbol, and
- every terminal is followed by a nonterminal symbol.

Thus, the following theorem holds.

Theorem 9.50. [31] *Every language $L \in RE$ can be represented in the form $h(L' \cap R)$, where h is a projection, $L' \in INS_3^{3,3}$, and $R \in LOC(2)$.*

The next theorem considers insertion systems with context-free rules. Since the *m*ark and migrate technique cannot be used in this case, the filtering of sentential forms that have the "proper structure" is performed by an intersection with the Dyck language. Formally, the theorem is stated as follows:

Theorem 9.51. [34] *Every language $L \in RE$ can be represented in the form $L = h(L' \cap \mathcal{D})$, where $L' \in INS_3^{0,0}, h$ is a projection, and $\mathcal{D}$ is the Dyck language.*

Proof. [Sketch] The proof of the theorem is based on a simulation of type-0 grammar (N, T, S, P) in Kuroda normal form by an insertion system. To simulate every context-sensitive production $r : AB \to CD$, a corresponding pair of rules is added to the insertion system: $(\lambda, \overline{BA}\overline{r}, \lambda)$, and (λ, CDr, λ). Respectively, to simulate a context-free production $r : A \to CD$, the following rules are added to the system: $(\lambda, \overline{A}\overline{r}, \lambda)$, and (λ, CDr, λ). In order to simulate the terminal productions $r : A \to a$, the following rules are added: $(\lambda, \overline{A}\overline{r}, \lambda)$, $(\lambda, a\overline{a}r, \lambda)$. Finally, the projection h is defined in a such way that it removes every nonterminal symbol at the end of derivation.

The argument for the proof of the statement is the following: since the sentential form at the end of derivation must be a word from $\mathcal{D}$ the insertions of each pair of rules corresponding to a production r must be done adjacently to the symbol(s) rewritten by production r. For example, consider a sentential form wAw' and a rule $r : A \to CD \in P$. The insertion system simulates the production as follows:

$$wAw' \Longrightarrow wA\overline{A}\overline{r}w' \Longrightarrow wCDrA\overline{A}\overline{r}w',$$

where $w, w' \in V^*$. In the case of a context-sensitive production $r : AB \to CD \in P$ we may have the following derivation:

$$wAw''Bw' \Longrightarrow wAw''B\overline{BA}\bar{r}w' \Longrightarrow wCDrAw''B\overline{BA}\bar{r}w',$$

where $w, w', w'' \in V^*, w' \in \mathcal{D}$. Hence, if the insertions are done at the proper positions, then the resulted string w coincides with the string from the grammar (supposing that h eliminates all nonterminals). The rigorous proof can be found in [34]. □

Let us note that if only context-free productions are considered in the proof of Theorem 9.51, then the rules corresponding to each context-free production $r : A \to BC$ appear adjacently to the left and right of A : $wAw' \Longrightarrow^* wCDrA\overline{A}rw'$. It is similar for terminal productions $r : A \to a$ where it can be obtained $wAw' \Longrightarrow wa\bar{a}rA\overline{A}rw'$. Hence, in order to obtain a context-free language, it is enough to consider the intersection with a language that only controls the appearance of parentheses that are placed nearby. This idea is used in the next theorem. The "only if" part of the theorem follows directly from Theorem 9.46 and from the closure of context-free languages with respect to weak codings and intersection with regular languages.

Theorem 9.52. [34, 31] *A language L is context-free if and only if it can be represented in the form $h(L' \cap R)$, where $L' \in INS_3^{0,0}$, h is a weak coding, and R is either from $STAR$ or from $LOC(4)$.*

9.8 Graph-Controlled Insertion-Deletion Systems

It was shown in previous sections that there are classes of insertion-deletion systems that cannot generate RE. Analogously to context-free grammars, a natural extension of insertion-deletion systems using the graph-controlled approach can be done. Such model introduces states (or labels of the program) associated to every insertion or deletion rule. The transition is performed by applying the corresponding rule and choosing the new state (thus the rule to be applied) among a specific set of rules. Another definition of this model can be made in the style of [33] or [6], see Subsection 9.2.2. This definition supposes that there are disjoint groups of insertion and deletion rules (corresponding to *membranes* from [33] or *components* from [6]). The transition is performed by first choosing and applying one of applicable rules from the current group and switching to the next group indicated in the rule description. Both definitions are equivalent. This is why we shall consider that in the subsequent text we use the second definition. We also remark that the last definition coincides with the definition of insertion-deletion P systems [33]. Moreover, all results on graph-controlled insertion-deletion systems were obtained under the name of insertion-deletion P systems.

We start with the following result.

Theorem 9.53. $PsStP_*(ins_1^{0,0}, del_1^{0,0}) = PsMAT$.

Proof. It is not difficult to see that dropping the requirement of the uniqueness of the instructions with the same label, the power of partially blind register machines does not change, see, e.g., [9]. We use this fact for the proof.

The inclusion $PsStP_*(ins_1^{0,0}, del_1^{0,0}) \subseteq PsMAT$ follows from the simulation of minimal context-free graph-controlled insertion-deletion systems by partially blind register machines, which are known to characterize $PsMAT$, see, e.g., [9]. Indeed, any rule $(\lambda, a, \lambda; q)_a \in R_p$ is simulated by instructions $p : (ADD(a), q)$. Similarly, a rule $(\lambda, a, \lambda; q)_e \in R_p$ is simulated by instructions $p : (SUB(a), q)$.

The output component i_0 is associated to the final state, while the halting is represented by the absence of the corresponding symbols (final zero-test) as follows. We assume that R_{i_0} has no insertion rules ($\emptyset$ can be generated by a trivial partially blind register machine), and the output registers correspond to those symbols that cannot be deleted by rules from R_{i_0}.

The converse inclusion follows from the simulation of partially blind register machines by graph-controlled insertion-deletion systems. Indeed, with every instruction p of the register machine we associate a component. Instruction $p : (ADD(A_k), q)$ is simulated by rule $(\lambda, A_k, \lambda; q)_a \in R_p$, and instruction $p : (SUB(A_k), q)$ by $(\lambda, A_k, \lambda; q)_e \in R_p$. Final zero-tests: rules $(\lambda, A_k, \lambda; \#)_e \in R_h$, $k \geq m$, should be inapplicable ($R_\# = \emptyset$). □

If the communication graph is a tree, one-way inclusion follows as a particular case. Non-extended systems are also a particular case.

Corollary 9.54. $PsSP_*(ins_1^{0,0}, del_1^{0,0}) \subseteq PsMAT$.

However, in terms of the generated languages such systems are not very powerful. Like in the case of context-free insertion-deletion systems there is no control on the position of insertion. Hence, the language $L = \{a^*b^*\}$

cannot be generated, even if the size of inserted strings is arbitrary. Hence we obtain:

Theorem 9.55. $REG \backslash LStP_*(ins_n^{0,0}, del_1^{0,0}) \neq \emptyset$*, for any* $n > 0$*.*

However, there are non-context-free languages that can be generated by such systems (even without deletion).

Theorem 9.56. $LStP_*(ins_1^{0,0}, del_0^{0,0}) \setminus CF \neq \emptyset$*.*

Proof. It is easy to see that the language $\{w \in \{a, b, c\}^* : |w|_a = |w|_b = |w|_c\}$ is generated by such a system with 3 nodes, inserting consecutively a, b and c. □

For the tree case the language $\{w \in \{a, b\}^* : |w|_a = |w|_b\}$ can be generated in a similar manner.

We show a more general inclusion:

Theorem 9.57. $ELStP_*(ins_n^{0,0}, del_1^{0,0}) \subset MAT$*, for any* $n > 0$*.*

Proof. [Sketch] Due to Lemma 9.2 we can suppose that there are no rules deleting the terminal symbols. Consider a graph-controlled insertion-deletion system $\Pi = (O, T, A, p_0, h, R_1, \ldots, R_n)$. Such a system can be simulated by a matrix grammar $G = (O \cup H, T, S, P)$ described below.

For an insertion instruction $(\lambda, a_1 \cdots a_n, \lambda; q)_a$ in component p, let P contain the matrix $\{p \to q, D \to Da_1D \cdots Da_nD\}$. For any deletion instruction $(\lambda, A, \lambda; q)_e$ in component p, let P contain the matrix $\{p \to q, A \to \lambda\}$. We also add to P three additional matrices: $\{h \to \lambda\}$, $\{D \to \lambda\}$ and $\{S \to p_0Da_1D \cdots Da_mD \mid w = a_1 \cdots a_m, w \in A\}$.

The above construction correctly simulates the system Π. Indeed, symbols D represent placeholders for all possible insertions. The first rule in the matrix simulates the navigation between components. □

Theorem 9.58. [22] $ELSP_5(ins_1^{1,0}, del_1^{1,0}) = RE$*.*

Proof. Let $G = (N, T, S, P)$ be a type-0 grammar in Kuroda normal form. We show that there is a system $\Pi \in ELSP_5(ins_1^{1,0}, del_1^{1,0})$ such that $L(\Pi) = L(G)$. Suppose that rules in P are ordered and $n = card(P)$. Consider a graph-controlled insertion-deletion system $\Pi = (V, T, \{SX\}, i_0, i_f, R_1, \cdots, R_5)$, where $V = N \cup T \cup \{P_1^i, P_2^i \mid 1 \leq i \leq 5\} \cup \{X\}$, the initial and the final components are $i_0 = i_f = 1$, and the communication graph has the (tree) structure presented on Figure 9.3.

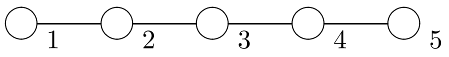


Fig. 9.3 Communication graph for Theorem 9.58.

For any context-sensitive production $i : AB \to CD \in P$ consider the rules:

$$\begin{aligned}
R_1^i =&\{r_i.1.1 : (\lambda, P_1^i, \lambda; 2)_a\};\\
R_2^i =&\{r_i.2.1 : (P_1^i, A, \lambda; 3)_e, \qquad r_i.2.2 : (\lambda, P_2^i, \lambda; 1)_e\};\\
R_3^i =&\{r_i.3.1 : (P_1^i, B, \lambda; 4)_e, \qquad r_i.3.2 : (P_2^i, C, \lambda; 2)_a\};\\
R_4^i =&\{r_i.4.1 : (P_1^i, P_2^i, \lambda; 5)_a, \qquad r_i.4.2 : (P_2^i, D, \lambda; 3)_a\};\\
R_5^i =&\{r_i.5.1 : (\lambda, P_1^i, \lambda; 4)_e\}.
\end{aligned}$$

For any context-free production $i : A \to BC \in P$ consider the rules

$$\begin{aligned}
R_1^i =&\{r_i.1.1 : (\lambda, P_1^i, \lambda; 2)_a\};\\
R_2^i =&\{r_i.2.1 : (P_1^i, A, \lambda; 3)_e, \qquad r_i.2.2 : (\lambda, P_2^i, \lambda; 1)_e\};\\
R_3^i =&\{r_i.3.1 : (P_1^i, X, \lambda; 4)_a, \qquad r_i.3.2 : (P_2^i, B, \lambda; 2)_a\};\\
R_4^i =&\{r_i.4.1 : (P_1^i, P_2^i, \lambda; 5)_a, \qquad r_i.4.2 : (P_2^i, C, \lambda; 3)_a\};\\
R_5^i =&\{r_i.5.1 : (\lambda, P_1^i, \lambda; 4)_e\}.
\end{aligned}$$

For any terminal production $i : A \to \alpha \in P, \alpha \in T$ consider the rules

$$\begin{aligned}
R_1^i =&\{r_i.1.1 : (\lambda, P_1^i, \lambda; 2)_a\};\\
R_2^i =&\{r_i.2.1 : (P_1^i, A, \lambda; 3)_e, \qquad r_i.2.2 : (\lambda, P_2^i, \lambda; 1)_e\};\\
R_3^i =&\{r_i.3.1 : (P_1^i, P_2^i, \lambda; 4)_a, \qquad r_i.3.2 : (P_2^i, \alpha, \lambda; 2)_a\};\\
R_4^i =&\{r_i.4.1 : (\lambda, P_1^i, \lambda; 3)_e\};\\
R_5^i =&\emptyset.
\end{aligned}$$

For every $\lambda-$production $i : A \to \lambda \in P$ consider the sets of rules

$$R_1^i = \{i.1.1 : (\lambda, A, \lambda; 1)_e, \}; R_2^i = R_3^i = R_4^i = R_5^i = \emptyset.$$

Now associate with the first component the set of rules $R_1 = \bigcup_{i=1}^n R_1^i \cup \{0 : (\lambda, X, \lambda; 1)_e\}$ and with the $k-$th component $k = 2, \dots, 5$ the set of rules $R_k = \bigcup_{i=1}^n R_k^i$. We claim that Π generates the same language as G.

We prove that every step of the derivation in G can be simulated in Π. Consider as example the production $i : AB \to CD \in R$. The simulation of this production is done as follows. Consider a string w_1ABw_2 in the component 1. Then, there is a following derivation

$$\begin{aligned}
&(1, w_1ABw_2) \Rrightarrow_{r_i.1.1} (2, w_1P_1^iABw_2) \Rrightarrow_{r_i.2.1} (3, w_1P_1^iBw_2) \Rrightarrow_{r_i.3.1}\\
&\Rrightarrow_{r_i.3.1} (4, w_1P_1^iw_2) \Rrightarrow_{r_i.4.1} (5, w_1P_1^iP_2^iw_2) \Rrightarrow_{r_i.5.1} (4, w_1P_2^iw_2) \Rrightarrow_{r_i.4.2}\\
&\quad\Rrightarrow_{r_i.4.2} (3, w_1P_2^iDw_2) \Rrightarrow_{r_i.3.2} (2, w_1P_2^iCDw_2) \Rrightarrow_{r_i.2.2} (1, w_1CDw_2).
\end{aligned}$$

The simulation of context-free, terminal and λ-productions can be done in a similar way.

It is easy to see that after insertion of P_1^i by the rule $r_i.1.1$, all rules corresponding to i-th production have to be applied until the rule $r_i.2.2$, otherwise the derivation will be blocked. We also note that every sentential form has at most one copy of symbols P_1^i and $P_2^i, i = 1, \ldots, n$ and none of them is present if the component 1 is active. We mention that one extra symbol $X \notin N \cup T$ appears when context-free productions are simulated. This is due to the fact that the total number of inserted and deleted symbols must be even. This symbol is finally deleted by deletion rule $0 : (\lambda, X, \lambda, 1)_d \in R_1$.

According to the definition of graph-controlled insertion-deletion systems the result of a computation consists of all strings over T which are obtained when the component 1 is active. Hence, there is a one-to-one correspondence between derivations in G and Π. Therefore $L(G) = L(\Pi)$. $\square$

A similar result can be obtained for the family $ELSP_5(ins_1^{1,0}del_1^{0,1})$.

Theorem 9.59. $ELSP_5(ins_1^{1,0}del_1^{0,1}) = RE$

Proof. [Sketch] We modify the construction from the previous theorem. Given a type-0 grammar G, consider the system Π as defined in Theorem 9.58. We perform the following changes to Π:

- for productions $AB \to CD$, replace $(P_1^i, A, \lambda; 3)_e$ in R_2 by $(\lambda, B, P_1^i; 3)_e$ and $(P_1^i, B, \lambda; 4)_e$ in R_3 by $(\lambda, A, P_1^i; 4)_e$,
- for other productions, replace $(P_1^i, A, \lambda; 3)_e$ in R_2 by $(\lambda, A, P_1^i; 3)_e$.

By applying the same argument as in the previous theorem we get the equivalence $L(\Pi) = L(G)$. $\square$

The symmetrical cases of the last two theorems also hold:

Corollary 9.60. $ELSP_5(ins_1^{0,1}, del_1^{0,1}) = ELSP_5(ins_1^{0,1}, del_1^{1,0}) = RE.$

Theorem 9.61. [23] $ELSP_4(ins_1^{1,0}, del_2^{0,0}) = RE.$

Proof. [Sketch] For a grammar $G = (N, T, S, P)$ in Pentonnen normal form, consider a system $\Pi = (V, T, \{SX\}, i_0, i_f, R_1, \ldots, R_4)$.

We define the nonterminal alphabet as $V = N \cup T \cup \{P_1^i, P_2^i, P_3^i \mid 1 \leq i \leq n\}$, where n is the number of productions in the grammar.

For every context sensitive production $i : AB \to AC \in P$ consider:

$$\begin{aligned}
R_1^i =& \{(A, P_1^i, \lambda; 2)_a\}; & & \\
R_2^i =& \{(P_1^i, P_2^i, \lambda; 3)_a, & & (\lambda, P_1^i P_3^i, \lambda; 1)_e\}; \\
R_3^i =& \{(\lambda, P_2^i B, \lambda; 4)_e, & & (P_3^i, C, \lambda; 2)_a\}; \\
R_4^i =& \{(P_1^i, P_3^i, \lambda; 3)_a\}; & &
\end{aligned}$$

For every context-free production $i : A \to BC \in P$ consider:

$$\begin{aligned}
R_1^i =& \{(A, P_1^i, \lambda; 2)_a\}; & & \\
R_2^i =& \{(P_1^i, C, \lambda; 3)_a, & & (\lambda, P_2^i, \lambda; 1)_e\}; \\
R_3^i =& \{(P_1^i, P_2^i, \lambda; 4)_a, & & (P_2^i, B, \lambda; 2)_a\}; \\
R_4^i =& \{(\lambda, AP_1^i, \lambda; 5)_e; & &
\end{aligned}$$

For every terminal production $i : A \to \alpha \in P$ consider:

$$\begin{aligned}
R_1^i =& \{(A, P_1^i, \lambda; 2)_a\}; & & \\
R_2^i =& \{(P_1^i, \alpha, \lambda; 3)_a, & & (\lambda, P_2^i P_3^i, \lambda; 1)_e\}; \\
R_3^i =& \{(P_1^i, P_2^i, \lambda; 4)_a, & & (P_2^i, P_3^i, \lambda; 2)_a\}; \\
R_4^i =& \{(\lambda, AP_1^i, \lambda; 3)_e\}; & &
\end{aligned}$$

For every $\lambda-$production $i : A \to \lambda \in P$ consider

$$R_1^i = \{(\lambda, A, \lambda; 1)_e\}; \quad R_2^i = R_3^i = R_4^i = \emptyset.$$

We associate with $k-$th component the set of rules $R_k = \bigcup_{i=1}^n R_k^i$. We mention that in this case the simulation of i-th production is controlled by P_1^i, P_2^i, P_3^i. Once a rule from R_1 is applied, inserting P_1^i for some $1 \leq i \leq n$, the only possible continuation of the derivation is to perform the sequence of rules corresponding to i-th production. We apply the same argument as in Theorem 9.58 in order to state the equivalence $L(G) = L(\Pi)$. □

Since RE is closed with respect to the mirror operation, by taking the mirror rules of Π we obtain:

Corollary 9.62. *$ELSP_4(ins_1^{0,1}, del_2^{0,0}) = RE$.*

By a similar straightforward simulation one can obtain the characterization of RE by exchanging the size of insertion and deletion rules.

Theorem 9.63. [23] *$ELSP_5(ins_2^{0,0}, del_1^{1,0}) = ELSP_5(ins_2^{0,0}, del_1^{0,1}) = RE$.*

Theorem 9.64. *$REG \setminus ELStP_*(ins_2^{0,0}, del_2^{0,0}) \neq \emptyset$.*

Proof. [Sketch] We show that $L_{ab} = \{a^*b\} \notin ELStP_k(ins_2^{0,0}, del_2^{0,0})$, for any $k \geq 1$. Assume the converse, and let Π be a graph-controlled insertion-deletion system having context-free rules that may insert or delete at most two symbols and that $L(\Pi) = L_{ab}$. By reasons similar to those in Lemma 9.11 and Lemma 9.13, it follows that for every finite derivation in Π one can construct a partition of rules $P_1 \cup \cdots \cup P_r, r \geq 1$ such that the overall effect of rules from each P_i, $i = 1, \cdots, r$ is the context-free insertion of at most two terminals. By taking a word of sufficient length, it is clear that some applications of rules from P_i which insert a or aa should be performed. Since the insertion is context-free, such an application can happen at the end of the word leading to a word having a preceded by b, causing a contradiction. $\square$

In the remaining part of the section we consider graph-controlled insertion systems without deletion rules.

First, we consider the Parikh image of graph-controlled context-free insertion systems. Let $SL_k, k \geq 1$ denote the set of all semi-linear sets of dimension k, and $SL = \bigcup_k SL_k$. There is an obvious relation between $StP_*(ins_n^{0,0}), n \geq 1$ and SL:

Theorem 9.65. *$PsStP_*(ins_n^{0,0}) = SL$.*

For the next theorems we consider the encodings of the generated languages in a similar way as in Section 9.8. In this way we establish equivalences between the classes of languages generated by graph-controlled insertion systems and the classes from the Chomsky hierarchy.

Theorem 9.66. [20] *Every language $L \in RE$ can be represented in the form $L = \varphi(h^{-1}(L'))$, where φ is a weak coding, h is a morphism, and $L' \in LSP_3(ins_2^{2,2})$.*

Proof. [Sketch] In order to prove the theorem we use "mark and migrate" technique explained in Section 9.7 for insertion systems. A simulation of type-0 grammars in the special Pentonnen normal form is performed. Let $G = (N, T, S, P)$ be such a grammar. Consider graph-controlled insertion-deletion system $\Pi = (V, T, \{S\$\}, i_0, i_f, R_1, R_2, R_3)$, where $V = T \cup N \cup F \cup \overline{F} \cup \{\#, \overline{\#}, \$\}$, $F = \{F_A, \mid A \in N\}$, $\overline{F} = \{\overline{F}_A, \mid A \in N\}$, and the initial and the final components are labeled by 1.

We assume that the communication graph of Π has the tree structure depicted in Figure 9.4.

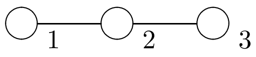


Fig. 9.4 Communication graph for Theorem 9.66.

The rules R_1, R_2, R_3 corresponding to the 1-st, the 2-nd, and the 3-rd component are defined as follows:

$$\begin{aligned}R_1 =&\{r_i.1.1 : (AB, \#C, \alpha; 1) \mid i : AB \to AC \in P, \alpha \in V\backslash\{\#, \overline{\#}\}\} \cup\\ &\{r_i.1.2 : (A, \#C, B\alpha; 1) \mid i : AB \to CB \in P, \alpha \in V\backslash\{\#, \overline{\#}\}\} \cup\\ &\{r_i.1.3 : (A, C, \alpha; 1) \mid i : A \to AC \in P, \alpha \in V\backslash\{\#, \overline{\#}\}\} \cup\\ &\{r_i.1.4 : (\lambda, C, A\alpha; 1) \mid i : A \to CA \in P, \alpha \in V\backslash\{\#, \overline{\#}\}\} \cup\\ &\{r_i.1.5 : (A, \#\delta, \alpha; 1) \mid i : A \to \delta \in P, \alpha \in V\backslash\{\#, \overline{\#}\}\}\cup\\ &\{r_A.1.6 : (A, \#F_A, \alpha; 2) \mid A \in N, F_A \in F, \alpha \in V\backslash\{\#, \overline{\#}\}\};\end{aligned}$$

$$\begin{aligned}R_2 =&\left\{\begin{array}{l|l} r_A.2.1 : (F_A, \#A, \alpha'; 1), & A \in N, F_A \in F, \overline{F_A} \in \overline{F},\\ r_A.2.2 : (\overline{F_A}, \overline{\#}A, \alpha'; 1) & \alpha' \in (N + T) \cdot (N + T + \$) + \$ \end{array}\right\} \cup\\ &\{r_A.2.3 : (F_AX, \#\overline{F_A}, \#; 3) \mid X \in F \cup N, \overline{F_B} \in \overline{F}, \alpha \in V\backslash\{\#, \overline{\#}\}\}\cup\\ &\{r_A.2.4 : (F_A\overline{F_B}, \overline{\#}F_A, \overline{\#}; 3), \qquad r_A.2.5 : (F_A\overline{\#}, F_A, \alpha; 3),\\ &r_A.2.6 : (\overline{F_A}\#, F_A, \alpha; 3), \qquad r_A.2.7 : (F_A\overline{\#}, F_A, \overline{\#}; 3),\\ &r_A.2.8 : (\overline{F_A}\#, F_A, \overline{\#}; 3), \qquad r_A.2.9 : (F_A\overline{\#}, \overline{F_A}, \#; 3),\\ &r_A.2.10 : (\overline{F_A}\#, \overline{F_A}, \#; 3)\};\end{aligned}$$

$$\begin{aligned}R_3 =&\{r_A.3.1 : (F_A, \#, \alpha; 2) \mid \alpha \in V, \alpha \neq \#\}\cup\\ &\{r_A.3.2 : (\overline{F_A}, \overline{\#}, \alpha'; 2) \mid \alpha' \in V, \alpha' \neq \overline{\#}\}.\end{aligned}$$

Consider also the morphism $h : V\backslash\{\#\} \to V$ and the weak coding $\varphi : V \to T \cup \{\lambda\}$ defined by:

$$h(a) = \begin{cases} a, & \text{if } a \in T,\\ a\# & \text{if } a \in V\backslash(T \cup \{\#\}) \end{cases} \qquad \varphi(a) = \begin{cases} a, & \text{if } a \in T,\\ \lambda & \text{if } a \in V\backslash T. \end{cases}$$

One may see that productions in P are in a one-to-one correspondence with the first five groups of insertion rules from R_1. Furthermore, the insertions performed by rules $r_i.1.1 - r_i.1.5$, $1 \leq i \leq |P|$, have the following properties:

- the rules can be only applied to the symbols that are not marked;
- the insertion marks the letter that is rewritten by the production.

Hence, for every derivation step in G a derivation step in Π can be considered (assuming that letters for context-sensitive productions are not separated by marking symbols).

Consider a pair of letters AB matching a production $AB \to AC$ or $AB \to CB \in P$. Suppose that this pair is separated by letters that have been marked. In this case, the rules from R_2 and R_3 are used in order to transfer a copy of letter $A \in N$ to the right-hand side of marked symbols. Indeed, every rule from $r_i.2.3 - r_i.2.10$ sees the next symbol to the right, and if it is marked, the rule inserts a copy of the symbol that have to be transferred to the right. For example consider the transfer of A in the string $AX\#C\$$ (here, we underline the inserted substrings)

$$(1, AX\#C\$) \Rrightarrow_{r.1.1} (2, A\underline{\#F_A}X\#C\$) \Rrightarrow_{r.2.3} (3, A\#F_AX\underline{\#\overline{F_A}}\#C\$) \Rrightarrow_{r.3.1}$$
$$(2, A\#F_A\underline{\#}X\#\overline{F_A}\#C\$) \Rrightarrow_{r.2.6} (3, A\#F_A\#X\#\overline{F_A}\underline{\#F_A}C\$) \Rrightarrow_{r.3.2}$$
$$(2, A\#F_A\#X\#\overline{F_A}\ \underline{\overline{\#}}\#F_AC\$) \Rrightarrow_{r.2.1} (1, A\#F_A\#X\#\overline{F_A}\ \overline{\#}\#F_A\underline{\#A}C\$).$$

The repetition of the 2-nd and the 3-rd active components makes a cycle until either the rule $r_i.2.1$ or the rule $r_i.2.2$ is applied. In this case, a copy of the symbol is inserted immediately at the left of either an unmarked nonterminal or a terminal symbol, or the rightmost mark.

We consider only those words obtained in Π where every nonterminal symbol have been marked, because otherwise the inverse morphism h^{-1} is not defined. This implies that every cycle happening in the 2-nd and the 3-rd components must be terminated. Finally, by applying the weak coding φ we eliminate every nonterminal and marking symbols.

Therefore, for every derivation in G one obtains a counterpart derivation in Π. This gives $L(G) = \varphi(h^{-1}(L(\Pi)))$. □

Let us consider graph-controlled insertion systems with left and right contexts of at most one symbol. This family characterizes the languages generated by context-free matrix grammars.

Theorem 9.67. [20] *A language L is in MAT if and only if it can be written in the form $L = \varphi(h^{-1}(L'))$, where $L' \in LSP_*(ins_2^{1,1})$, φ is a weak coding and h is a morphism.*

Proof. [Sketch] In order to prove the "only if" part of the claim, let us consider a language $L \in MAT$. Let $G = (N, T, S, M)$ be a matrix grammar in the binary normal form such that $L = L(G)$. Hence, we can assume that matrices in M are labeled by integers $1, \cdots, n$ and each matrix in M has the form $i : (A \to BC, A' \to B'C')$, where $A, A' \in N$ and $B, C, B', C' \in$

$N \cup T \cup \{\lambda\}$. Consider the following graph-controlled insertion system Π with nonterminal alphabet $V = N \cup T \cup \{\#\} \cup \{C_i, C_i' \mid i = 1 \cdots n\}$, the initial and the final component labeled by "1", initial string S, and the communication graph with the structure represented in Figure 9.5.

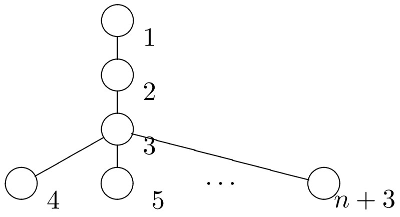


Fig. 9.5 Communication graph for Theorem 9.67.

Let $i : (A \to BC, A' \to B'C')$ be a matrix in M. Then consider the sets of rules of the size $(2, 1, 1)$ that correspond to the i-th production:

$$\begin{aligned}
R_1^i =&\{r_i.1.1 : (A, \#C_i, \alpha; 2) && \mid \alpha \in V\backslash\{\#\}\};\\
R_2^i =&\{r_i.2.1 : (C_i, BC, \alpha; 3), \quad r_i.2.2 : (C_i', \#, \alpha; 1) && \mid \alpha \in V\backslash\{\#\}\};\\
R_3^i =&\{r_i.3.1 : (C_i, \#, \alpha; i+3), \quad r_i.3.2 : (C_i', B'C', \alpha; 2) && \mid \alpha \in V\backslash\{\#\}\};\\
R_{i+3} =&\{r_{i+3}.4 : (A', \#C_i', \alpha; 3) && \mid \alpha \in V\backslash\{\#\}\}.
\end{aligned}$$

We associate with the k-th component $k = 1, 2, 3$ the set of rules $R_k = \bigcup_{i=1}^n R_k^i$, and with the k-th component $k = 4 \cdots n+3$ the set R_k.

Let h and φ be a morphism and a weak coding correspondingly defined as follows:

$$h(a) = \begin{cases} a, & \text{if } a \in T,\\ a\# & \text{if } a \in V\backslash(T \cup \{\#\}) \end{cases} \qquad \varphi(a) = \begin{cases} a, & \text{if } a \in T,\\ \lambda & \text{if } a \in V\backslash T. \end{cases}$$

We claim that $L(G) = \varphi(h^{-1}(L(\Pi)))$. Indeed, Π simulates productions of M in a direct way. Every sentential form contains at most one unmarked symbol from $\{C_i, C_i' \mid i = 1 \cdots n\}$. Whenever the rule $i.1.1$ is applied, the only possible derivation is to complete all the rules corresponding to the i-th production. Consider sentential form $w_1AwA'w_2$, where $w_1, w_2, w \in V^*$ and A, A' are not marked.

$$\begin{aligned}
&(1, w_1AwA'w_2) \Rrightarrow_{r_i.1.1} (2, w_1A\#C_iwA'w_2) \Rrightarrow_{r_i.2.1}\\
&(3, w_1A\#C_iBCwA'w_2) \Rrightarrow_{r_i.3.1} (4, w_1'A'w_2) \Rrightarrow_{r_{n+i}.4} (3, w_1'A'\#C_i'w_2) \Rrightarrow\\
&\qquad \Rrightarrow_{r_i.3.2} (2, w_1'A'\#C_i'B'C'w_2) \Rrightarrow_{r_i.4.2} (1, w_1'A'\#C_i'\#B'C'w_2),
\end{aligned}$$

where $w_1' = w_1A\#C_i\#BCw$. Hence, the derivation marks nonterminals A, A' and inserts BC, $B'C'$ to the right of $A\#$ and $A'\#$, respectively. At the end, by applying the inverse morphism and the weak coding we remove every marked nonterminal. Hence, we have $L(G) \subseteq \varphi(h^{-1}(L(\Pi)))$.

The inverse inclusion is obvious, because every rule in Π has its counterpart in G. Moreover, the case when the derivation in Π is blocked corresponds to the case in which the simulation of a matrix cannot be completed. Hence, the "only if" part of the theorem holds.

We mention that the family of languages MAT is closed with respect to inverse morphisms and weak codings. Hence, in order to prove the "if" part of the theorem it is enough to show that for any graph-controlled insertion system Π there is a matrix grammar G such that $L(\Pi) = L(G)$. We extend the construction of the context-free grammar used in Theorem 9.46 for the case of matrix grammars.

Let $\Pi = (V, V, A, i_0, i_f, R_1, \cdots, R_n)$ be an arbitrary graph-controlled insertion system. Consider the matrix grammar $G = (N, V, S, M)$ with nonterminal alphabet $N = \{Q_i \mid i = 1, \cdots, n\} \cup \{D_{a,b} \mid a, b \in V \cup \{\lambda\}\}$ and the set of matrices $M = M_1 \cup M_2 \cup M_3$ with

$$\begin{aligned}
M_1 =& \{(S \to Q_{i_0}\delta(\lambda, w, \lambda)) \mid w \in A\};\\
M_2 =& \{(D_{a,b} \to a) \mid D_{a,b} \in D, a, b \in T\} \cup \{(Q_{i_f} \to \lambda)\};\\
M_3 =& \{(Q_i \to Q_j, D_{\overline{a_1},\overline{a_2}} \to \delta(\overline{a_1}, w, \overline{a_2})) \mid (a_1, w, a_2; j) \in R_i,\\
& \quad \text{for } l = 1, 2 \ \ \overline{a_l} = \begin{cases} a_l, & \text{if } a_l \in V,\\ t, \forall t \in V \cup \{\lambda\}, & \text{if } a_l = \lambda \end{cases} \quad \};
\end{aligned}$$

where for every $a_1, a_2 \in V \cup \{\lambda\}, w \in V^*$ we denote by $\delta(a_1, w, a_2)$ the following

$$\delta(a_1, \lambda, a_2) = D_{a_1,a_2}, \ \delta(a_1, b_1 \cdots b_k, a_2) = D_{a_1,b_1} D_{b_1,b_2} \cdots D_{b_{k-1},b_k} D_{b_k,a_2}.$$

The simulation of Π by the matrix grammar is based on the encoding of pairs of adjacent letters by nonterminals from D. So, the encoded pair can be used as a context for an insertion rule. In addition, the label of the active component is represented by a nonterminal in Q. A rule $(a_1, b_1 \cdots b_k, a_2, j) \in R_i, a_1, a_2 \in V \cup \{\lambda\}, b_1 \cdots b_k \in V^k$ can be simulated by the grammar iff the sentential form contains both Q_i and D_{a_1,a_2}. As a result, the label representing the active component is rewritten to Q_j and D_{a_1,a_2} is rewritten to the string $D_{a_1,b_1} D_{b_1,b_2} \cdots D_{b_{k-1},b_k} D_{b_k,a_2}$. It is clear that the string preserves one symbol (left) context. In order to simulate rules that have no contexts we introduce productions with an arbitrary context: $\overline{a_l} \in V \cup \{\lambda\}, l = 1, 2$.

The simulation of Π by the grammar starts with a nondeterministic choice of an axiom from A. Then, during the derivation each rule from

$R_1, \cdots, R_n$ with the context (a_1, a_2) can be applied iff the productions having D_{a_1,a_2} in the left hand side can be applied. Finally, the string over V can be produced by the grammar as soon as Q_{i_f} is deleted from the sentential form. The deletion of Q_{i_f} specifies that Π activates the final component. As there is a one-to-one correspondence between derivations in G and Π, we obtain $L(\Pi) = L(G)$. Hence, $LSP_*(ins_*^{1,1}) \subseteq MAT$. □

9.9 Graph-Controlled Insertion-Deletion Systems with Priorities

In this section we present the results on context-free one-symbol graph-controlled insertion-deletion systems with priorities. Once a priority of deletion over insertion is introduced, $PsRE$ can be characterized, but in terms of language generation such systems cannot generate too much, because there is no control on the position of an inserted symbol. If one-sided contextual insertion or deletion rules are used, then this can be controlled and all recursively enumerable languages can be generated. The same result holds if a context-free deletion of two symbols is allowed. Most of the results in this section are from [2] and [3].

Minimal context-free graph-controlled insertion-deletion systems with priorities do generate $PsRE$. In case of general communication graph this is particularly easy to see: jumping to an instruction corresponds to switching to the associated component, and the entire construction is a composition of graphs shown in Figure 9.6. The decrement instruction works correctly because of priority of deletion over insertion.

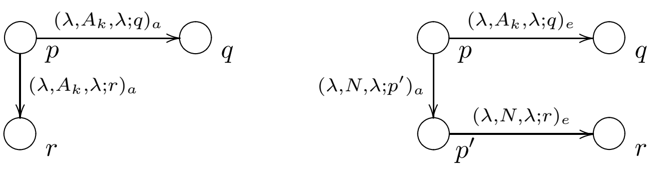

$(\lambda,A_k,\lambda;q)_a$
p
q
$(\lambda,A_k,\lambda;r)_a$
r
$(\lambda,A_k,\lambda;q)_e$
p
q
$(\lambda,N,\lambda;p')_a$
$(\lambda,N,\lambda;r)_e$
p'
r

Fig. 9.6 Simulating $p : (ADD(k), q, r)$(left) and $p : (SUB(k), q, r)$ (right).

For the tree-like communication graph, the proof is more sophisticated.

Theorem 9.68. $PsSP_*(ins_1^{0,0} < del_1^{0,0}) = PsRE$.

Proof. [Sketch] The proof is done by showing that for any register machine $\mathcal{M} = (d, Q, q_0, h, P)$ there is a graph-controlled insertion-deletion system $\Pi \in PsSP_*(ins_1^{0,0} < del_1^{0,0})$ with $Ps(\mathcal{M}) \subseteq Ps(\Pi)$. Then the existence of register machines generating $PsRE$ implies $PsRE \subseteq Ps(\Pi)$. The converse inclusion follows from the Church-Turing thesis.

Let Q_+ (Q_-) be the sets of labels of increment (conditional decrement, respectively) instructions of a register machine, and let $Q = Q_+ \cup Q_- \cup \{h\}$ represent all instructions. Consider a graph-controlled insertion-deletion system with the alphabet $O = Q \cup \{A_i \mid 1 \leq i \leq d\} \cup \{Y\}$ and the following structure (illustrated in Figure 9.7).

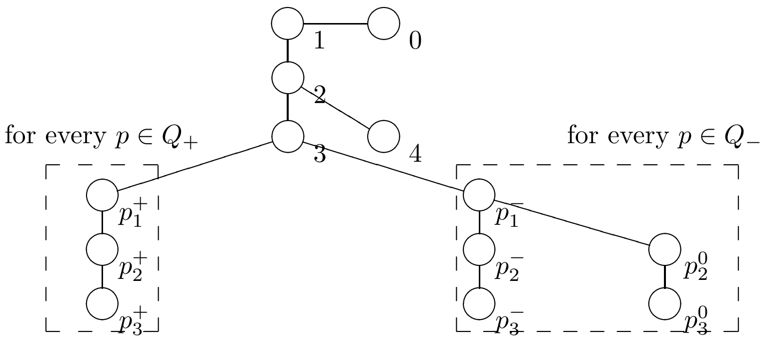


Fig. 9.7 Communication graph for Theorem 9.68. The structures in the dashed rectangles are repeated for every instruction of the register machine.

The system has initial string q_0, initial component 3, final component 0, and the following rules.

$$
\begin{aligned}
&R_1 = \{1 : (\lambda, Y, \lambda; 0)_e\},\\
&R_2 = \{2.1 : (\lambda, Y, \lambda; 1)_a\} \cup \{2.2 : (\lambda, Y, \lambda; 4)_e\},\\
&R_3 = \{3.1 : (\lambda, p, \lambda; p_1^+)_e \mid p \in Q_+\} \cup \{3.2 : (\lambda, p, \lambda; p_1^-)_e \mid p \in Q_-\}\\
&\quad \cup \{3.3 : (\lambda, Y, \lambda; 3)_e\} \cup \{3.4 : (\lambda, h, \lambda; 2)_e\},
\end{aligned}
$$

For any rule $p : (ADD(k), q, s)$, $R_{p_3^+} = \emptyset$ and

$$
\begin{aligned}
R_{p_1^+} = \{ \quad & a.1.1 :(\lambda, A_k, \lambda; p_2^+)_a, && a.1.2 :(\lambda, Y, \lambda; 3)_a\},\\
R_{p_2^+} = \{ \quad & a.2.1 :(\lambda, q, \lambda; p_1^+)_a, && a.2.1' :(\lambda, s, \lambda; p_1^+)_a,\\
& a.2.2 :(\lambda, q, \lambda; p_3^+)_e, && a.2.2' :(\lambda, s, \lambda; p_3^+)_e\},
\end{aligned}
$$

For any rule $p:(SUB(k),q,s)$, $R_{p_3^-}=R_{p_3^0}=\emptyset$ and

$$\begin{aligned}
R_{p_1^-}=\{\ & e.1.1:(\lambda,A_k,\lambda;p_2^-)_e, && e.1.2:(\lambda,Y,\lambda;p_2^-)_a, && e.1.3:(\lambda,Y,\lambda;3)_e\},\\
R_{p_2^-}=\{\ & e.2.1:(\lambda,q,\lambda;p_1^-)_a, && e.2.2:(\lambda,q,\lambda;p_3^-)_e,\\
& e.2.3:(\lambda,s,\lambda;p_3^-)_e, && e.2.4:(\lambda,Y,\lambda;p_2^-)_a\},\\
R_{p_2^0}=\{\ & e.3.1:(\lambda,s,\lambda;p_1^-)_a, && e.3.2:(\lambda,q,\lambda;p_3^0)_e, && e.3.3:(\lambda,s,\lambda;p_3^0)_e\}.
\end{aligned}$$

When component 3 is active, configurations $(p,x_1,\dots,x_n)$ of $\mathcal{M}$ are encoded by strings

$$Perm(pA_1^{x_1}\dots A_n^{x_n}Y^t),\ t\geq 0.$$

We say that such strings are in the *simulating* form. Clearly, in the initial configuration the string is already in the simulating form.

To prove that system Π correctly simulates $\mathcal{M}$, we prove the following claims:

(1) For any transition $(p,x_1\dots x_n)\Longrightarrow(q,x_1',\dots,x_n')$ in $\mathcal{M}$ there exists a computation in Π from the string $Perm(pA_1^{x_1}\dots A_n^{x_n}Y^t)$ and active component 3 to the string $Perm(qA_1^{x_1'}\dots A_n^{x_n'}Y^{t'})$, $t'\geq 0$ and active component 3 such that during this computation component 3 is not active at all intermediate steps and, moreover, this computation is unique.
(2) In any successful computation in Π (yielding a non-empty result), only strings of the above form exist when component 3 is active.
(3) The result $(x_1,\dots,x_n)$ in Π is obtained if and only if a string of form $Perm(hA_1^{x_1}\dots A_n^{x_n})$ appears when component 3 is active.

Now we prove each claim above. Consider a string $Perm(pA_1^{x_1}\dots A_n^{x_n}Y^t)$, $t\geq 0$ and active component 3 of Π. Assume that p is associated to the instruction $p:(ADD(k),q,s)\in P$. The only applicable rule in Π is from the group 3.1 (in the future we simply say rule 3.1) yielding the string $Perm(A_1^{x_1}\dots A_n^{x_n}Y^t)$ and active component p_1^+. After that, rule $a.1.1$ is applied yielding string $Perm(A_1^{x_1}\dots A_k^{x_k+1}\dots A_n^{x_n}Y^t)$ and active component p_2^+. Following that, one of rules $a.2.1$ or $a.2.1'$ is applied; then applying rule $a.1.2$ leads to one of strings $Perm(zA_1^{x_1}\dots A_k^{x_k+1}\dots A_n^{x_n}Y^{t+1})$, $z\in\{q,s\}$, in the simulating form.

Now suppose p is associated to the instruction $p:(SUB(k),q,s)\in P$. Then, the only applicable rule in component 3 is 3.2, which yields the string $Perm(A_1^{x_1}\dots A_n^{x_n}Y^t)$ and active component p_1^-. Now if $x_k>0$, then, due to the priority, rule $e.1.1$ will be applied, followed by application of rules $e.2.4$, $e.2.1$ and $e.1.3$, which yields the string

$Perm(qA_1^{x_1} \dots A_k^{x_k-1} \dots A_n^{x_n} Y^{t'})$ that is in the simulating form. If $x_k = 0$, then rule $e.1.2$ will be applied (provided that all symbols Y were previously deleted by rule 3.3), followed by rules $e.3.1$ and $e.1.3$, which leads to the string $Perm(sA_1^{x_1} \dots A_n^{x_n})$ that is in the simulating form.

To show that component 3 is not active during the intermediate steps, we prove the following property:

Property 9.69. For any component p_1^+ or p_1^-, if it is activated by component 3, it leads to activation of the "child" component (p_2^+, p_2^- or p_2^0), while if it is activated from the "child" component, it leads to activation of component 3.

Component p_1^+ or p_1^- can only be active if in the previous step symbol p was deleted by one of rules 3.1 or 3.2. If one of rules $a.1.2$ or $e.1.3$ is applied, then component 3 will be active and the string will be in the form $Perm(A_1^{x_1} \dots A_n^{x_n} Y^t)$, which cannot evolve anymore because all rules in component 3 imply the presence of some symbol from the set Q. Hence, a "child" component is activated. In the next step, the control will return from the "child" component by one of rules $a.2.1$, $a.2.1'$, $e.2.1$ or $e.3.1$ inserting a symbol from Q. If the string activates a "child" component again, then it will be sent to a trap component (p_3^+, p_3^- or p_3^0) by rules deleting symbols from Q. Hence the only possibility is to go to component 3 (a string that employed component p_2^- will additionally use rule $e.2.4$).

For the second claim, it suffices to observe that the property above ensures that when component 3 is active, only one symbol from Q can be present in the string.

The third claim holds since a string may activate component 2 if and only if the final label h of $\mathcal{M}$ appears while component 3 is active. Then, the string is checked for the absence of symbols Y by rule 2.2 (note that symbols Y can be erased by 3.3 of component 3) and sent to component 0 by rules 2.1 and 1.

By induction on the number of computational steps we obtain that Π simulates any computation in $\mathcal{M}$. Claims 1 and 2 imply that it is not possible to generate other strings, and Claim 3 implies that the same result is obtained. $\square$

We now show that small graph-controlled insertion-deletion systems with priority generate all recursively enumerable languages, at the price of either one-symbol context in either type of operation, or deletion of weight two.

Although Theorem 9.68 shows that the systems from the previous section are quite powerful, they cannot generate RE without control on the place where a symbol is inserted ($REG \setminus LStP_*(ins_n^{0,0} < del_1^{0,0}) \neq \emptyset$ for any $n > 0$, see Theorem 9.55). Once we allow a context in insertion or deletion rules, they can.

Theorem 9.70. $LSP_*(ins_1^{0,1} < del_1^{0,0}) = RE$.

Proof. [Sketch] It suffices to simulate register machines with WRITE instructions. We implement this instruction as an ADD instruction, except there is a special marker deleted at the end, and the "written" symbol has to be inserted to the left of the marker, as follows:

- Replace any write instruction $p : (WRITE(A), q, s)$, $A \in T$ of the machine by instructions $p : (ADD(A), q, s)$, considering output symbols A like new dummy registers. Construct the system Π as in Theorem 9.68.
- Change the initial string to q_0M;
- Replace rules $a.1.1$ $((\lambda, A, \lambda; p_2^+)_a \in R_{p_1^+})$ by $(\lambda, A, M; p_2^+)_a$ for $A \in T$;
- Add a new component s, make it final, and let $R_0 = \{(\lambda, M, \lambda; s)_e\}$,

It is easy to see that the above construction permits to correctly simulate the register machine with write instructions. □

Taking $\mathcal{M}$ in the left context yields the mirror language. Since RE is closed with respect to the mirror operation, the following corollary holds:

Corollary 9.71. $LSP_*(ins_1^{1,0} < del_1^{0,0}) = RE$.

A similar is proved if contextual deleting operation is allowed instead.

Theorem 9.72. $LSP_*(ins_1^{0,0} < del_1^{1,0}) = RE$.

Proof. [Sketch] As in Theorem 9.70, we use the construction from Theorem 9.68. However, additional components are needed to simulate the write instructions.

We modify the construction of Theorem 9.68 as follows. Let Q_s be the set of labels of WRITE instructions of a register machine. We add the following components (the modification of the communication graph shown in Figure 9.8):

As in Theorem 9.68, the initial string is q_0 and the starting component is 3. The system contains sets of rules R_1, R_2, $R_{p_1^+}, R_{p_2^+}, R_{p_3^+}$, $R_{p_1^-}, R_{p_2^-}, R_{p_3^-}$, $R_{p_2^0}, R_{p_3^0}$ defined as in Theorem 9.68. There are also following additional

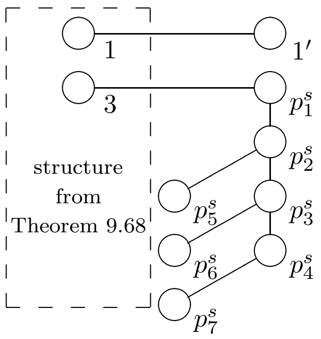


Fig. 9.8 Communication graph for Theorem 9.72.

rules for instructions $p : (WRITE(A), q)$ (the ruleset R'_3 shall be added to R_3).

$$\begin{aligned}
R_{1'} &= \{ \quad 1' : (\lambda, M, \lambda; 0)_e\}, \\
R'_3 &= \{ \quad 3.5 : (\lambda, p, \lambda; p_1^s)_e \mid p \in Q_s\}, \\
R_{p_1^s} &= \{ \quad w.1.1 : (\lambda, M, \lambda; p_2^s)_a, \qquad w.1.2 : (\lambda, M, \lambda; 3)_e\}, \\
R_{p_2^s} &= \{ \quad w.2.1 : (\lambda, M', \lambda; p_3^s)_a, \qquad w.2.2 : (\lambda, M', \lambda; p_1^s)_e\} \\
&\cup \{ \quad w.2.3 : (M, x, \lambda; p_5^s)_e \mid x \in O\}, \\
R_{p_3^s} &= \{ \quad w.3.1 : (\lambda, A, \lambda; p_4^s)_a, \qquad w.3.2 : (\lambda, Y, \lambda; p_2^s)_a\} \\
&\cup \{ \quad w.3.3 : (x, M, \lambda; p_6^s)_e \mid x \in O \setminus \{M', q\}\}, \\
R_{p_4^s} &= \{ \quad w.4.1 : (\lambda, q, \lambda; p_3^s)_a, \qquad w.4.2 : (M', M, \lambda; p_7^s)_e\}, \\
R_{p_5^s} &= \emptyset, \qquad R_{p_6^s} = \emptyset, \; R_{p_7^s} = \emptyset.
\end{aligned}$$

The WRITE instructions are simulated as follows. Suppose the configuration of register machine is $pA_1^{x_1} \dots A_d^{x_d}$ and the word $a_1 \dots a_n$ is written on the output tape. The corresponding simulating string in Π will be of form $p \sqcup\!\sqcup\, w$, where $w = Perm(A_1^{x_1} \dots A_d^{x_d} Y^t) \sqcup\!\sqcup\, a_1 \dots a_n$, $t \geq 0$. After the deletion of the state symbol p, a marker M is inserted in the string by rule $w.1.1$. If M is not inserted at the right end of the string, in the next step rule $w.2.3$ is applicable and the string activates the trap component p_5^s. In the next step symbol M' is inserted in the string. If it is not inserted before M, then the string is sent to region p_6^s by rule $w.3.3$. Hence, at this moment the contents of region p_3^s is $wM'M$. If rule $w.3.2$ is used, then the string $Y \sqcup\!\sqcup\, w$ activates component 3 and no rule is applicable anymore. Otherwise, symbol A is inserted by rule $w.3.1$. If it is not between M' and M, then rule $w.4.2$ is applicable and the string activates component p_7^s. After that, q is inserted between A and M, otherwise the trapping rule $w.3.3$ is applicable. At this moment, the configuration of the system consists of the string $w_tM'AqM$ and active component is p_3^s. Now if the rule $w.3.1$ is used, then the string activates the trap component by rule $w.4.1$. Otherwise, rule $w.3.2$ should be used, followed by the application of rules $w.2.2$ and $w.1.2$, leading to string $Y \sqcup\!\sqcup\, wAq$ and active component 3. Hence, symbol A is appended at the end of the string. At the end of the computation, all symbols Y are deleted when the string gets component $1'$ where the symbol M is further deleted and the string is sent to the final component. Hence, all symbols from $O - T$ are deleted and a word generated by $\mathcal{M}$ is obtained.

The converse inclusion $LSP_*(ins_1^{0,0} < del_1^{1,0}) \subseteq RE$ can be obtained from the Church-Turing thesis. □

Since RE is closed with respect to the mirror operation, it holds that

Corollary 9.73. *$LSP_*(ins_1^{0,0} < del_1^{0,1}) = RE$.*

Notice that the contextual deletion was used only to check for erroneous evolutions. It can be replaced by a context-free deletion of two symbols.

Theorem 9.74. *$LSP_*(ins_1^{0,0} < del_2^{0,0}) = RE$.*

Proof. We modify the proof of Theorem 9.72 as follows.

- Replace rules $w.2.3$ $((M, x, \lambda; p_5^s)_e \in R_{p_2^s})$ by rules $(\lambda, Mx, \lambda; p_5^s)_e$,
- Replace rules $w.3.3$ $((x, M, \lambda; p_6^s)_e \in R_{p_3^s})$ by rules $(\lambda, xM, \lambda; p_6^s)_e$,
- Replace rules $w.4.2$ $((M', M, \lambda; p_7^s)_e \in R_{p_4^s})$ by rules $(\lambda, M'M, \lambda; p_7^s)_e$.

The role of the new rules is the same as the role of the rules that were replaced. More exactly, the system checks whether two certain symbols are consecutive and if so, the string is blocked in a non-output region. □

We mention that the counterpart of Theorem 9.74 obtained by interchanging parameters of the insertion and deletion rules is not true, see Theorem 9.55.

9.10 Bibliographical Remarks

Insertion systems, without using the deletion operation, were first considered in [10], however the idea of the context adjoining was exploited long before by [25]. Context-free insertion systems as a generalization of concatenation were first considered in [11, 12]. A formal language study of both context-free insertion and deletion operations was done in [15], however the operations were considered separately. The articles [8, 13] investigate the power of the insertion and deletion operations. Both operations were first considered together in [18] and related formal language investigations can be found in several places; we mention only [27] and [32]. The biological motivation of insertion-deletion operations lead to their study in the framework of molecular computing, see, for example, [7], [16], [35], [38].

The universality of context-free insertion-deletion systems of size $(2,0,0;3,0,0)$ and $(3,0,0;2,0,0)$ was shown in [26], while the optimality of this result was shown in [40]. The latter article suggested considering the sizes of each context as a complexity measure and not the maximum as it was done before. One-sided insertion-deletion systems were first considered in [28] and the graph-controlled variant was considered in [22]. Graph-controlled insertion-deletion systems with priorities were introduced in [2].

Other variants of the insertion operation and different control mechanisms can be found in [15, 14, 5].

References

1. Alhazov, A., Freund, R. and Riscos-Núñez, A. (2005). One and two polarizations, membrane creation and objects complexity in P systems, in *SYNASC*, Vol. Seventh International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC 2005), 25-29 September 2005, Timisoara, Romania (IEEE Computer Society), pp. 385–394.
2. Alhazov, A., Krassovitskiy, A., Rogozhin, Y. and Verlan, S. (2009). P systems with minimal insertion and deletion, in *Proc. of Seventh Brainstorming Week on Membrane Computing, BWMC09*, pp. 9–22.
3. Alhazov, A., Krassovitskiy, A., Rogozhin, Y. and Verlan, S. (2010). P systems with minimal insertion and deletion, *Theor. Comput. Sci.*, 2010, (in publication).
4. Benne, R. (1993). *RNA Editing: The Alteration of Protein Coding Sequences of RNA* (Ellis Horwood, Chichester, West Sussex).
5. Biegler, F., Burrell, M. J. and Daley, M. (2007). Regulated rna rewriting: Modelling rna editing with guided insertion, *Theor. Comput. Sci.* **387**, 2, pp. 103 – 112, descriptional Complexity of Formal Systems.
6. Csuhaj-Varjú, E. and Dassow, J. (1990). On cooperating/distributed grammar systems, *Elektronische Informationsverarbeitung und Kybernetik* **26**, 1/2, pp. 49–63.
7. Daley, M., Kari, L., Gloor, G. and Siromoney, R. (1999). Circular contextual insertions/deletions with applications to biomolecular computation, in *SPIRE/CRIWG*, pp. 47–54.
8. Domaratzki, M. and Okhotin, A. (2004). Representing recursively enumerable languages by iterated deletion, *Theor. Comput. Sci.* **314**, 3, pp. 451–

457.

9. Freund, R., Ibarra, O., Păun, G. and Yen, H.-C. (2005). Matrix languages, register machines, vector addition systems, in *Third Brainstorming Week on Membrane Computing* (Sevilla), pp. 155–168.
10. Galiukschov, B. (1981). Semicontextual grammars, *Matem. Logica i Matem. Lingvistika* , pp. 38–50 Tallin University, (in Russian).
11. Haussler, D. (1982). *Insertion and Iterated Insertion as Operations on Formal Languages*, Ph.D. thesis, Univ. of Colorado at Boulder.
12. Haussler, D. (1983). Insertion languages, *Information Sciences* **31**, 1, pp. 77–89.
13. Ito, M., Kari, L. and Thierrin, G. (1997). Insertion and deletion closure of languages, *Theor. Comput. Sci.* **183**, 1, pp. 3–19.
14. Ito, M. and Sugiura, R. (2004). n-insertion on languages, in N. Jonoska, G. Paun and G. Rozenberg (eds.), *Aspects of Molecular Computing*, *Lecture Notes in Computer Science*, Vol. 2950 (Springer), pp. 213–218.
15. Kari, L. (1991). *On Insertion and Deletion in Formal Languages*, Ph.D. thesis, University of Turku.
16. Kari, L., Păun, G., Thierrin, G. and Yu, S. (1997). At the crossroads of DNA computing and formal languages: Characterizing RE using insertion-deletion systems, in *Proc. of 3rd DIMACS Workshop on DNA Based Computing* (Philadelphia), pp. 318–333.
17. Kari, L. and Sosík, P. (2008). On the weight of universal insertion grammars, *Theor. Comput. Sci.* **396**, 1-3, pp. 264–270.
18. Kari, L. and Thierrin, G. (1996). Contextual insertions/deletions and computability, *Inf. Comput.* **131**, 1, pp. 47–61.
19. Kleene, S. C. (1956). Representation of events in nerve nets and finite automata, in C. Shannon and J. McCarthy (eds.), *Automata Studies* (Princeton University Press, Princeton, NJ), pp. 3–41.
20. Krassovitskiy, A. (2009). On the power of insertion P systems of small size, in *Proc. of Seventh Brainstorming Week on Membrane Computing*, pp. 29–44.
21. Krassovitskiy, A., Rogozhin, Y. and Verlan, S. (2008a). Further results on insertion-deletion systems with one-sided contexts, in C. Martín-Vide, F. Otto and H. Fernau (eds.), *Language and Automata Theory and Applications, Second International Conference, LATA 2008, Tarragona, Spain, March 13-19, 2008. Revised Papers*, *Lecture Notes in Computer Science*, Vol. 5196 (Springer), pp. 333–344.
22. Krassovitskiy, A., Rogozhin, Y. and Verlan, S. (2008b). One-sided insertion and deletion: Traditional and P systems case, in E. Csuhaj-Varjú, R. Freund, M. Oswald and K. Salomaa (eds.), *International Workshop on Computing with Biomolecules, August 27th, 2008, Wien, Austria* (Druckerei Riegelnik), pp. 53–64.
23. Krassovitskiy, A., Rogozhin, Y. and Verlan, S. (2009). Computational power of P systems with small size insertion and deletion rules, in T. Neary, D. Woods, A. K. Seda and N. Murphy (eds.), *Proceedings International Workshop on The Complexity of Simple Programs, Cork, Ireland, 6-7th December 2008*, *EPTCS*, Vol. 1, pp. 108–117.

24. Krassovitskiy, A., Rogozhin, Y. and Verlan, S. (2010). Computational power of insertion-deletion (P) systems with rules of size two, *Natural Computing*, 2010, (in publication).
25. Marcus, S. (1969). Contextual grammars, *Rev. Roum. Math. Pures Appl.* **14**, pp. 1525–1534.
26. Margenstern, M., Păun, G., Rogozhin, Y. and Verlan, S. (2005). Context-free insertion-deletion systems, *Theor. Comput. Sci.* **330**, 2, pp. 339–348.
27. Martín-Vide, C., Paun, G. and Salomaa, A. (1998). Characterizations of recursively enumerable languages by means of insertion grammars, *Theor. Comput. Sci.* **205**, 1-2, pp. 195–205.
28. Matveevici, A., Rogozhin, Y. and Verlan, S. (2007). Insertion-deletion systems with one-sided contexts, in J. O. Durand-Lose and M. Margenstern (eds.), *Machines, Computations, and Universality, 5th International Conference, MCU 2007, Orléans, France, September 10-13, 2007, Proceedings*, *Lecture Notes in Computer Science*, Vol. 4664 (Springer), pp. 205–217.
29. Minsky, M. (1967). *Computation: Finite and Infinite Machines* (Prentice Hall, NJ, Englewood Cliffs), ISBN 0131655639.
30. Onodera, K. (2003). A note on homomorphic representation of recursively enumerable languages with insertion grammars, *Transactions of Information Processing Society of Japan* **44**, 5, pp. 1424–1427.
31. Onodera, K. (2009). New morphic characterizations of languages in Chomsky hierarchy using insertion and locality, in A. H. Dediu, A.-M. Ionescu and C. Martín-Vide (eds.), *Language and Automata Theory and Applications, Third International Conference, LATA 2009, Tarragona, Spain, April 2-8, 2009.*, *Lecture Notes in Computer Science*, Vol. 5457 (Springer), pp. 648–659.
32. Păun, G. (1997). *Marcus Contextual Grammars* (Kluwer Academic Publishers, Norwell, MA, USA), ISBN 0792347838.
33. Păun, G. (2002). *Membrane Computing. An Introduction* (Springer-Verlag).
34. Păun, G., Pérez-Jiménez, M. J. and Yokomori, T. (2008). Representations and characterizations of languages in Chomsky hierarchy by means of insertion-deletion systems, *Int. J. Found. Comput. Sci.* **19**, 4, pp. 859–871.
35. Păun, G., Rozenberg, G. and Salomaa, A. (1998). *DNA Computing: New Computing Paradigms* (Springer), ISBN 3540641963.
36. Rozenberg, G. and Salomaa, A. (eds.) (1997). *Handbook of Formal Languages* (Springer-Verlag, Berlin).
37. Smith, W. D. (1996). DNA computers in vitro and in vivo, in R. Lipton and E. Baum (eds.), *Proceedings of DIMACS Workshop on DNA Based Computers*, DIMACS Series in Discrete Math. and Theoretical Computer Science (Amer. Math. Society), pp. 121–185.
38. Takahara, A. and Yokomori, T. (2002). On the computational power of insertion-deletion systems, in M. Hagiya and A. Ohuchi (eds.), *DNA Computing, 8th International Workshop on DNA Based Computers, DNA8, Sapporo, Japan, June 10-13, 2002, Revised Papers*, *Lecture Notes in Computer Science*, Vol. 2568, pp. 269–280.
39. Takahara, A. and Yokomori, T. (2003). On the computational power of

insertion-deletion systems, *Natural Computing* **2**, 4, pp. 321–336.
40. Verlan, S. (2007). On minimal context-free insertion-deletion systems, *Journal of Automata, Languages and Combinatorics* **12**, 1-2, pp. 317–328.
41. Wood, D. (1987). *Theory of Computation* (Harper and Row).

Chapter 10

Accepting Networks of Evolutionary Word and Picture Processors: A Survey*

Florin Manea[1]
Faculty of Computer Science,
Otto-von-Guericke University of Magdeburg,
PSF 4120, D-39016 Magdeburg, Germany

Carlos Martín-Vide
Research Group on Mathematical Linguistics, Rovira i Virgili University,
Avinguda Catalunya 35, 43002, Tarragona, Spain,
E-mail: `carlos.martin@urv.cat`

Victor Mitrana
[1] *Faculty of Mathematics and Computer Science,*
University of Bucharest,
Str. Academiei 14, 70109 Bucharest, Romania,
E-mail: {`flmanea,mitrana`}`@fmi.unibuc.ro`

10.1 Introduction

The origin of the networks of evolutionary processors is twofold. A basic architecture for parallel and distributed symbolic processing, related to the Connection Machine [32] as well as the Logic Flow paradigm [23], consists of several processors, each of them being placed in a node of a virtual complete graph, which are able to handle data associated with the respective

*Florin Manea acknowledges support from the *Alexander von Humboldt Foundation.* Victor Mitrana's work was supported by the PIV Program of the *Agency for Management of University and Research Grants.*

node. Each node processor acts on the local data in accordance with some predefined rules, and then local data becomes a mobile agent which can navigate in the network following a given protocol. Only that data which is able to pass a filtering process can be communicated. This filtering process may require to satisfy some conditions imposed by the sending processor, by the receiving processor or by both of them. All the nodes send simultaneously their data and the receiving nodes handle also simultaneously all the arriving messages, according to some strategies, see [26, 32].

On the other hand, in [18] one considers a computing model inspired by the evolution of cell populations, which might model some properties of evolving cell communities at the syntactical level. Cells are represented by words which describe their DNA sequences. Informally, at any moment of time, the evolutionary system is described by a collection of words, where each word represents one cell. Cells belong to species and their community evolves according to mutations and division which are defined by operations on words. Only those cells are accepted as surviving (correct) ones which are represented by a word in a given set of words, called the genotype space of the species. This feature parallels with the natural process of evolution.

A network of language processors (see [19]) consists of several language identifying devices (language processors) associated with nodes of a virtual graph that rewrite words (representing the current state of the nodes) according to some prescribed rewriting mode and communicate the obtained words along the network using input and output filters defined by the membership condition to regular languages.

Similar ideas may be met in other bio-inspired models as *tissue-like membrane systems* [56] or models from Distributed Computing area like *parallel communicating grammar systems* [57].

In [13] the concept (considered from a formal language theory point of view in [19]) was modified in the following way inspired from cell biology. Each processor placed in a node is a very simple processor, an evolutionary processor. By an evolutionary processor we mean a processor which is able to perform very simple operations, namely point mutations in a DNA sequence (insertion, deletion or substitution of a pair of nucleotides). More generally, each node may be viewed as a cell having genetic information encoded in DNA sequences which may evolve by local evolutionary events, that is point mutations. Each node is specialized just for one of these evolutionary operations. Furthermore, the data in each node is organized in the form of multisets of words (each word appears in an arbitrarily large number of copies), and all copies are processed in parallel such that all the possible events that can take place do actually take place. Consequently, hybrid networks of evolutionary processors might be viewed as bio-inspired computing models. We want to stress from the very beginning that we are not concerned here with a possible biological implementation, though a matter of great importance. We are aware of the fact that modeling genetic evolutionary steps in this simple form is a demanding task requiring more than the systems surveyed in this paper.

A series of papers was devoted to different variants of this model viewed as language generating devices, see [1–5, 11, 14, 17, 22]. The work [48] is an early survey in this area.

The goal of this work is to survey the results reported so far regarding three variants of accepting networks of cell-like processors: networks of evolutionary processors, networks of splicing processors (both these processors act on words), and networks of evolutionary picture processors. The results surveyed here concern computational power, computational and descriptional complexity aspects, existence of universal networks, efficiency of these models viewed as problem solvers, and the relationships between them. It is questionable whether this approach gives something back to the evolutionary genetics studies, but we believe that it gives a little to the complexity theory. Characterizing **NP**, **P**, and **PSPACE** is not sufficient but the work surveyed here makes a step in this direction. It is worth mentioning that the general idea of the model is to show that very simple processors (based on pretty simple replacements) working synchronously in parallel and exchanging data to each other under a simple control mechanism (filters based on the symbol presence and absence) are able to efficiently simulate Turing machines and characterize complexity classes.

10.2 Basic Definitions

10.2.1 *Preliminaries*

We start by summarizing the notions used throughout the paper. An *alphabet* is a finite and nonempty set of symbols. The cardinality of a finite set A is written $card(A)$. Any sequence of symbols from an alphabet V is called word over V. The set of all words over V is denoted by V^* and the empty word is denoted by ε. The length of a word x is denoted by $|x|$ while $alph(x)$ denotes the minimal alphabet W such that $x \in W^*$.

We consider here the following definition of 2-tag systems that appears in [59]. It is slightly different but equivalent to those from [15, 58]. A 2-tag system $T = (V, \phi)$ consists of a finite alphabet of symbols V, containing a special *halting symbol* H and a finite set of rules $\phi : V \setminus \{H\} \to V^+$ such that $|\phi(x)| \geq 2$ or $\phi(x) = H$. Furthermore, $\phi(x) = H$ for just one $x \in V \setminus \{H\}$. A halting word for the system T is a word that contains the halting symbol H or whose length is less than 2; the transformation t_T (called the tag operation) is defined on the set of non-halting words as follows: if x is the leftmost symbol of a non-halting word w, then $t_T(w)$ is the result of deleting the leftmost 2 symbols of w and then appending the word $\phi(x)$ at the right end of the obtained word. A computation by a 2-tag system as above is a finite sequence of words produced by iterating the transformation t_T, starting with an initially given non-halting word w and halting when a halting word is produced. Note that a computation is not considered to exist unless a halting word is produced in finitely-many iterations. We recall that such restricted 2-tag systems are universal [59].

A nondeterministic Turing machine is a construct $M = (Q, V, U, \delta, q_0, B, F)$, where Q is a finite set of states, V is the input alphabet, U is the tape alphabet, $V \subset U$, q_0 is the initial state, $B \in U \setminus V$ is the "blank" symbol, $F \subseteq Q$ is the set of final states, and δ is the transition mapping, $\delta : (Q \setminus F) \times U \to 2^{Q\times(U\setminus\{B\})\times\{R,L\}}$. In this paper, we assume without loss of generality that any Turing machine we consider has a semi-infinite tape (bounded to the left) and makes no stationary moves; the computation of such a machine is described in [31, 51, 62]. An input word is accepted if and only if after a finite number of moves the Turing machine enters a final state. The language accepted by the Turing machine is a set of all accepted words. We say a Turing machine *decides* a language L if it accepts L and moreover halts on every input. The reader is referred to [27, 31, 51] for the classical time and space complexity classes defined for Turing machines.

In the course of its evolution, the genome of an organism mutates by different processes. At the level of individual genes the evolution proceeds by local operations (point mutations) which substitute, insert and delete nucleotides of the DNA sequence. In what follows, we define some rewriting operations that will be referred as *evolutionary operations* since they may be viewed as linguistic formulations of local gene mutations. We say that a rule $a \to b$, with $a, b \in V \cup \{\varepsilon\}$ is a *substitution rule* if both a and b are not ε; it is a *deletion rule* if $a \neq \varepsilon$ and $b = \varepsilon$; it is an *insertion rule* if $a = \varepsilon$ and $b \neq \varepsilon$. The set of all substitution, deletion, and insertion rules over an alphabet V are denoted by Sub_V, Del_V, and Ins_V, respectively.

Given a rule σ as above and a word $w \in V^*$, we define the following *actions* of σ on w:

- If $\sigma \equiv a \to b \in Sub_V$, then
  $\sigma^*(w) = \begin{cases} \{ubv : \ \exists u, v \in V^* \ (w = uav)\}, \\ \{w\}, \ \text{otherwise.} \end{cases}$

Note that a rule as above is applied to all occurrences of the letter a in different copies of the word w. An implicit assumption is that arbitrarily many copies of w are available.

- If $\sigma \equiv a \to \varepsilon \in Del_V$, then
  $\sigma^*(w) = \begin{cases} \{uv : \ \exists u, v \in V^* \ (w = uav)\}, \\ \{w\}, \ \text{otherwise} \end{cases}$
  $\sigma^r(w) = \begin{cases} \{u : \ w = ua\}, \\ \{w\}, \ \text{otherwise} \end{cases} \qquad \sigma^l(w) = \begin{cases} \{v : \ w = av\}, \\ \{w\}, \ \text{otherwise} \end{cases}$
- If $\sigma \equiv \varepsilon \to a \in Ins_V$, then
  $\sigma^*(w) = \{uav : \ \exists u, v \in V^* \ (w = uv)\}$,
  $\sigma^r(w) = \{wa\}, \qquad \sigma^l(w) = \{aw\}$.

Note that $\alpha \in \{*, l, r\}$ expresses the way of applying a deletion or insertion rule to a word, namely at any position ($\alpha = *$), in the left ($\alpha = l$), or in the right ($\alpha = r$) end of the word, respectively. The note for the substitution operation mentioned above remains valid for insertion and deletion at any position. For every rule σ, action $\alpha \in \{*, l, r\}$, and $L \subseteq V^*$, we define the *α-action of σ on L* by $\sigma^\alpha(L) = \bigcup_{w \in L} \sigma^\alpha(w)$. Given a finite set of rules M, we define the *α-action of M* on the word w and the language L by:

$$M^\alpha(w) = \bigcup_{\sigma \in M} \sigma^\alpha(w) \ \text{ and } \ M^\alpha(L) = \bigcup_{w \in L} M^\alpha(w),$$

respectively.

For two disjoint and nonempty subsets P and F of an alphabet V and a word z over V, we define the following two predicates

$$rc_s(z; P, F) \equiv P \subseteq alph(z) \ \wedge \ F \cap alph(z) = \emptyset$$
$$rc_w(z; P, F) \equiv alph(z) \cap P \neq \emptyset \ \wedge \ F \cap alph(z) = \emptyset.$$

The construction of these predicates is based on *context conditions* defined by the two sets P (*permitting contexts/symbols*) and F (*forbidding contexts/symbols*). Informally, both conditions requires that no forbidding symbol is present in w; furthermore the first condition requires all permitting symbols to appear in w, while the second one requires at least one permitting symbol to appear in w. It is plain that the first condition is stronger than the second one.

For every language $L \subseteq V^*$ and $\beta \in \{s, w\}$, we define:

$$rc_\beta(L, P, F) = \{z \in L \mid rc_\beta(z; P, F)\}.$$

An *evolutionary processor over* V is a 5-tuple (M, PI, FI, PO, FO), where:

– Either $(M \subseteq Sub_V)$ or $(M \subseteq Del_V)$ or $(M \subseteq Ins_V)$ holds. The set M represents the set of evolutionary rules of the processor. As one can see, a processor is "specialized" in one evolutionary operation, only.

– $PI, FI \subseteq V$ are the *input* permitting/forbidding contexts of the processor, while $PO, FO \subseteq V$ are the *output* permitting/forbidding contexts of the processor (with $PI \cap FI = \emptyset$ and $PO \cap FO = \emptyset$).

We denote the set of evolutionary processors over V by EP_V. Clearly, the evolutionary processor described here is a mathematical concept similar to that of an evolutionary algorithm, both being inspired from the Darwinian evolution. As we mentioned above, the rewriting operations we have considered might be interpreted as mutations and the filtering process described above might be viewed as a selection process. Recombination is missing but it was asserted that evolutionary and functional relationships between genes can be captured by taking only local mutations into consideration [63]. However, another type of processor based on recombination only, called splicing processor has been considered as well in a series of works which will be surveyed in the next sections.

10.2.2 *Accepting Networks of Evolutionary Processors*

An *accepting network of evolutionary processors* (ANEP for short) is an 8-tuple $\Gamma = (V, U, G, N, \alpha, \beta, x_I, x_O)$, where:

- V and U are the input and network alphabet, respectively, $V \subseteq U$.
- $G = (X_G, E_G)$ is an undirected graph without loops with the set of vertices X_G and the set of edges E_G. G is called the *underlying graph* of the network.
- $N : X_G \longrightarrow EP_U$ is a mapping which associates with each node $x \in X_G$ the evolutionary processor $N(x) = (M_x, PI_x, FI_x, PO_x, FO_x)$.
- $\alpha : X_G \longrightarrow \{*, l, r\}$; $\alpha(x)$ gives the action mode of the rules of node x on the words existing in that node.
- $\beta : X_G \longrightarrow \{s, w\}$ defines the type of the *input/output filters* of a node. More precisely, for every node, $x \in X_G$, the following filters are defined:

$$\text{input filter: } \rho_x(\cdot) = rc_{\beta(x)}(\cdot; PI_x, FI_x),$$
$$\text{output filter: } \tau_x(\cdot) = rc_{\beta(x)}(\cdot; PO_x, FO_x).$$

That is, $\rho_x(w)$ (resp. τ_x) indicates whether or not the word w can pass the input (resp. output) filter of x. Moreover, $\rho_x(L)$ (resp. $\tau_x(L)$) is the set of words of L that can pass the input (resp. output) filter of x.

- $x_I, x_O \in X_G$ are the *input* and the *output* node of Γ, respectively.

We say that $card(X_G)$ is the size of Γ. If α and β are constant functions, then the network is said to be *homogeneous.* In the theory of networks some types of underlying graphs are common like *rings, stars, grids*, etc. In most of the cases considered here, we focus on *complete* networks i.e., networks having a complete underlying graph; last section is an exception, as we discuss an incomplete ANEP that simulates a given ANEPFC (see the meaning of the abbreviation ANEPFC in the next subsection).

A *configuration* of an ANEP Γ as above is a mapping $C : X_G \longrightarrow 2^{V^*}$ which associates a set of words with every node of the graph. A configuration may be understood as the sets of words which are present in any node at a given moment. Given a word $w \in V^*$, the initial configuration of Γ on w is defined by $C_0^{(w)}(x_I) = \{w\}$ and $C_0^{(w)}(x) = \emptyset$ for all $x \in X_G \setminus \{x_I\}$.

When changing by an evolutionary step, each component $C(x)$ of the configuration C is changed in accordance with the set of evolutionary rules M_x associated with the node x and the way of applying these rules $\alpha(x)$. Formally, we say that the configuration C' is obtained in *one evolutionary step* from the configuration C, written as $C \Longrightarrow C'$, iff

$$C'(x) = M_x^{\alpha(x)}(C(x)) \text{ for all } x \in X_G.$$

When changing by a communication step, each node processor $x \in X_G$ sends one copy of each word it has, which is able to pass the output filter of x, to all the node processors connected to x and receives all the words sent by any node processor connected with x providing that they can pass its input filter. Formally, we say that the configuration C' is obtained in *one communication step* from configuration C, written as $C \vdash C'$, iff

$$C'(x) = (C(x) \setminus \tau_x(C(x))) \cup \bigcup_{\{x,y\} \in E_G} (\tau_y(C(y)) \cap \rho_x(C(y)))$$

for all $x \in X_G$. Note that words which leave a node are eliminated from that node. If they cannot pass the input filter of any node, they are lost.

Let Γ be an ANEP, the computation of Γ on the input word $w \in V^*$ is a sequence of configurations $C_0^{(w)}, C_1^{(w)}, C_2^{(w)}, \ldots$, where $C_0^{(w)}$ is the initial configuration of Γ, $C_{2i}^{(w)} \Longrightarrow C_{2i+1}^{(w)}$ and $C_{2i+1}^{(w)} \vdash C_{2i+2}^{(w)}$, for all $i \geq 0$. Note that the configurations are changed by the alternative application of evolutionary and communication steps. By the previous definitions, each configuration $C_i^{(w)}$ is uniquely determined by the configuration $C_{i-1}^{(w)}$. A computation *halts* (and it is said to be *halting*) if one of the following two conditions holds:

(i) There exists a configuration in which the set of words existing in the output node x_O is non-empty. In this case, the computation is said to be an *accepting computation.*

(ii) There exist two identical configurations obtained either in consecutive evolutionary steps or in consecutive communication steps.

The *language accepted* by the ANEP Γ is $L_a(\Gamma) = \{w \in V^* \mid$ the computation of Γ on w is an accepting one$\}$. We denote by $\mathcal{L}(ANEP)$ the class of languages accepted by ANEPs.

We say that an ANEP Γ decides the language $L \subseteq V^*$, and write $L(\Gamma) = L$ iff $L_a(\Gamma) = L$ and the computation of Γ on every $x \in V^*$ halts.

10.2.3 *Accepting Networks of Evolutionary Processors with Filtered Connections*

A model closely related to that of ANEPs, introduced in [25] and further studied in [24, 35], is that of *accepting networks of evolutionary processors with filtered connections* (ANEPFCs for short). An ANEPFC may be viewed as an ANEP where the filters are shifted from the nodes on the edges. Therefore, instead of having a filter at both ends of an edge on each direction, there is only one filter disregarding the direction.

An ANEPFC is a 9-tuple

$$\Gamma = (V, U, G, \mathcal{R}, \mathcal{N}, \alpha, \beta, x_I, x_O),$$

where:

- V, U, $G = (X_G, E_G)$, have the same meaning as for ANEP,
- $\mathcal{R} : X_G \longrightarrow 2^{Sub_U} \cup 2^{Del_U} \cup 2^{Ins_U}$ is a mapping which associates with each node *the set of evolutionary rules* that can be applied in that node. Note that each node is associated only with one type of evolutionary rules, namely for every $x \in X_G$ either $\mathcal{R}(x) \subset Sub_U$ or $\mathcal{R}(x) \subset Del_U$ or $\mathcal{R}(x) \subset Ins_U$ holds.
- $\alpha : X_G \longrightarrow \{*, l, r\}$; $\alpha(x)$ gives *the action mode of the rules* of node x on the words existing in that node.

- $\mathcal{N} : E_G \longrightarrow 2^U \times 2^U$ is a mapping which associates with each edge $e \in E_G$ *the permitting and forbidding filters of that edge*; formally, $\mathcal{N}(e) = (P_e, F_e)$, with $P_e \cap F_e = \emptyset$.
- $\beta : E_G \longrightarrow \{s, w\}$ defines *the filter type of an edge.*
- $x_I, x_O \in X_G$ are *the input and the output node* of Γ, respectively.

Note that every ANEPFC can be immediately transformed into an equivalent ANEPFC with a complete underlying graph by adding the edges that are missing and associate with them filters that do not allow any word to pass. Note that such a simplification is not always possible for ANEPs.

A configuration of an ANEPFC is defined in the same way as a configuration of an ANEP is defined above. An evolutionary step is also defined in the same way as above.

Differently, when changing by a communication step, in an ANEPFC, each node-processor $x \in X_G$ sends one copy of each word it contains to every node-processor y connected to x, provided they can pass the filter of the edge between x and y. It keeps no copy of these words but receives all the words sent by any node processor z connected with x providing that they can pass the filter of the edge between x and z. In this case, no word is lost.

The language accepted and decided by an ANEPFC is defined as in the case of ANEPs. We denote by $\mathcal{L}(ANEPFC)$ the class of languages accepted by ANEPFCs.

10.2.4 *Timed Accepting Networks of Evolutionary Processors*

The ANEP computing model was modified in [37] to obtain *timed accepting networks of evolutionary processors* (TANEP for short). Such a TANEP is a triple $\mathcal{T} = (\Gamma, f, b)$, where $\Gamma = (V, U, G, N, \alpha, \beta, x_I, x_O)$ is an ANEP, $f : V^* \to \mathbb{N}$ is a Turing computable function, called *clock*, and $b \in \{0, 1\}$ is a bit called the *accepting-mode bit.*

In this setting, the computation of a TANEP $\mathcal{T} = (\Gamma, f, b)$ on the input word w is the finite sequence of configurations of the ANEP Γ: $C_0^{(w)}, C_1^{(w)}, \ldots, C_{f(w)}^{(w)}$. The language accepted by $\mathcal{T}$ is defined as:

- if $b = 1$ then: $L(\mathcal{T}) = \{w \in V^* \mid C_{f(w)}^{(w)}(x_O) \neq \emptyset\}$
- if $b = 0$ then: $L(\mathcal{T}) = \{w \in V^* \mid C_{f(w)}^{(w)}(x_O) = \emptyset\}$

Intuitively we may think that a TANEP $\mathcal{T} = (\Gamma, f, b)$ is a triple that consists in an ANEP, a Turing Machine and a bit. For an input word w we first compute $f(w)$ on the tape of the Turing Machine (by this we mean that on the tape there exist $f(w)$ elements of 1, while the rest are blanks). Then Γ starts it computation, at each evolutionary or communication step of the network we delete an 1 from the tape of the Turing Machine. We stop when no 1 is found on the tape. Finally, according to the value of the accepting-mode bit and to the emptiness of $C_{f(w)}^{(w)}(x_O)$, we decide whether w is accepted or not.

10.2.5 *Computational Complexity Classes*

We define some computational complexity measures on the ANEP/ ANEPFC computing model. To this aim we consider an ANEP/ANEPFC Γ with the input alphabet V that halts on every input. The *time complexity* of the halting computation $C_0^{(x)}$, $C_1^{(x)}$, $C_2^{(x)}$, $\ldots C_m^{(x)}$ of Γ on $x \in V^*$ is denoted by $Time_\Gamma(x)$ and equals m. The time complexity of Γ is the function from $I\!N$ to $I\!N$,

$$Time_\Gamma(n) = \max\{Time_\Gamma(x) \mid x \in V^*, |x| = n\}.$$

In other words, $Time_\Gamma(n)$ delivers the maximal number of computational steps done by Γ on input words of length n.

For a function $f : I\!N \longrightarrow I\!N$ and $\mathcal{X} \in \{ANEP_k, ANEPFC_k \mid k \geq 1\}$, where $ANEP_k/ANEPFC_k$ denotes the class of all ANEPs/ANEPFCs of size k, we define:

$$\mathbf{Time}_\mathcal{X}(f(n)) = \{L \mid \text{there exists an network } \Gamma \text{ which is of type } \mathcal{X} \text{ and decides } L, \text{ and } n_0 \text{ such that } \forall n \geq n_0 (Time_\Gamma(n) \leq f(n))\}.$$

Moreover, we write $\mathbf{PTime}_\mathcal{X} = \bigcup_{k \geq 0} \mathbf{Time}_\mathcal{X}(n^k)$.

The *space complexity* of the halting computation $C_0^{(x)}$, $C_1^{(x)}$, $C_2^{(x)}$, $\ldots C_m^{(x)}$ of Γ on $x \in V^*$ is denoted by $Space_\Gamma(x)$ and is defined by the relation::

$$Space_\Gamma(x) = \max_{i \in \{1,\ldots,m\}} (\max_{z \in X_G} card(C_i^{(x)}(z))).$$

The space complexity of Γ is the function from $I\!N$ to $I\!N$,

$$Space_\Gamma(n) = \max\{Space_\Gamma(x) \mid x \in V^*, |x| = n\}.$$

Thus $Space_\Gamma(n)$ returns the maximal number of distinct words existing in a node of Γ during a computation on an input word of length n.

For a function $f : \mathbb{N} \longrightarrow \mathbb{N}$ and $\mathcal{X} \in \{ANEP_k, ANEPFC_k \mid k \geq 1\}$we define

$$\mathbf{Space}_{\mathcal{X}}(f(n)) = \{L \mid \text{there exists an network } \Gamma \text{ which is of type } \mathcal{X} \text{ and decides } L, \text{ and } n_0 \text{ such that } \forall n \geq n_0(Space_\Gamma(n) \leq f(n))\}.$$

Moreover, we write $\mathbf{PSpace}_{\mathcal{X}} = \bigcup\limits_{k\geq 0} \mathbf{Space}_{\mathcal{X}}(n^k)$.

The *length complexity* of the halting computation $C_0^{(x)}$, $C_1^{(x)}$, $C_2^{(x)}$, $\ldots C_m^{(x)}$ of Γ on $x \in L$ is denoted by $Length_\Gamma(x)$ and is defined by the relation:

$$Length_\Gamma(x) = \max_{w\in C_i^{(x)}(z), i\in\{1,\ldots,m\}, z\in X_G} |w|.$$

The length complexity of Γ is the function from $\mathbb{N}$ to $\mathbb{N}$,

$$Length_\Gamma(n) = \max\{Length_\Gamma(x) \mid x \in V^*, |x| = n\}.$$

Unlike the *Space* measure, $Length_\Gamma(n)$ computes the length of the longest word existing in a node of Γ during a computation on an input word of length n.

For a function $f : \mathbb{N} \longrightarrow \mathbb{N}$ and $\mathcal{X} \in \{ANEP_k, ANEPFC_k \mid k \geq 1\}$ we define $\mathbf{Length}_{\mathcal{X}}(f(n)) =\{L \mid$ there exists an network Γ which is of type $\mathcal{X}$ and decides L, and n_0 such that $\forall n \geq n_0(Length_\Gamma(n)\leq f(n))\}$.
Moreover, we write $\mathbf{PLength}_{\mathcal{X}} = \bigcup\limits_{k\geq 0} \mathbf{Length}_{\mathcal{X}}(n^k)$.

In the case of a TANEP $\mathcal{T} = (\Gamma, f, b)$ the time complexity definitions are the following: for the word $x \in V^*$ we define the time complexity of the computation on x as the number of steps that the TANEP makes having the word x as input, $Time_{\mathcal{T}}(x) = f(x)$. Consequently, we define the time complexity of $\mathcal{T}$ as a partial function from $\mathbb{N}$ to $\mathbb{N}$, that verifies: $Time_{\mathcal{T}}(n) = \max\{f(x) \mid x \in L(\mathcal{T}), |x| = n\}$. For a function $g : \mathbb{N} \longrightarrow \mathbb{N}$ we define:

$$\mathbf{Time}_{TANEP}(g(n)) = \{L \mid L = L(\mathcal{T}) \text{for a TANEP } \mathcal{T} = (\Gamma, f, 1) \text{ with } Time_{\mathcal{T}}(n) \leq g(n) \text{ for some } n \geq n_0\}.$$

Moreover, we write $\mathbf{PTime}_{TANEP} = \bigcup\limits_{k\geq 0} \mathbf{Time}_{TANEP}(n^k)$.

Note that the above definitions were given for TANEPs with the accepting-mode bit set to 1. Similar definitions are given for the case when the accepting-mode bit set to 0. For a function $f : \mathbb{N} \longrightarrow \mathbb{N}$ we define, as in the former case:

$$\mathbf{CoTime}_{TANEP}(g(n)) = \{L \mid L = L(\mathcal{T})\text{for a TANEP } \mathcal{T} = (\Gamma, f, 0) \text{ with } \\ Time_{\mathcal{T}}(n) \leq g(n) \text{ for some } n \geq n_0\}.$$

We define $\mathbf{CoPTime}_{TANEP} = \bigcup_{k\geq 0} \mathbf{CoTime}_{TANEP}(n^k)$.

10.3 Computational Power

10.3.1 *Computational Power of ANEPs/ANEPFCs*

The main results obtained so far state that nondeterministic Turing machines can be simulated by ANEPs and ANEPFCs. In other words, this shows that these computational models are *complete*. Our characterizations may be viewed a bit unfair if one considers that a computation of a nondeterministic Turing machine could be defined as a sequence of sets of IDs. In this setting, the machine may be viewed as computing deterministically. However, this is not a natural definition comparing to our approach: it is considered to be biologically feasible to have sufficiently many identical copies of a molecule. By techniques of genetic engineering, in a polynomial number of lab operations one can get an exponential number of identical molecules.

Theorem 10.1. [38, 44] *For every nondeterministic single-tape Turing machine M, accepting/deciding a language L, there exists an ANEP Γ, accepting/deciding the same language L. Moreover, if M works within $f(n)$ time, then $Time_\Gamma(n) \in \mathcal{O}(f(n))$, and if M works within $f(n)$ space, then $Length_\Gamma(n) \in \mathcal{O}(\max\{n, f(n)\})$.*

Theorem 10.2. [24] *For every nondeterministic single-tape Turing machine M, accepting/deciding a language L, there exists an ANEPFC Γ, accepting/deciding the same language L. Moreover, if M works within $f(n)$ time, then $Time_\Gamma(n) \in \mathcal{O}(f(n))$, and if M works within $f(n)$ space, then $Length_\Gamma(n) \in \mathcal{O}(\max\{n, f(n)\})$.*

It is worth mentioning that the proofs of both the above theorems are based on a common strategy: we simulate in parallel all the possible computations of the given nondeterministic Turing Machine. Moreover, in the both constructions the networks we design depend on the Turing Machine: that is, their size, their rules, their filters, are all defined according to the structure of the Turing Machine. In the following we will see how one can construct networks that accept a given language but do not depend in such a great measure on a device accepting that language. The main reason for which we mention these initial results is to show several techniques by which one can obtain completeness results for different computation models.

The reversal of both theorems hold as well:

Theorem 10.3. [25, 38] *For any ANEP/ANEPFC Γ accepting the language L, there exists a single-tape Turing machine M accepting L. Moreover, M can be constructed such that it accepts in $\mathcal{O}((Time_\Gamma(n))^2)$ computational time and in $\mathcal{O}(Length_\Gamma(n))$ space.*

Therefore we have:

Theorem 10.4.
1. $\mathcal{L}(ANEP)$ equals the class of recursively enumerable languages.
2. $\mathcal{L}(ANEPFC)$ equals the class of recursively enumerable languages.
3. $\mathbf{NP} = \mathbf{PTime}_{ANEP} = \mathbf{PTime}_{ANEPFC}$.
4. $\mathbf{PSPACE} = \mathbf{PLength}_{ANEP} = \mathbf{PLength}_{ANEPFC}$.

10.3.2 *The Role of Evolutionary Operations in ANEPs*

We denote by ANNIEP, ANNDEP, ANNSEP, ANIEP, ANDEP, and ANSEP the class of ANEPs without inserting nodes, without deletion nodes, without substitution nodes, with insertion nodes only, with deletion nodes only, with substitution nodes only, respectively. The corresponding classes of languages are denoted by $\mathcal{L}(ANNIEP)$, $\mathcal{L}(ANNdEP)$, $\mathcal{L}(ANNsEP)$, $\mathcal{L}(ANIEP)$, $\mathcal{L}(ANDEP)$, and $\mathcal{L}(ANSEP)$. In this section we ignore the empty word when we define a language and the empty set when we define a class of languages. In what follows we recall a few results regarding the computational power of some of these networks.

The results regarding the computational power of ANSEPs are summarized in the next theorem.

Theorem 10.5. [21]
1. Every language R that is commutative and semi-linear lies in $\mathcal{L}(ANSEP)$.
2. The class $\mathcal{L}(ANSEP)$ contains non-context-free languages even over the one-letter alphabet.
3. Every language in $\mathcal{L}(ANSEP)$ is polynomially recognizable.

The results regarding the computational power of ANDEPs are summarized in the next theorem.

Theorem 10.6. [21]
1. The class $\mathcal{L}(ANDEP)$ contains non-context-free languages.
2. A language over the unary alphabet is in $\mathcal{L}(ANDEP)$ if and only if it is one of these languages: $\{a\}$, $\{aa\}$, $\{a\}\{a\}^*$, or $\{aa\}\{a\}^*$.

The results regarding the computational power of ANIEPs are summarized in the next theorem.

Theorem 10.7. [21] *A language L over the alphabet V is in $\mathcal{L}(ANIEP)$ if and only if there are the subsets $V_1, V_2, \dots, V_n$ of V, for some $n \geq 1$, not necessarily pairwise disjoint, such that*

$$L = \bigcup_{i=1}^{n} \{x \in V_i^+ \mid |x|_a \geq 1, \text{ for all } a \in V_i\}.$$

The results regarding the computational power of ANNIEPs are summarized in the next theorem.

Theorem 10.8. ([20])
1. Every language in $\mathcal{L}(ANNIEP)$ is context-sensitive.
2. The class $\mathcal{L}(ANNIEP)$ contains all linear context-free languages and non-semi-linear languages.

10.3.3 *Complexity Classes and Size Complexity*

The last two statements in Theorem 10.4 were improved from the size complexity point of view: **NP** equals the class of languages accepted in polynomial time by ANEPs with 29 nodes and the class of languages accepted in polynomial time by ANEPFCs with 26 nodes (see [45, 24]). the ideas employed were quite simple. Instead of constructing a network that accepts the given language and works with the alphabet of that languages, we construct two networks: one that encodes the language in a special way over a binary alphabet, and one that accepts the encoded language. Combining these two modules one gets a constant size network for any recursively enumerable language. And this is because the encoding can be done by a very simple network (that simply rewrites the letters according to the mapping defined by a block code), and, since the size of the networks designed in the proof of Theorem 10.1 depended only on the size of the input alphabet, the encoded language can be accepted by a network of constant size. However, the rules and filters of the network still depend in great measure on the language that we want to accept.

We begin with the case of ANEPFCs.

Theorem 10.9. [24] *For any recursively enumerable (recursive) language L, accepted (decided) by a Turing machine $M = (Q, V, W, q_0, B, F, \delta)$, there exists an ANEPFC Γ, of size 26, accepting (deciding) L. Moreover,*
1. if $L \in NTIME(f(n))$ then $Time_\Gamma(n) \in \mathcal{O}(f(|W|n))$.
2. if $L \in NSPACE(f(n))$ then $Length_\Gamma(n) \in \mathcal{O}(f(|W|n))$.

A similar result can be obtained for ANEPs.

Theorem 10.10. [45] *For any recursively enumerable (recursive) language L, accepted (decided) by a Turing machine $M = (Q, V, W, q_0, B, F, \delta)$, there exists an ANEP Γ, of size 29, accepting (deciding) L. Moreover,*
1. if $L \in NTIME(f(n))$ then $Time_\Gamma(n) \in \mathcal{O}(f(|W|n))$.
2. if $L \in NSPACE(f(n))$ then $Length_\Gamma(n) \in \mathcal{O}(f(|W|n))$.

A consequence of Theorems 10.9 and 10.10 is the following characterization of **NP** and **PSPACE**:

Theorem 10.11. [24, 25, 38]
1. $\mathbf{NP} = \mathbf{PTime}_{ANEP_{29}} = \mathbf{PTime}_{ANEPFC_{26}}$.
2. $\mathbf{PSPACE} = \mathbf{PLength}_{ANEP_{29}} = \mathbf{PLength}_{ANEPFC_{26}}$.

The sizes presented in this theorem have been decreased in [36] and [35]. The results were obtained also via simulations of Turing machines, but based on different ideas. In the previous cases we had a network architecture which resembled more to a computer program based on different procedures, every node playing a well defined role (there were nodes used to check some conditions, nodes that simulated a move of the Turing Machine, etc.). In this case, while using the same basic idea of simulating in parallel all the computations of a Turing machine, the techniques used are quite different: the process is more intricate and it is based on a rotate-and-simulate technique.

Theorem 10.12. [35, 36]
1. $\mathbf{NP} = \mathbf{PTime}_{ANEP_{10}} = \mathbf{PTime}_{ANEPFC_{16}}$.
2. $\mathbf{PSPACE} = \mathbf{PLength}_{ANEP_{10}} = \mathbf{PLength}_{ANEPFC_{16}}$.

The next statements are immediate.

Theorem 10.13. [35, 36]
1. Every recursively enumerable language can be accepted by an ANEP of size 10.
2. Every recursively enumerable language can be accepted by an ANEPFC of size 16.

It is worth mentioning that the size in the first statement of this theorem is not optimal; in [2] one proves that ANEPs with 7 nodes can accept all recursively enumerable languages. However, this results cannot be extended to obtain networks that accept recursively enumerable languages as efficiently as the nondeterministic Turing machine do, as the proof in [2] is based on the simulation of a phrase-structure grammar. However, this result raises an interesting question: is there a trade-off between the size complexity of a network and the NEP-time complexity of a language? It remains also as an open problem to check if such a result can be obtained also for ANEPFCs.

Theorem 10.14. [2]
Every recursively enumerable language can be accepted by an ANEP of size 7.

One can also obtain a characterization of **P**, also based on the result of Theorem 10.1:

Theorem 10.15. [38] *A language $L \in \mathbf{P}$ if and only if L is decided by an ANEP/ ANEPFC Γ such that there exist two polynomials P, Q with $Space_\Gamma(n) \leq P(n)$ and $Time_\Gamma(n) \leq Q(n)$.*

It is worth mentioning that the last theorem does not say that the inclusion $\mathbf{PSpace}_{\mathcal{X}} \cap \mathbf{PTime}_{\mathcal{X}} \subseteq \mathbf{P}$ holds, for some $\mathcal{X} \in \{ANEP, ANEPFC\}$. The following facts are not hard to follow: we proved in Theorem 10.11 that every NP language, hence the NP-complete language 3-CNF-SAT, is in $\mathbf{PTime}_{\mathcal{X}}$; but, it is easy to see that 3-CNF-SAT can be decided also by a deterministic Turing Machine, working in exponential time and polynomial space. By Proposition 10.1, such a machine can be simulated by an ANEP/ANEPFC that uses polynomial space (but exponential time as well). This shows that 3-CNF-SAT is in $\mathbf{PTime}_{\mathcal{X}} \cap \mathbf{PSpace}_{\mathcal{X}}$, but it is not in **P**, unless $\mathbf{P} = \mathbf{NP}$.

Timed ANEPs (TANEPs) provide a framework in which both **NP** and **CoNP** can be characterized uniformly.

Theorem 10.16. [37]
1. $\mathbf{PTime}_{TANEP} = \mathbf{NP}$.
2. $\mathbf{CoPTime}_{TANEP} = \mathbf{CoNP}$.

Theorem 10.16 provides a common framework for accepting both languages from **NP** and from **CoNP**. For example, suppose that we want to accept a language L.

- If $L \in \mathbf{NP}$, then by the proof of Theorems 10.1, we can construct a polynomial TANEP $\mathcal{T} = (\Gamma, f, 1)$ that accepts L.
- If $L \in \mathbf{CoNP}$, it follows that $\mathbf{Co}L \in \mathbf{NP}$, and by the proofs of Theorems 10.1, we can construct a polynomial TANEP $\mathcal{T} = (\Gamma, f, 1)$ that accepts $\mathbf{Co}L$. We obtain that $(\Gamma, f, 0)$ accepts L.

Thus, Theorem 10.16 proves that the languages (the decision problems) that are efficiently recognized (respectively, solved) by the TANEPs, with both 0 and 1 as possible values for the accepting-mode bit, are those from $\mathbf{NP} \cup \mathbf{CoNP}$.

10.4 Universal ANEPs and ANEPFCs

In this section we recall a series of results regarding universal ANEPs and ANEPFCs. That is, ANEPs (or ANEPFCs) that can simulate the computation performed by any other ANEP (or ANEPFC, respectively) on a arbitrary word, assuming that the simulated network and its input word are given in an encoded form to the universal ANEP (respectively ANEPFC).

In [41], one describes a way of encoding an arbitrary ANEP/ANEPFC using the fixed alphabet:

$$A = \{\$, \#, r, l, *, (s), (w), 0, 1, 2, \bullet, \rightarrow\}.$$

Note that a similar encoding can be accomplished using the binary alphabet $A = \{0, 1\}$ only. However, in order to make the exposure more readable, this binary encoding is not used, though the results we are going to discuss in the sequel can be easily carried over the encodings over the binary alphabet.

The idea is to construct an ANEP Γ_U, such that if the input word of Γ_U is $< \Gamma >< w >$, for some ANEP Γ and word w the followings hold:

• Γ_U halts on the input $< \Gamma >< w >$ if and only if Γ halts on the input w.

• $< \Gamma >< w >$ is accepted by Γ_U if and only if w is accepted by Γ.

The first step of this construction is to define a Turing machine that behaves as described in the next theorem proved in [44].

Theorem 10.17. [41, 44] *There exists a deterministic Turing machine T_U, with the input alphabet A, satisfying the following conditions on any input $< \Gamma >< w >$, where Γ is an arbitrary ANEP and w is a word over the input alphabet of Γ:*

- *T_U halts on the input $< \Gamma >< w >$ if and only if Γ halts on the input w.*
- *$< \Gamma >< w >$ is accepted by T_U if and only if w is accepted by Γ.*

It is worth mentioning that the Turing machine T_U can be constructed such that its working alphabet is A plus a blank symbol.

From Theorem 10.1 it follows that one can construct an ANEP Γ_U that implements the same behavior as T_U. In this way we get a universal ANEP. We have shown:

Theorem 10.18. [41, 44] *There exists an ANEP Γ_U, with the input alphabet A, satisfying the following conditions on any input $< \Gamma >< w >$, where Γ is an arbitrary ANEP and w is a word over the input alphabet of Γ:*

- *Γ_U halts on the input $< \Gamma >< w >$ if and only if Γ halts on the input w.*
- *$< \Gamma >< w >$ is accepted by Γ_U if and only if w is accepted by Γ.*

Moreover, $size(\Gamma_U) = 5card(A) + 13$

Recall that A can be reduced to a binary alphabet, hence the size of Γ_U can be reduced to 23.

Using a similar technique as the above, one can encode ANEPFCs over a fixed alphabet A. In this setting one can prove the following result similar to Theorem 10.17.

Theorem 10.19. [24] *There exists a deterministic Turing machine T_U, with the input alphabet A, satisfying the following conditions on any input $< \Gamma >< w >$, where Γ is an arbitrary ANEPFC and w is a word over the input alphabet of Γ:*

- *T_U halts on the input $< \Gamma >< w >$ if and only if Γ halts on the input w.*
- *$< \Gamma >< w >$ is accepted by T_U if and only if w is accepted by Γ.*

As in the case of ANEPs, by Theorem 10.1, we derive the existence of a universal ANEPFC.

Theorem 10.20. [24] *There exists a ANEPFC* Γ_U*, of size* $2|A| + 14$*, with the input alphabet* A*, satisfying the following conditions on any input* $< \Gamma >< w >$*, where* Γ *is an arbitrary ANEPFC and* w *is a word over the input alphabet of* Γ*:*
- Γ_U *halts on the input* $< \Gamma >< w >$ *if and only if* Γ *halts on the input* w*.*
- $< \Gamma >< w >$ *is accepted by* Γ_U *if and only if* w *is accepted by* Γ*.*

Since A can be reduced to the binary alphabet, we obtain the existence of a universal ANEPFC of size 18.

The existence of an universal ANEP/ANEPFC can be used to prove the following result:

Theorem 10.21. 1. *For every recursively enumerable (recursive) language* L *one can effectively construct an ANEP* Γ*, with* $size(\Gamma) = 6|A| + 19$*, accepting (deciding)* L*.*
2. *For every recursively enumerable (recursive) language* L *one can effectively construct an ANEPFC* Γ*, with* $size(\Gamma) = 3|A| + 21$*, accepting (deciding)* L*.*
In the above, A *is the alphabet over which we encode NEPs, respectively NEPFCs, described above.*

The ANEP/ANEPFC constructed in this Theorem is not necessarily efficient from the time complexity, length complexity or from the size complexity point of view. Indeed, the computation of the network on w requires to run, as a subroutine, the universal corresponding network on an input encoding a network that accepts the given language and w. This may cause an exponential growth in the time, respectively length, needed to accept L. However, only a small number of the nodes of the ANEP/ANEPFC constructed in Theorem 10.21, for a particular recursively enumerable language L, effectively depend on that language. The rest of the nodes remain unchanged for all the possible languages L since they are basically a copy of the universal ANEP/ANEPFC. In a way, this fixes one of the drawbacks of the completeness results shown in the previous Section. Now we have shown that one can accept recursively enumerable languages in a rather uniform way: encode the language, run the universal network on the encoded words.

The results regarding the size of a universal ANEP/ANEPFC presented in the previous section can be improved significantly by simulating another universal computational model, the 2-tag systems. In this way we have obtained the following results (see [35, 36]).

Theorem 10.22. [36] *For every 2-tag system $T = (V, \phi)$ there exists a complete ANEP Γ of size 6 such that $L(\Gamma) = \{w \mid T \text{ halts on } w\}$.*

A similar result holds for ANEPFCs.

Theorem 10.23. [35] *For every 2-tag system $T = (V, \phi)$ there exists a complete ANEPFC Γ of size 10 such that $L(\Gamma) = \{w \mid T \text{ halts on } w\}$.*

Since 2-tag systems are universal [15, 59], the following theorem is immediate:

Theorem 1. [35, 36]
1. *There exists a universal ANEP with 6 nodes.*
2. *There exists a universal ANEPFC with 10 nodes.*

We remark that these results only show that for any recursively enumerable language one can construct a small ANEP/ANEPFC that accepts a special encoding of that language. Since 2-tag systems efficiently simulate deterministic Turing machines ([65]), the previous result also shows that ANEPs with 6 nodes and ANEPFCs with 10 nodes also simulate efficiently deterministic Turing machines.

10.5 A Direct Simulation

It is clear that filters associated with each node of an ANEP allow a strong control of the computation. However, by moving the filters from the nodes to the edges, the possibility of controlling the computation seems to be diminished. For instance, there is no possibility to lose data during the communication steps. In spite of this fact, we have seen that ANEPFCs are still computationally complete. This means that moving the filters from the nodes to the edges does not decrease the computational power of the model. Although the two variants are equivalent from the computational power point of view, a direct proof would have been worthwhile. In [7] it was shown that the two models can efficiently simulates each other, namely each computational step in one model is simulated in a constant number of computational steps in the other. This is particularly useful when one wants to translate the solution of a problem from one model into the other.

Note that a translation via a Turing machine, by the construction shown in the previous sections, squares the time complexity of the new solution.

The simulations presented in [7] may lead to underlying graphs of the simulating networks that differ very much from the underlying graphs of the simulated networks. Simulations preserving the type of the underlying graph of the simulated network represent a matter of interest which is not solved yet. Furthermore, the simulation of an ANEPFC that halts on every input leads to an ANEP that halts on every input as well. However, the other simulation does not preserves this property of the simulated network. It is known from the simulation of Turing machines by ANEPs and ANEPFCs presented in the previous sections that the languages decided by ANEPs can be also decided by ANEPFCs. It remains an open problem to modify the simulation of ANEPs by ANEPFCs such that the halting property is preserved.

10.6 Accepting Networks of Splicing Processors

In the case of accepting networks of splicing processors (ANSP for short), the point mutations associated with each node are replaced by the missing operation (recombination), which is present here in the form of splicing. This computing model is similar to some extent to the test tube distributed systems based on splicing introduced in [16] and further explored in [55]. However, there are several differences: first, the model proposed in [16] is a language generating mechanism while ours is an accepting one; second, we use a single splicing step, while every splicing step in [16] is actually an infinite process consisting of iterated splicing steps; third, each splicing step in our model is reflexive; fourth, the filters of our model are based on random context conditions while those considered in [16] are based on membership conditions; fifth, at every splicing step a set of auxiliary words, always the same and proper to every node, is available for splicing. Along the same lines, we want to stress the differences between this model and the time-varying distributed H systems, a generative model introduced in [54] and further studied in [46, 53, 52]. The computing strategy of such a system is that the passing of words from a set of rules to another one is specified by a cycle. Thus only those words that are obtained at one splicing step by using a set of rules are passed in a circular way to the next set of rules. This means that words which cannot be spliced at some step disappear from the computation while words produced at different splicing steps cannot be spliced together. Now, the differences between time-varying distributed H systems and ANSPs are evident: each node of an ANSP has a set of auxiliary words, words obtained at different splicing steps in different nodes can be spliced together, the communication of words is not done in a circular way, since identical copies of the same word are sent out to all the nodes, the control of communication is accomplished by filters.

A *splicing rule* over a finite alphabet V is a word of the form $u_1\#u_2\$v_1\#v_2$ such that u_1, u_2, v_1, and v_2 are in V^* and such that $\$$ and $\#$ are two symbols not in V.

For a splicing rule $r = u_1\#u_2\$v_1\#v_2$ and for $x, y, w, z \in V^*$, we say that r produces (w, z) from (x, y) (denoted by $(x, y) \vdash_r (w, z)$) if there exist some $x_1, x_2, y_1, y_2 \in V^*$ such that $x = x_1u_1u_2x_2$, $y = y_1v_1v_2y_2$, $z = x_1u_1v_2y_2$, and $w = y_1v_1u_2x_2$.

For a language L over V and a set of splicing rules R we define

$$\sigma_R(L) = \{z, w \in V^* \mid (\exists u, v \in L, r \in R)[(u, v) \vdash_r (z, w)]\}.$$

Similarly, for two languages L_1 and L_2, over V, and a set of splicing rules R we define

$$\sigma_R(L_1, L_2) = \{z, w \in V^* \mid (\exists u \in L_1, v \in L_2, r \in R)[(u, v) \vdash_r (z, w)]\}.$$

A *splicing processor* over V is a 6-tuple (S, A, PI, FI, PO, FO), where S a finite set of splicing rules over V, A a finite set of auxiliary words over V, and all the other parameters have the same meaning as in the definition of evolutionary processors. Now an ANSP can be defined similarly to an ANEP except that the processors associated with nodes are splicing processors.

A *configuration* of an ANSP Γ is a mapping $C : X_G \to 2^{U^*}$ which associates a set of words to every node of the graph. By convention, the auxiliary words do not appear in any configuration.

There are two ways to change a configuration, by a splicing step or by a communication step. When changing by a splicing step, each component $C(x)$ of the configuration C is changed according to the set of splicing rules S_x, whereby the words in the set A_x are available for splicing. Formally, configuration C' is obtained in one splicing step from the configuration C, written as $C \Rightarrow C'$, iff for all $x \in X_G$

$$C'(x) = \sigma_{S_x}(C(x) \cup A_x).$$

Since each word present in a node, as well as each auxiliary word, appears in an arbitrarily large number of identical copies, all possible splicings are assumed to be done in one splicing step. If the splicing step is defined as $C \Longrightarrow C'$, iff

$$C'(x) = \sigma_{S_x}(C(x), A_x) \text{ for all } x \in X_G,$$

then all processors of Γ are called *restricted* and Γ itself is said to be restricted.

A communication step and the language accepted/decided by an ANSP are defined in the same way to those for ANEP. The definitions of the complexity classes defined on ANEPs can be straightforwardly carried over ANSPs. On the other hand, accepting networks of splicing processors with filtered connections (ANSPFC) are defined similarly to ANEPFCs.

10.6.1 *Computational Power and Complexity Results for ANSPs/ANSPFCs*

The main result in [43] is:

Theorem 10.24. [43] *For any recursively enumerable language L, accepted by the Turing machine $M = (Q, V, U, \delta, q_0, B, F)$, there exists a ANSP (restricted or not) Γ such that $L(\Gamma) = L$ and $size(\Gamma) = 2card(U) + 2$.*

This is a key result in proving one of the main result in [42].

Theorem 10.25. [42] *For every recursively enumerable language L there exists an ANSP (restricted or not) Γ such that $L(\Gamma) = L$ and $size(\Gamma) = 2card(A) + 3$.*

We want to stress that only the rules in the node input node depend on the language L, and the encoding that we use for its symbols. The parameters of the other nodes do not depend in any way on the language L, on the encoding of the ANSP, or on the symbols of in the input alphabet of Γ. Finally, the size of the ANSP Γ proposed in the proof of this theorem can be decreased to 7.

As a consequence, we have

Theorem 10.26.
1. $\mathbf{NP} \subseteq \mathbf{PTime}_{ANSP_7} \cap \mathbf{PTime}_{ANSP_7^{(r)}}$.
2. $\mathbf{PSPACE} \subseteq \mathbf{PLength}_{ANSP_7} \cap \mathbf{PTime}_{ANSP_7^{(r)}}$.

A similar result is also valid for ANSPFCs (see [12]):

Theorem 10.27.
1. $\mathbf{NP} \subseteq \mathbf{PTime}_{ANSPFC_4} \cap \mathbf{PTime}_{ANSPFC_4^{(r)}}$.
2. $\mathbf{PSPACE} \subseteq \mathbf{PLength}_{ANSPFC_4} \cap \mathbf{PTime}_{ANSPFC_4^{(r)}}$.

The converse of this theorem is valid for restricted variants, namely

Theorem 10.28. [12]
1. $\mathbf{NP} = \mathbf{PTime}_{ANSPFC_4^{(r)}}$.
2. $\mathbf{PSPACE} = \mathbf{PLength}_{ANSPFC_4^{(r)}}$.

The size presented in Theorem 10.25 can be optimized as shown in [34]:

Theorem 10.29. *For any language L, accepted (decided) by a deterministic Turing Machine M, there exists an ANSP Γ, of size 2, accepting (deciding) L. Consequently, all recursively enumerable (recursive) languages are accepted (decided) by ANSPs of size 2.*

Since, by its definition, ANSPs need at least two nodes to accept any non-trivial language, these results go a long way in settling this issue, leaving, however, one open problem: the efficient simulation of nondeterministic Turing machines by ANSPs with two nodes.

Also the size reported in Theorem 10.24 has been reduced to an "almost" optimal value:

Theorem 10.30. [34] *All languages in **NP** can be decided by ANSPs of size 3 working in polynomial time.*

We finish this section by pointing out that the last part of [34] deals universal ANSPs. One shows how to construct a small universal ANSP and make several considerations on the computational efficiency of universal ANSPs.

10.7 Problem Solving with ANEPs/ANEPFCs

The results presented in the previous sections together with the fact that ANEPs, ANEPFCs and ANSPs are deterministic and computationally complete devices inspired from cell biology and amenable to be used as a problem solver.

Although the results in the previous sections state that every problem in **NP** can be solved in polynomial time using different variants of accepting networks, the results are obtained by simulating a nondeterministic Turing machine; thus we still have to obtain a classic solution to a problem, and then translate it in terms of ANEPs/ANEPFCs/ANSPs. To overtake this drawback, a series of papers discussed how ANEPs, ANEPFCs and ANSPs can be viewed as problem solvers.

Recall that a possible correspondence between decision problems and languages can be done via an encoding function which transforms an instance of a given decision problem into a word, see, e.g., [27]. We say that a decision problem P is solved in time $\mathcal{O}(f(n))$ by ANEPs/ANEPFCs/ANSPs if there exists a family $\mathcal{G}$ of ANEPs/ANEPFCs/ANSPs such that the following conditions are satisfied:

(1) The encoding function of any instance p of P having size n can be computed by a deterministic Turing machine in time $\mathcal{O}(f(n))$.

(2) For each instance p of size n of the problem one can effectively construct, in time $\mathcal{O}(f(n))$, an ANEP/ANEPFC/ANSP $\Gamma(p) \in \mathcal{G}$ which decides, again in time $\mathcal{O}(f(n))$, the word encoding the given instance. This means that the word is decided if and only if the solution to the given instance of the problem is "YES". This effective construction is called an $\mathcal{O}(f(n))$ time solution to the considered problem.

If an ANEP/ANEPFC/ANSP $\Gamma \in \mathcal{G}$ constructed above decides the language of words encoding all instances of the same size n, then the construction of Γ is called a uniform solution. Intuitively, a solution is uniform if for problem size n, we can construct a unique ANEP/ANEPFC/ANSP solving all instances of size n taking the (reasonable) encoding of instance as "input".

In [39] uniform linear time solutions using ANEPs to the 3-CNF-SAT and Hamiltonian Path problems are proposed; in [44] a uniform linear solution to the Vertex-Cover problem is proposed. In [25] one proposes another uniform linear time solution to the Vertex-Cover problem, solved this time by ANEPFCs. Uniform linear time solutions to the SAT and Hamiltonian Path problems with ANSPs and ANSPFCs are discussed in [40]

10.8 Accepting Networks of Picture Processors

10.8.1 *Pictures and Picture Languages*

Picture languages defined by different mechanisms have been studied extensively in the literature. A new model of recognizable picture languages, extending to two dimensions the characterization of the one-dimensional recognizable languages in terms of alphabetic morphisms of local languages, has been introduced in [28]. Similarly to the word case, there were proposed characterizations of recognizable picture series, see, e.g., [10, 50]. An early survey on automata recognizing rectangular pictures languages is [33], a bit more recent one considering different mechanisms defining picture languages, not necessarily rectangular, is [61] and an even more recent and concise one is [29]. Rather unexpected connections between different types of picture languages and logics were reported in [30, 49].

We recall here, following [9], the main results obtained for accepting networks whose nodes process pictures.

The definitions and notations concerning two-dimensional languages are taken from [29]. The set of natural numbers from 1 to n is denoted by $[n]$. The cardinality of a finite set A is denoted by $card(A)$. Let V be an alphabet, V^* the set of one-dimensional words over V and ε the empty word. A *picture* (or two-dimensional word) over the alphabet V is a two-dimensional array of elements from V. We denote the set of all pictures over the alphabet V by V^*_*, while the empty picture will be still denoted by ε. A two-dimensional language over V is a subset of V^*_*. The minimal alphabet containing all symbols appearing in a picture π is denoted by $alph(\pi)$. Let π be a picture in V^*_*; we denote the number of rows and the number of columns of π by $\overline{\underline{\pi}}$ and $|\pi|$, respectively. The pair $(\overline{\underline{\pi}}, |\pi|)$ is called *the size* of the picture π. The size of the empty picture ε is obviously $(0, 0)$. The set of all pictures over V of size (m, n), where $m, n \geq 1$, is denoted by V^n_m. The symbol placed at the intersection of the ith row with the jth column of the picture π, is denoted by $\pi(i, j)$. The row picture of size $(1, n)$ containing occurrences of the symbol a only is denoted by a^n_1. Similarly the column picture of size $(m, 1)$ containing occurrences of the symbol a only is denoted by a^1_m.

We recall informally the row and column concatenation operations between pictures. For a formal definition the reader is referred to [33] or [29]. The row concatenation of two pictures π of size (m, n) and ρ of size (m', n') is denoted by $\pi \circledR \rho$ and is defined only if $n = n'$. The picture $\pi \circledR \rho$ is obtained by adding the picture ρ after the last row of π. Analogously one defines the column concatenation denoted by ⓒ. We now define four new operations, in some sense the inverse operations of the row and column concatenation. Let π and ρ be two pictures of size (m, n) and (m', n'), respectively. We define

- The column right-quotient of π with ρ: $\pi/_{\rightarrow}\rho = \theta$ iff $\pi = \theta \textcircled{c} \rho$.
- The column left-quotient of π with ρ: $\pi/_{\leftarrow}\rho = \theta$ iff $\pi = \rho \textcircled{c} \theta$.
- The row down-quotient of π with ρ to the right: $\pi/_{\downarrow}\rho = \theta$ iff $\pi = \theta \textcircled{R} \rho$.
- The column up-quotient of π with ρ: $\pi/_{\uparrow}\rho = \theta$ iff $\pi = \rho \textcircled{R} \theta$.

10.8.2 *Picture Processors*

Let V be an alphabet; a rule of the form $a \to b(X)$, with $a, b \in V \cup \{\varepsilon\}$ and $X \in \{-, |\}$ is called an *evolutionary rule*. For any rule $a \to b(X)$, X indicates which component of a picture (row if $X = -$ or column if $X = |$) the rule is applied to. We say that a rule $a \to b(X)$ is a *substitution rule* if both a and b are not ε, is a *deletion rule* if $a \neq \varepsilon$, $b = \varepsilon$, and is an *insertion rule* if $a = \varepsilon$, $b \neq \varepsilon$. In this paper we shall ignore insertion rules because we want to process every given picture in a space bounded by the size of that picture. We denote by $RSub_V = \{a \to b(-) \mid a, b \in V\}$ and $RDel_V = \{a \to \varepsilon(-) \mid a \in V\}$. The sets $CSub_V$ and $CDel_V$ are defined analogously.

Given a rule σ as above and a picture $\pi \in V_m^n$, we define the following *actions* of σ on π:

• If $\sigma \equiv a \to b(|) \in CSub_V$, then

$$\sigma^{\leftarrow}(\pi) = \begin{cases} \{\pi' \in V_m^n : \exists i \in [m](\pi(i,1) = a \ \&\ \pi'(i,1) = b), \pi'(k,1) = \pi(k,1), \\ k \in [m] \setminus \{i\}, \pi'(j,l) = \pi(j,l), (j,l) \in [m] \times ([n] \setminus \{1\})\} \\ \\ \{\pi\}, \text{ if the first column of } \pi \text{ does not contain any occurrence} \\ \text{ of the letter } a. \end{cases}$$

$$\sigma^{\rightarrow}(\pi) = \begin{cases} \{\pi' \in V_m^n : \exists i \in [m](\pi(i,n) = a \ \&\ \pi'(i,n) = b), \pi'(k,n) = \pi(k,n), \\ k \in [m] \setminus \{i\}, \pi'(j,l) = \pi(j,l), (j,l) \in [m] \times [n-1]\} \\ \\ \{\pi\}, \text{ if the last column of } \pi \text{ does not contain any occurrence} \\ \text{ of the letter } a. \end{cases}$$

$$\sigma^{*}(\pi) = \begin{cases} \{\pi' \in V_m^n : \ \exists (i,j) \in [n] \times [m] \text{ such that } \pi(i,j) = a \text{ and} \\ \pi'(i,j) = b, \pi'(k,l) = \pi(k,l), \forall (k,l) \in ([n] \times [m]) \setminus \{(i,j)\}\} \\ \\ \{\pi\}, \text{ if no column of } \pi \text{ contains any occurrence of the letter } a. \end{cases}$$

Note that a rule as above is applied to all occurrences of the letter a either in the first or in the last or in any column of π, respectively, in different copies of the picture π. Analogously, we define:

• If $\sigma \equiv a \to b(-) \in RSub_V$, then

$$\sigma^{\uparrow}(\pi) = \begin{cases} \{\pi' \in V_m^n : \exists i \in [n](\pi(1,i) = a \ \& \ \pi'(1,i) = b), \pi'(1,k) = \pi(1,k), \\ \forall k \in [n] \setminus \{i\}, \pi'(j,l) = \pi(j,l), \forall (j,l) \in ([m] \setminus \{1\}) \times [n]\} \\ \\ \{\pi\}, \text{ if the first row of } \pi \text{ does not contain any occurrence} \\ \text{ of the letter } a. \end{cases}$$

$$\sigma^{\downarrow}(\pi) = \begin{cases} \{\pi' \in V_m^n : \exists i \in [n](\pi(m,i) = a \ \& \ \pi'(m,i) = b), \pi'(m,k) = \pi(m,k), \\ \forall k \in [n] \setminus \{i\}, \pi'(j,l) = \pi(j,l), \forall (j,l) \in [m-1] \times [n]\} \\ \\ \{\pi\}, \text{ if the last row of } \pi \text{ does not contain any occurrence} \\ \text{ of the letter } a. \end{cases}$$

$\sigma^*(\pi) = \rho^*(\pi)$, where $\rho \equiv a \to b(|) \in CSub_V$.

• If $\sigma \equiv a \to \varepsilon(|) \in CDel_V$, then

$$\sigma^{\leftarrow}(\pi) = \begin{cases} \pi/_{\leftarrow}\rho, \text{ where } \rho \text{ is the leftmost column of } \pi, \text{ if the leftmost} \\ \text{column of } \pi \text{ does contain at least one occurrence of the letter } a \\ \\ \pi, \text{ if the leftmost column of } \pi \text{ does not contain any occurrence} \\ \text{of the letter } a. \end{cases}$$

$$\sigma^{\rightarrow}(\pi) = \begin{cases} \pi/_{\rightarrow}\rho, \text{ where } \rho \text{ is the rightmost column of } \pi, \text{ if the rightmost} \\ \text{column of } \pi \text{ does contain at least one occurrence of the letter } a \\ \\ \pi, \text{ if the rightmost column of } \pi \text{ does not contain any occurrence} \\ \text{of the letter } a. \end{cases}$$

$$\sigma^*(\pi) = \begin{cases} \{\pi_1 \copyright \pi_2 \mid \pi = \pi_1 \copyright \rho \copyright \pi_2, \text{ for some } \pi_1, \pi_2 \in V_*^* \text{ and } \rho \text{ is a} \\ \text{column of } \pi_1 \text{ that contains an occurrence of the letter } a\} \\ \\ \{\pi\}, \text{ if } \pi \text{ does not contain any occurrence of the letter } a. \end{cases}$$

In an analogous way we define:

• If $\sigma \equiv a \to \varepsilon(-) \in RDel_V$, then

$$\sigma^{\uparrow}(\pi) = \begin{cases} \pi/_{\uparrow}\rho, \text{ where } \rho \text{ is the first row of } \pi, \text{ if the first row} \\ \ \text{of } \pi \text{ does contain at least one occurrence of the letter } a \\ \\ \pi, \text{ if the first row of } \pi \text{ does not contain any occurrence} \\ \text{of the letter } a. \end{cases}$$

$$\sigma^{\downarrow}(\pi) = \begin{cases} \pi/_{\downarrow}\rho, \text{ where } \rho \text{ is the last row of } \pi, \text{ if the last row} \\ \ \text{of } \pi \text{ does contain at least one occurrence of the letter } a \\ \\ \pi, \text{ if the last row of } \pi \text{ does not contain any occurrence} \\ \text{of the letter } a. \end{cases}$$

$$\sigma^{*}(\pi) = \begin{cases} \{\pi_1 \circledR \pi_2 \mid \pi = \pi_1 \circledR \rho \circledR \pi_2, \text{ for some } \pi_1, \pi_2 \in V_*^* \text{ and } \rho \text{ is a} \\ \text{row of } \pi_1 \text{ that contains an occurrence of the letter } a\} \\ \\ \{\pi\}, \text{ if } \pi \text{ does not contain any occurrence of the letter } a. \end{cases}$$

For every rule σ, action $\alpha \in \{*, \leftarrow, \rightarrow, \uparrow, \downarrow\}$, and $L \subseteq V_*^*$, we define the *α-action of σ on L* by $\sigma^{\alpha}(L) = \bigcup_{\pi \in L} \sigma^{\alpha}(\pi)$. Given a finite set of rules M, we define the *α-action of M* on the picture π and the language L by:

$$M^{\alpha}(\pi) = \bigcup_{\sigma \in M} \sigma^{\alpha}(\pi) \ \text{ and } \ M^{\alpha}(L) = \bigcup_{\pi \in L} M^{\alpha}(\pi),$$

respectively. In what follows, we shall refer to the rewriting operations defined above as *evolutionary picture operations* since they may be viewed as the 2-dimensional linguistic formulations of local gene mutations.

For two disjoint subsets P and F of an alphabet V and a picture π over V, we define the following two predicates which will define later two types of filters:

$$\begin{array}{lcrcl} rc_s(\pi; P, F) & \equiv & P \subseteq alph(\pi) & \wedge & F \cap alph(\pi) = \emptyset \\ rc_w(\pi; P, F) & \equiv & alph(\pi) \cap P \neq \emptyset & \wedge & F \cap alph(\pi) = \emptyset. \end{array}$$

Note that these conditions are similar to those defined in Section 10.2.2, for words. For every picture language $L \subseteq V_*^*$ and $\beta \in \{s, w\}$, we define:

$$rc_\beta(L, P, F) = \{\pi \in L \mid rc_\beta(\pi; P, F) = \texttt{true}\}.$$

An *evolutionary picture processor over* V is a 5-tuple (M, PI, FI, PO, FO), where:

– Either ($M \subseteq CSub_V$) or ($M \subseteq RSub_V$) or ($M \subseteq CDel_V$) or ($M \subseteq RDel_V$). The set M represents the set of evolutionary rules of the processor. As one can see, a processor is "specialized" into one type of evolutionary operation, only.

– $PI, FI \subseteq V$ are the *input* sets of permitting/forbidding symbols (contexts) of the processor, while $PO, FO \subseteq V$ are the *output* sets of permitting/forbidding symbols of the processor (with $PI \cap FI = \emptyset$ and $PO \cap FO = \emptyset$).

We denote the set of evolutionary picture processors over V by EPP_V.

10.8.3 *Accepting Networks of Evolutionary Picture Processors*

An *accepting network of evolutionary picture processors* (ANEPP for short) is a 8-tuple $\Gamma = (V, U, G, N, \alpha, \beta, x_I, Out)$, where:

- V and U are the input and network alphabet, respectively, $V \subseteq U$.
- $G = (X_G, E_G)$ is an undirected graph without loops with the set of vertices X_G and the set of edges E_G. G is called the *underlying graph* of the network.
- $N : X_G \longrightarrow EPP_V$ is a mapping which associates with each node $x \in X_G$ the evolutionary processor $N(x) = (M_x, PI_x, FI_x, PO_x, FO_x)$.
- $\alpha : X_G \longrightarrow \{*, \leftarrow, \rightarrow, \uparrow, \downarrow\}$; $\alpha(x)$ gives the action mode of the rules of node x on the pictures existing in that node.
- $\beta : X_G \longrightarrow \{s, w\}$ defines the type of the *input/output filters* of a node. More precisely, for every node, $x \in X_G$, the following filters are defined:

$$\begin{aligned} \text{input filter: } & \rho_x(\cdot) = rc_{\beta(x)}(\cdot; PI_x, FI_x), \\ \text{output filter: } & \tau_x(\cdot) = rc_{\beta(x)}(\cdot; PO_x, FO_x). \end{aligned}$$

  That is, $\rho_x(\pi)$ (resp. $\tau_x(\pi)$) indicates whether or not the picture π can pass the input (resp. output) filter of x. More generally, $\rho_x(L)$ (resp. $\tau_x(L)$) is the set of pictures of L that can pass the input (resp. output) filter of x.
- $x_I \in X_G$ is the *input* node and $Out \subset X_G$ is the set of *output* nodes of Γ.

We say that $card(X_G)$ is the size of Γ. A *configuration* of an ANEPP Γ as above is a mapping $C : X_G \longrightarrow 2_f^{U_*^*}$ which associates a finite set of pictures with every node of the graph. A configuration may be understood as the sets of pictures which are present in any node at a given moment. Given a picture $\pi \in V_*^*$, the initial configuration of Γ on π is defined by $C_0^{(\pi)}(x_I) = \{\pi\}$ and $C_0^{(\pi)}(x) = \emptyset$ for all $x \in X_G \setminus \{x_I\}$.

A configuration can change via either an *evolutionary step* or a *communication step.* When changing via an evolutionary step, each component $C(x)$ of the configuration C is changed in accordance with the set of evolutionary rules M_x associated with the node x and the way of applying these rules $\alpha(x)$. Formally, we say that the configuration C' is obtained in *one evolutionary step* from the configuration C, written as $C \Longrightarrow C'$, iff

$$C'(x) = M_x^{\alpha(x)}(C(x)) \text{ for all } x \in X_G.$$

When changing via a communication step, each node processor $x \in X_G$ sends one copy of each picture it has, which is able to pass the output filter of x, to all the node processors connected to x and receives all the pictures sent by any node processor connected with x provided that they can pass its input filter.

Formally, we say that the configuration C' is obtained in *one communication step* from configuration C, written as $C \vdash C'$, iff

$$C'(x) = (C(x) \setminus \tau_x(C(x))) \cup \bigcup_{\{x,y\} \in E_G} (\tau_y(C(y)) \cap \rho_x(C(y))) \text{ for all } x \in X_G.$$

Note that pictures that cannot pass the output filter of a node remain in that node and can be further modified in the subsequent evolutionary steps, while pictures that can pass the output filter of a node are expelled. Further, all the expelled pictures that cannot pass the input filter of any node are lost.

Let Γ be an ANEPP, the computation of Γ on an input picture $\pi \in V_*^*$ is a sequence of configurations $C_0^{(\pi)}, C_1^{(\pi)}, C_2^{(\pi)}, \ldots$, where $C_0^{(\pi)}$ is the initial configuration of Γ on π, $C_{2i}^{(\pi)} \Longrightarrow C_{2i+1}^{(\pi)}$ and $C_{2i+1}^{(\pi)} \vdash C_{2i+2}^{(\pi)}$, $\forall i \geq 0$. Note that configurations are changed by alternative steps. By the previous definitions, each configuration $C_i^{(\pi)}$ is uniquely determined by $C_{i-1}^{(\pi)}$. A computation *halts*, and is said to be *weak (strong) halting*, if one of the following two conditions holds:

(i) There exists a configuration in which the set of pictures existing in at least one output node (all output nodes) is non-empty. In this case, the computation is said to be a weak (strong) *accepting computation.*

(ii) There exist two identical configurations obtained either in consecutive evolutionary steps or in consecutive communication steps.

For the rest of this paper we consider only ANEPPs that halt on every input. The *picture language weakly (strongly) accepted* by Γ is

$$L_{wa(sa)}(\Gamma) = \{\pi \in V_*^* \mid \text{ the computation of } \Gamma \text{ on } \pi \text{ is a weak (strong) accepting one}\}.$$

10.8.4 *Computational Power*

We first establish a useful relationship between the two classes $\mathcal{L}_{wa}(ANEPP)$ and $\mathcal{L}_{sa}(ANEPP)$. As it was expected, we have

Theorem 10.31. $\mathcal{L}_{wa}(ANEPP) \subseteq \mathcal{L}_{sa}(ANEPP)$.

We now compare the classes $\mathcal{L}_{wa}(ANEPP)$ and $\mathcal{L}_{sa}(ANEPP)$ of picture languages weakly and strongly accepted by ANEPPs, respectively, with $\mathcal{L}(LOC)$ and $\mathcal{L}(REC)$ denoting the classes of local and recognizable picture languages, respectively [28].

Theorem 10.32. $\mathcal{L}_{wa}(ANEPP) \setminus \mathcal{L}(REC) \neq \emptyset$.

We do not know whether the inclusion $\mathcal{L}(REC) \subset \mathcal{L}_{wa}(ANEPP)$ holds, however a large part of $\mathcal{L}(REC)$ is included in $\mathcal{L}_{wa}(ANEPP)$ as the next result states. We recall that the complement of any local language is recognizable [28].

Theorem 10.33. *The complement of every local language can be weakly accepted by an ANEPP.*

It is worth mentioning that similar results have been obtained for accepting networks of picture processors with filtered connections considered in [8] but the relationship between the two variants is still unknown.

10.8.5 *Solving Picture Matching with ANEPPs*

A natural problem is to find a pattern (a given picture) in a given picture. This problem is widely known as the two-dimensional pattern matching problem. It is largely motivated by different aspects in low-level image processing [60]. The more general problem of picture matching (the picture is not obligatory a two-dimensional array) is widely known in Pattern Recognition field and is connected with Image Analysis and Artificial Vision [47, 64].

In [9] an ANEPP that weakly accepts the singleton language formed by a given picture of size $(2, 2)$ is constructed. Based on this ANEPP one proves that the two-dimensional pattern matching problem can be solved by ANEPPs with weak acceptance (or the problem is weakly decided by ANEPP) provided that the pattern is of size $(2,2)$. Can this result be extended to patterns of arbitrary size? In [9] one shows that it can be extended to patterns of size (i,n) and (n,i) for any $1 \leq i \leq 3$ and $n \geq 1$. However, the general problem of pattern matching in a given picture remains open.

Theorem 10.34. *Given a finite set F of patterns of size (i,n) and (n,i) for all $1 \leq i \leq 3$ and $n \geq 1$, the pattern matching problem with patterns from F can be weakly decided by ANEPPs.*

Various algorithms exist for the exact two-dimensional matching problem. The fastest algorithms for finding a rectangular picture pattern in a given picture of size (n,m) run in $\mathcal{O}(n \times m)$ time, see, e.g., [6, 66]. It is rather easy to note that an ANEPP which weakly decides whether a pattern of size (i,p), $1 \leq i \leq 3$, appears in a given picture of size (n,m) does this in $\mathcal{O}(n+m)$ computational (evolutionary and communication) steps. On the other hand, the space complexity of the algorithm proposed in [66] is $\mathcal{O}(n \times m)$, while the number of pictures moving through the network is pretty large. We recall that some biological phenomena are sources of inspiration for our model. In this context, it is considered to be biologically feasible to have sufficiently many identical copies of a molecule.

As we have seen, the general problem of weakly deciding whether a given pattern appears in a picture remained open. A brief discussion seems in order. The main idea in the proof of Theorem 10.34 can be informally described as follows. For each picture in F, we have a subnetwork accepting exactly that picture. Given an input picture, another part of the total network extracts arbitrarily a subpicture which is given to all subnetworks accepting pictures from F. If at least one of them recognizes it, the input picture is accepted. We were not able to extend this strategy to patterns of arbitrary size as we could not design a network able to weakly accept exactly a picture of a size different than those mentioned in the statement of Theorem 10.34. On the other hand, for every picture we can define a network strongly accepting exactly that picture. Unfortunately, this construction cannot be harmonized with the aforementioned strategy. Consequently, one may naturally ask: Could the problem be strongly decided?

References

1. Alhazov, A., Csuhaj-Varjú, E., Martín-Vide, C. and Rogozhin, Y. (2008). About Universal Hybrid Networks of Evolutionary Processors of Small Size, in *Proc. LATA 2008, LNCS 5196*, pp. 28–39.
2. Alhazov, A., Csuhaj-Varjú, E., Martín-Vide, C. and Rogozhin, Y. (2009a). On the size of computationally complete hybrid networks of evolutionary processors, *Theor. Comput. Sci.* **410**, 35, pp. 3188–3197.
3. Alhazov, A., Enguix, G. B. and Rogozhin, Y. (2009b). Obligatory Hybrid Networks of Evolutionary Processors, in *Proc. ICAART 2009*, pp. 613–618.
4. Alhazov, A., Martín-Vide, C., Truthe, B., Dassow, J. and Rogozhin, Y. (2009c). On Networks of Evolutionary Processors with Nodes of Two Types, *Fundam. Inf.* **91**, 1, pp. 1–15.
5. Alhazov, A. and Rogozhin, Y. (2008). About Precise Characterization of Languages Generated by Hybrid Networks of Evolutionary Processors with One Node, *The Computer Science Journal of Moldova* **16**, 3, pp. 364–376.
6. Amir, A., Benson, G. and Farach, M. (1992). Alphabet independent two dimensional matching, in *STOC '92: Proceedings of the twenty-fourth annual ACM symposium on Theory of computing* (ACM, New York, NY, USA), pp. 59–68.
7. Bottoni, P., Labella, A., Manea, F., Mitrana, V. and Sempere, J. M. (2009a). Filter Position in Networks of Evolutionary Processors Does Not Matter: A Direct Proof, in *DNA Computing and Molecular Programming: 15th International Conference, DNA 15, Revised Selected Papers, LNCS 5877* (Springer-Verlag, Berlin, Heidelberg), pp. 1–11.
8. Bottoni, P., Labella, A., Manea, F., Mitrana, V. and Sempere, J. M. (2009b). Networks of Evolutionary Picture Processors with Filtered Connections, in *UC '09: Proceedings of the 8th International Conference on Unconventional Computation, LNCS 5715* (Springer-Verlag, Berlin, Heidelberg), pp. 70–84.
9. Bottoni, P., Labella, A., Mitrana, V. and Sempere, J. M. (2010). Networks of Evolutionary Picture Processors, *submitted* .
10. Bozapalidis, S. and Grammatikopoulou, A. (2005). Recognizable Picture Series, *Journal of Automata, Languages and Combinatorics* **10**, 2/3, pp. 159–183.
11. Castellanos, J., Leupold, P. and Mitrana, V. (2005). On the Size Complexity of Hybrid Networks of Evolutionary Processors, *Theor. Comput. Sci.* **330**, 2, pp. 205–220.
12. Castellanos, J., Manea, F., de Mingo López, L. F. and Mitrana, V. (2007). Accepting Networks of Splicing Processors with Filtered Connections, in *Proc. MCU 2007, LNCS 4664*, pp. 218–229.
13. Castellanos, J., Martin-Vide, C., Mitrana, V. and Sempere, J. M. (2001). Solving NP-Complete Problems With Networks of Evolutionary Processors, in *IWANN '01: Proceedings of the 6th International Work-Conference on Artificial and Natural Neural Networks, LNCS 2084* (Springer-Verlag, London, UK), pp. 621–628.

14. Castellanos, J., Martín-Vide, C., Mitrana, V. and Sempere, J. M. (2003). Networks of Evolutionary Processors, *Acta Inf.* **39**, 6-7, pp. 517–529.
15. Cocke, J. and Minsky, M. (1964). Universality of Tag Systems with P = 2, *J. ACM* **11**, 1, pp. 15–20.
16. Csuhaj-Varjú, E., Kari, L. and Paun, G. (1996). Test Tube Distributed Systems Based on Splicing, *Computers and Artificial Intelligence* **15**, 2-3.
17. Csuhaj-Varjú, E., Martín-Vide, C. and Mitrana, V. (2005). Hybrid Networks of Evolutionary Processors are Computationally Complete, *Acta Inf.* **41**, 4-5, pp. 257–272.
18. Csuhaj-Varjú, E. and Mitrana, V. (2000). Evolutionary Systems: A Language Generating Device Inspired by Evolving Communities of Cells, *Acta Inf.* **36**, 11, pp. 913–926.
19. Csuhaj-Varjú, E. and Salomaa, A. (1997). Networks of Parallel Language Processors, in *New Trends in Formal Languages - Control, Cooperation, and Combinatorics (to Jürgen Dassow on the occasion of his 50th birthday), LNCS 1218* (Springer-Verlag, London, UK), pp. 299–318.
20. Dassow, J. and Mitrana, V. (2009). Accepting Networks of Non-inserting Evolutionary Processors, *T. Comp. Sys. Biology* **11**, pp. 187–199.
21. Dassow, J., Mitrana, V. and Truthe, B. (2010). The Role of Evolutionary Operations in Accepting Networks of Evolutionary Processors, *submitted* .
22. Dassow, J. and Truthe, B. (2007). On the Power of Networks of Evolutionary Processors, in *Proc. MCU 2007, LNCS 4664*, pp. 158–169.
23. De Errico, L. and Jesshope, C. (1994). Towards a New Architecture for Symbolic Processing, in *AIICSR'94: Proceedings of the sixth international conference on Artificial intelligence and information-control systems of robots* (World Scientific Publishing Co., Inc., River Edge, NJ, USA), pp. 31–40.
24. Dragoi, C. and Manea, F. (2008). On the Descriptional Complexity of Accepting Networks of Evolutionary Processors with Filtered Connections, *Int. J. Found. Comput. Sci.* **19**, 5, pp. 1113–1132.
25. Dragoi, C., Manea, F. and Mitrana, V. (2007). Accepting Networks of Evolutionary Processors with Filtered Connections, *J. UCS* **13**, 11, pp. 1598–1614.
26. Fahlman, S. E., Hinton, G. E. and Sejnowski, T. J. (1983). Massively Parallel Architectures for AI: NETL, Thistle, and Boltzmann Machines, in *Proc. AAAI 1983*, pp. 109–113.
27. Garey, M. R. and Johnson, D. S. (1990). *Computers and Intractability; A Guide to the Theory of NP-Completeness* (W. H. Freeman & Co., New York, NY, USA).
28. Giammarresi, D. and Restivo, A. (1992). Recognizable Picture Languages, *IJPRAI* **6**, 2–3, pp. 241–256.
29. Giammarresi, D. and Restivo, A. (1997). Two-dimensional Languages, in *Handbook of formal languages, vol. 3: beyond words* (Springer-Verlag New York, Inc., New York, NY, USA), pp. 215–267.
30. Giammarresi, D., Restivo, A., Seibert, S. and Thomas, W. (1996). Monadic Second-order Logic over Rectangular Pictures and Recognizability by Tiling Systems, *Inf. Comput.* **125**, 1, pp. 32–45.
31. Hartmanis, J. and Stearns, R. E. (1965). On the Computational Complexity

of Algorithms, *Trans. Amer. Math. Soc.* **117**, pp. 533–546.
32. Hillis, W. D. (1986). *The Connection Machine* (MIT Press, Cambridge, MA, USA).
33. Inoue, K. and Takanami, I. (1991). A Survey of Two-dimensional Automata Theory, *Inf. Sci.* **55**, 1-3, pp. 99–121.
34. Loos, R., Manea, F. and Mitrana, V. (2009a). On Small, Reduced, and Fast Universal Accepting Networks of Splicing Processors, *Theor. Comput. Sci.* **410**, 4-5, pp. 406–416.
35. Loos, R., Manea, F. and Mitrana, V. (2009b). Small Universal Accepting Networks of Evolutionary Processors with Filtered Connections, *Proc. DCFS 2009 – EPTCS* **3**, pp. 173–182.
36. Loos, R., Manea, F. and Mitrana, V. (2010). Small Universal Accepting Hybrid Networks of Evolutionary Processors, *Acta Inf.* **47**, 2, pp. 133–146.
37. Manea, F. (2005). Timed Accepting Hybrid Networks of Evolutionary Processors, in *Proc. IWINAC 2005 (2), LNCS 3562*, pp. 122–132.
38. Manea, F., Margenstern, M., Mitrana, V. and Perez-Jimenez, M. J. (2010). A New Characterization of NP, P and PSPACE with Accepting Hybrid Networks of Evolutionary Processors, *Theor. Comp. Sys.* **46**, 2, pp. 174–192.
39. Manea, F., Martín-Vide, C. and Mitrana, V. (2004). Solving 3CNF-SAT and HPP in Linear Time Using WWW, in *Proc. MCU 2004, LNCS 3354*, pp. 269–280.
40. Manea, F., Martín-Vide, C. and Mitrana, V. (2005). Accepting Networks of Splicing Processors, in *Proc. CiE 2005, LNCS 3526*, pp. 300–309.
41. Manea, F., Martín-Vide, C. and Mitrana, V. (2006a). A Universal Accepting Hybrid Network of Evolutionary Processors, *Electr. Notes Theor. Comput. Sci.* **135**, 3, pp. 95–105.
42. Manea, F., Martín-Vide, C. and Mitrana, V. (2006b). All NP-Problems Can Be Solved in Polynomial Time by Accepting Networks of Splicing Processors of Constant Size, in *Proc. DNA 2006, LNCS 4287*, pp. 47–57.
43. Manea, F., Martín-Vide, C. and Mitrana, V. (2007a). Accepting Networks of Splicing Processors: Complexity Results, *Theor. Comput. Sci.* **371**, 1-2, pp. 72–82.
44. Manea, F., Martín-Vide, C. and Mitrana, V. (2007b). On the Size Complexity of Universal Accepting Hybrid Networks of Evolutionary Processors, *Mathematical Structures in Computer Science* **17**, 4, pp. 753–771.
45. Manea, F. and Mitrana, V. (2007). All NP-problems Can Be Solved in Polynomial Time by Accepting Hybrid Networks of Evolutionary Processors of Constant Size, *Inf. Process. Lett.* **103**, 3, pp. 112–118.
46. Margenstern, M. and Rogozhin, Y. (2001). Time-Varying Distributed H Systems of Degree 1 Generate All Recursively Enumerable Languages, in *Words, Semigroups, and Transductions* (World Scientific), pp. 329–339.
47. Marriott, K. and Meyer, B. (eds.) (1998). *Visual language theory* (Springer-Verlag New York, Inc., New York, NY, USA).
48. Martín-Vide, C. and Mitrana, V. (2005). Networks of evolutionary processors: Results and perspectives, in *Molecular Computational Models: Unconventional Approaches*, pp. 78–114.

49. Mäurer, I. (2006). Weighted Picture Automata and Weighted Logics, in *Proc. STACS 2006, LNCS 3884*, pp. 313–324.
50. Mäurer, I. (2007). Characterizations of Recognizable Picture Series, *Theor. Comput. Sci.* **374**, 1-3, pp. 214–228.
51. Papadimitriou, C. M. (1994). *Computational Complexity* (Addison-Wesley, Reading, Massachusetts).
52. Paun, A. (1999). On Time-Varying H Systems, *Bulletin of the EATCS* **67**, pp. 157–164.
53. Paun, G. (1996). Regular Extended H Systems are Computationally Universal, *Journal of Automata, Languages and Combinatorics* **1**, 1, pp. 27–36.
54. Paun, G. (1997). DNA Computing: Distributed Splicing Systems, in *Structures in Logic and Computer Science, LNCS 1261*, pp. 353–370.
55. Paun, G. (1998). Distributed Architectures in DNA Computing Based on Splicing: Limiting the Size of Components, in *Proc. Unconventional Models of Computation, DMTCS*, pp. 323–335.
56. Paun, G. (2000). Computing with Membranes, *J. Comput. Syst. Sci.* **61**, 1, pp. 108–143.
57. Paun, G. and Sântean, L. (1989). Parallel Communicating Grammar Systems: The Regular Case, *Annals of University of Bucharest, Ser. Matematica-Informatica* **38**, pp. 55–63.
58. Post, E. L. (1943). Formal Reductions of the General Combinatorial Decision Problem, *Amer. J. Math.* **65**, 2, pp. 197–215.
59. Rogozhin, Y. (1996). Small Universal Turing Machines, *Theor. Comput. Sci.* **168**, 2, pp. 215–240.
60. Rosenfeld, A. and Kak, A. C. (1982). *Digital Picture Processing* (Academic Press, Inc., Orlando, FL, USA).
61. Rosenfeld, A. and Siromoney, R. (1993). Picture languages—a survey, *Lang. Des.* **1**, 3, pp. 229–245.
62. Rozenberg, G. and Salomaa, A. (1997). *Handbook of Formal Languages* (Springer-Verlag New York, Inc., Secaucus, NJ, USA).
63. Sankoff, D., Leduc, G., Antoine, N., Paquin, B., Lang, F. and Cedergren, R. (1992). Gene Order Comparisons for Phylogenetic Inference: Evolution of the Mitochondrial Genome, *Proceedings of the National Academy of Sciences of the United States of America* **89**, 14, pp. 6575–6579.
64. Wang, P. and Bunke, H. (1997). *Handbook on Optical Character Recognition and Document Image Analysis* (World Scientific).
65. Woods, D. and Neary, T. (2006). On the Time Complexity of 2-tag Systems and Small Universal Turing Machines, in *FOCS '06: Proceedings of the 47th Annual IEEE Symposium on Foundations of Computer Science* (IEEE Computer Society, Washington, DC, USA), pp. 439–448.
66. Zhu, R. F. and Takaoka, T. (1989). A Technique for Two-dimensional Pattern Matching, *Commun. ACM* **32**, 9, pp. 1110–1120.

This page is intentionally left blank

Chapter 11

Quantum Automata and Periodic Events

Carlo Mereghetti and Beatrice Palano
Dipartimento di Scienze dell'Informazione,
Università degli Studi di Milano,
via Comelico 39, 20135 Milano, Italy,
E-mail: {mereghetti,palano}@dsi.unimi.it

We study the phenomenon of *periodicity* on quantum automata from three points of view. First, we provide an efficient algorithm for *testing* the periodicity of stochastic events induced by measure-once one-way unary quantum automata (1qfa's). Second, we design an algorithm for the *synthesis* of succinct unary 1qfa's inducing periodic events. To evaluate the size of the resulting 1qfa's, we relate the number of states of a minimal 1qfa inducing a linear approximation of a periodic event p to the harmonic structure of p. Next, we study the complexity of our synthesis algorithm. Third, we *apply* our synthesis algorithm for building succinct 1qfa's that accept unary periodic languages. We also prove a lower bound on the size of 1qfa's accepting unary periodic languages.

11.1 Introduction

Quantum computing is a research area halfway between computer science and physics [21]. In the early 80's, Feynman suggested that the computational power of quantum mechanical processes might be beyond that of traditional computation models [17]. Almost at the same time, Benioff already determined that such processes are at least as powerful as Turing machines [4]. Discussing the notion of "quantum computational device", Deutsch introduced the model of quantum Turing machine as a physically realizable model for a quantum computer [16]. From the point of view of structural complexity, Bernstein and Vazirani defined the class BQP of problems solvable in polynomial time on quantum Turing machines with an arbitrary small error probability, and they settled the comparison with the corresponding probabilistic class BPP [5].

The most famous result witnessing quantum power is Shor's algorithm for integer factorization, running in polynomial time on a quantum computer [36]. This is to be compared with the currently known best classical factoring algorithms, featuring exponential running time (see, e.g., [24]). Another relevant progress is made by Grover, who gives a quantum algorithm for searching an item in an unsorted database of n items in time $O(\sqrt{n})$, against classical database searching where $O(n)$ time is required [20].

The power of quantum paradigm crucially relies on the typical features of quantum systems: superposition, interference, and observation. The state of a quantum machine can be seen as a linear combination of classical states (superposition). A unitary transformation makes the machine evolve from superposition to superposition. Superposition can transfer the complexity of the problem from a large number of sequential steps to a large number of coherently superposed quantum states. Entanglement is used to create complicated correlations allowing interference between the "parallel computations" performed by the quantum machine. Furthermore, the machine can be observed: this process makes the current superposition collapse to a particular state.

Efforts have been made to construct quantum devices, and their realization seems to be a very difficult task (see, e.g., [21, 32, 34]). We can hardly expect to see a fully quantum computer in the near future, while it is reasonable to think of classical computers incorporating small quantum components [3] such as, e.g., *quantum finite automata* (qfa's, for short). Qfa's are computational devices particularly interesting since they represent a theoretical model for a quantum computer with finite memory. Several variants of qfa's have been introduced, some of which with physical motivations: measure-once [6, 14, 31], measure-many [2, 23], measure-only [11], enhanced [33], reversible [19], with control language [9]. Basically, these models differ in the measurement policy. Here, we will focus on the simplest model which is represented by *measure-once* one-way qfa's (1qfa's, for short). Very roughly speaking, a measure-once 1qfa can be regarded as a classical finite state automaton, on which the quantum paradigm is imposed. Yet, the probability of accepting strings is evaluated by "observing just once", at the end of input processing. Investigations on measure-once 1qfa's typically follows two directions, aiming to settle comparisons with classical automata models. On the one hand, their computational power (the class of accepted languages) has been investigated and established. Quite surprisingly, the class of languages accepted by measure-once 1qfa's with isolated cut point is a proper subclass (group languages [35]) of regular languages [6, 14]. On the other hand, several results witness that sometimes measure-once 1qfa's turn out to have a *higher descriptional power*, resulting in very small size devices.

One of the main attractive tasks where such a descriptional superiority shows up is the ability of realizing *periodic behaviors.* The simplest and meaningful form of periodicity can be appreciated by focusing on *unary* devices, i.e., working on single-letter input alphabets. To be more precise, let A be a measure-once 1qfa with unary input alphabet $\{\sigma\}$. The stochastic event induced by A is the function $p_A : \mathbb{N} \to [0,1]$ such that $p_A(n)$ is the probability that A accepts the string σ^n. Such an event is said to be *n-periodic* whenever p_A is a function of period n. Some generalizations enlarging the notion of periodicity to general alphabets have been proposed and studied in the literature [10, 29]. Here, we will be dealing with unary events only. The possibility of efficiently reproducing periodic events has a lot of relevant applications especially in language theory, where it often leads to the construction of small accepting automata.

In this paper, we are going to provide a survey on results coming from three natural lines of investigation on periodicity by measure-once 1qfa's: *testing*, *synthesis*, *applications.*

First, we focus on the problem of *testing* whether a given measure-once 1qfa induces an event of a given periodicity. We prove that this problem is decidable by using generating functions arguments. Moreover, we have that our deciding algorithm is polynomial in time whenever running on automata with complex amplitudes of rational components. The problem of testing periodicity on a more general model of quantum automata is studied in [13].

Second, we study the problem, tackled in [7], of *synthesizing* small measure-once 1qfa's inducing given periodic events. Given an n-periodic event p, we provide a polynomial time algorithm which builds a measure-once 1qfa A with $O(\sqrt{n})$ states inducing the event $ap + b$ for reals $a > 0$, $b \geq 0$ satisfying $a + b \leq 1$. Actually, we show that the size of a measure-once 1qfa inducing $ap + b$ can be related to the harmonic structure of p by the notion of *difference cover* [15]. A difference cover for a set $X \subseteq \mathbb{Z}_n$ is any set

$\Delta \subseteq \mathbb{Z}_n$ such that any element of X can be obtained as a difference modulo n of two elements in Δ. We prove that the number s of states of a minimal measure-once 1qfa inducing $ap+b$ satisfies the bound $q \leq s \leq 4q+1$, where q is the cardinality of a minimum difference cover for the support set $\{j \in \mathbb{Z}_n \mid$ the jth coefficient of the discrete Fourier transform of p is not null$\}$. So, in order to reduce automata size, it turns out to be relevant the problem of computing minimum difference covers. We show that this problem is NP-hard [28].

Third, we give some *applications* of these results to unary periodic language acceptance by measure-once 1qfa's. A unary language is n-periodic whenever its characteristic function is n-periodic. Periodic languages are a very well studied benchmark for testing the descriptional power of different kinds of automata [30]. We prove that any n-periodic language can be accepted with isolated cut point by a measure-once 1qfa with $O(\sqrt{n})$ states [27]. This result emphasizes a quadratic gap in the descriptional power between the quantum and classical (also, probabilistic) paradigm. Yet, we prove a lower limit to quantum automata succinctness. In fact, we show the existence of n-periodic languages that cannot be accepted by measure-once 1qfa's inducing periodic events with less than $\sqrt{n/(3\log n)}$ states [8].

11.2 Preliminaries

We assume that the reader is familiar with basic notions on formal language and complexity theory (see, e.g., [18, 22]). Below, we present preliminaries on linear algebra, and the model of quantum automaton we shall be dealing with.

11.2.1 *Linear Algebra*

We quickly recall some notions of linear algebra. For more details, we refer the reader to, e.g., [26, 25]. The field of complex numbers is denoted by $\mathbb{C}$. Given a complex number $z \in \mathbb{C}$, its *conjugate* is denoted by z^*, and its *modulus* is $|z| = \sqrt{zz^*}$. We let $\mathbb{C}^{n\times m}$ denote the set of $n \times m$ matrices with entries in $\mathbb{C}$. Given a matrix $M \in \mathbb{C}^{n\times m}$, for $1 \leq i \leq n$ and $1 \leq j \leq m$, we let M_{ij} denote its (i,j)th entry. The *transpose* of M is the matrix $M^T \in \mathbb{C}^{m\times n}$ satisfying ${M^T}_{ij} = M_{ji}$, while we let M^* the matrix ${M^*}_{ij} = (M_{ij})^*$. The *adjoint* of M is the matrix $M^\dagger = (M^T)^*$.

For matrices $A, B \in \mathbb{C}^{n\times m}$, their *sum* is the $n \times m$ matrix $(A+B)_{ij} = A_{ij}+B_{ij}$. For matrices $C \in \mathbb{C}^{n\times m}$ and $D \in \mathbb{C}^{m\times r}$, their *product* is the $n\times r$

matrix $(CD)_{ij} = \sum_{k=1}^{m} C_{ik}D_{kj}$. For matrices $A \in \mathbb{C}^{n\times m}$ and $B \in \mathbb{C}^{p\times q}$, their *direct sum* and *Kronecker (or direct) product* are the $(m+p)\times(n+q)$ and $mp \times nq$ matrices defined, respectively, as

$$A \oplus B = \begin{pmatrix} A & \mathbf{0} \\ \mathbf{0} & B \end{pmatrix}, \quad A \otimes B = \begin{pmatrix} A_{11}B & \cdots & A_{1m}B \\ \vdots & \ddots & \vdots \\ A_{n1}B & \cdots & A_{nm}B \end{pmatrix},$$

where $\mathbf{0}$ denotes zero-matrices of suitable dimensions. When operations are allowed by matrices dimensions, we have that $(A\otimes B)\cdot(C\otimes D) = AC\otimes BD$ and $(A \oplus B) \cdot (C \oplus D) = AC \oplus BD$.

An Hilbert space of dimension n is the linear space $\mathbb{C}^{1\times n}$ equipped with sum and product by elements in $\mathbb{C}$, in which, for vectors $\pi, \xi \in \mathbb{C}^{1\times n}$, the *inner product* $\langle \pi, \xi \rangle = \pi\xi^\dagger$ is defined. The i-th component of π is denoted by π_i. The (l^2) *norm* of π is given by $\| \pi \| = \sqrt{\langle \pi, \pi \rangle}$. If $\langle \pi, \xi \rangle = 0$ (and $\| \pi \| = 1 = \| \xi \|$), we say that π is *orthogonal* (*orthonormal*) to ξ. Two subspaces $X, Y \subseteq \mathbb{C}^{1\times n}$ are orthogonal if any vector in X is orthogonal to any vector in Y. In this case, the linear space generated by $X \cup Y$ is denoted by $X \oplus Y$.

A matrix $M \in \mathbb{C}^{n\times n}$ can be view as the morphism $\pi \mapsto \pi M$ of the Hilbert space $\mathbb{C}^{1\times n}$ into itself. If $MM^\dagger = M^\dagger M$ then M is said to be *normal*. Two important subclasses of normal matrices are the unitary and the Hermitian matrices. M is said to be *unitary* whenever $MM^\dagger = I = M^\dagger M$. The *eigenvalues* of unitary matrices are complex numbers of modulus 1, i.e., they are in the form $e^{i\theta}$, for some real θ. This fact characterizes the class of unitary matrices if we restrict to normal matrices. Alternative characterizations of normal and unitary matrices are contained, respectively, in

Proposition 11.1. [25] (Thm. 4.10.3) *A matrix $M \in \mathbb{C}^{n\times n}$ is normal if and only if there exists a unitary matrix $X \in \mathbb{C}^{n\times n}$ such that $M = XDX^\dagger$, where $D = \mathrm{diag}(\nu_1, \ldots, \nu_n)$ is the diagonal matrix of the eigenvalues of M.*

Proposition 11.2. [25] (Thms. 4.7.24, 4.7.14) *A matrix $M \in \mathbb{C}^{n\times n}$ is unitary if and only if:*

(i) its rows are mutually orthonormal vectors;
(ii) $\| \pi M \| = \| \pi \|$, for each vector $\pi \in \mathbb{C}^{1\times n}$.

The matrix M is said to be *Hermitian* whenever $M = M^\dagger$. Given an Hermitian matrix $\mathcal{O} \in \mathbb{C}^{n\times n}$, let $c_1, \ldots, c_s$ be its eigenvalues and $E_1, \ldots, E_s$

the corresponding eigenspaces. It is well known that each eigenvalue c_k is real, that E_i is orthogonal to E_j, for $1 \leq i \neq j \leq s$, and that $E_1 \oplus \cdots \oplus E_s = \mathbb{C}^{1\times n}$. In this case, we say that $\{E_1, \ldots, E_s\}$ is a decomposition of $\mathbb{C}^{1\times n}$. In fact, every vector $\pi \in \mathbb{C}^{1\times n}$ can be uniquely decomposed as $\pi = \pi_1 + \cdots + \pi_s$, for unique $\pi_j \in E_j$. An Hermitian matrix is *positive semidefinite* if and only if all its eigenvalues are non negative. Alternative characterizations are contained in

Proposition 11.3. [26] (Thms. 4.12, 4.8) *An Hermitian matrix $M \in \mathbb{C}^{n\times n}$ is positive semidefinite if and only if:*

(i) $\pi M \pi^\dagger \geq 0$, *for each vector* $\pi \in \mathbb{C}^{1\times n}$;
(ii) $M = YY^\dagger$, *for some matrix* $Y \in \mathbb{C}^{n\times n}$.

11.2.2 *Quantum Finite Automata*

In this paper, we are interested only in *measure-once* quantum finite automata [2, 14, 31]. Hereafter, the attribute measure-once will always be understood. The "hardware" of a *one-way quantum finite automaton* is that of a classical finite automaton. Thus, we have an input tape which is scanned by an input head moving one position right at each move[1], plus a finite state control. Formally:

Definition 11.4. A one-way quantum finite automaton (1qfa, for short) is a quintuple $A = (Q, \Sigma, \pi(0), \delta, F)$, where

- $Q = \{s_1, s_2, \ldots, s_q\}$ is the finite set of states,
- Σ is the finite input alphabet,
- $\pi(0) \in \mathbb{C}^{1\times q}$, with $\| \pi(0) \|^2 = 1$, is the vector of the initial amplitudes of the states,
- $F \subseteq Q$ is the set of accepting states,
- $\delta : Q \times \Sigma \times Q \to \mathbb{C}$ is the transition function mapping into the set of complex numbers having square modulus not exceeding 1; $\delta(s_i, \sigma, s_j)$ is the amplitude of reaching the state s_j from the state s_i, upon reading σ. The transition function must satisfy the following condition of well-formedness: for any $\sigma \in \Sigma$ and $1 \leq i, j \leq q$,

$$\sum_{k=1}^{q} \delta(s_i, \sigma, s_k)\, \delta^*(s_j, \sigma, s_k) = \begin{cases} 1 \text{ if } i = j \\ 0 \text{ otherwise.} \end{cases}$$

[1]This kind of automata are sometimes referred to as *real time* automata [31], stressing the fact that they can never present stationary moves.

It is useful to express the transition function on $\sigma \in \Sigma$ as the *transition matrix* $U(\sigma) \in \mathbb{C}^{q \times q}$ with $U(\sigma)_{ij} = \delta(s_i, \sigma, s_j)$. Since δ satisfies the condition of well-formedness above displayed, the rows of each $U(\sigma)$ are easily seen to be mutually orthonormal vectors and hence, by Proposition 11.2(i), *$U(\sigma)$'s are unitary matrices.* The 1qfa A can thus be represented as a triple $A = (\pi(0), \{U(\sigma)\}_{\sigma \in \Sigma}, \eta_F)$, where $\eta_F \in \{0,1\}^{1 \times n}$ is the characteristic vector of the accepting states.

Let us briefly discuss how A works. At any given time t, the *state* of A is a *superposition* of the states in Q, and is represented by a vector $\pi(t)$ *of norm 1* in the Hilbert space $l^2(Q)$ (the space of mappings from Q to $\mathbb{C}$ with l^2 norm). In particular, $\pi(t)_i$ is the amplitude of the state s_i. The computation on input $x = x_1 \cdots x_n \in \Sigma^*$ starts in the superposition $\pi(0)$. After k steps, i.e., after reading the first k input symbols, the state of A is the superposition

$$\pi(k) = \pi(0)U(x_1) \cdots U(x_k).$$

Since $\| \pi(0) \| = 1$ and $U(x_i)$'s are unitary matrices, Proposition 11.2(ii) ensures that $\| \pi(k) \| = 1$. After entering the final superposition $\pi(n) = \pi(0) \prod_{i=1}^{n} U(x_i)$, we *observe* A by the *standard observable* $\mathcal{O} = \{l^2(F), l^2(Q \setminus F)\}$, which is basically the decomposition of $l^2(Q)$ into the two orthogonal subspaces spanned by the accepting and nonaccepting states, respectively. The *probability that A accepts x* is given by the square norm of the projection of $\pi(n)$ onto $l^2(F)$. Formally:

$$p_{acc}(x) = \sum_{\{j \mid (\eta_F)_j = 1\}} |(\pi(0) \prod_{i=1}^{n} U(x_i))_j|^2.$$

A *stochastic event* is a function $p : \Sigma^* \to [0,1]$. The stochastic event *induced by the 1qfa A* is the function $p_A : \Sigma^* \to [0,1]$ defined, for any $x \in \Sigma^*$, as $p_A(x) = p_{acc}(x)$. The *language accepted by A with cut point λ* is the set

$$L_{A,\lambda} = \{x \in \Sigma^* \mid p_A(x) > \lambda\}.$$

A language L is said to be accepted by A *with isolated cut point* λ, if there exists $\varepsilon > 0$ such that, for any $x \in L$ ($x \notin L$), we have $p_A(x) \geq \lambda + \varepsilon$ ($\leq \lambda - \varepsilon$).

A 1qfa A is *unary* whenever $|\Sigma| = 1$. In this case, we let $\Sigma = \{\sigma\}$, and we can write $A = (\pi(0), U, \eta_F)$ since we have a unique transition matrix U. With a slight abuse of notation, we will be writing k for the input string σ^k. The probability of accepting k now writes as

$$p_{acc}(k) = \sum_{\{j \mid (\eta_F)_j = 1\}} |(\pi(0)U^k)_j|^2. \qquad (11.1)$$

The stochastic event *induced by the unary 1qfa* A is the function $p_A : \mathbb{N} \to [0,1]$, with $p_A(k) = p_{acc}(k)$. A stochastic event $p : \mathbb{N} \to [0,1]$ is said to be *n-periodic* if it is an n-periodic function. In this case, p can be clearly represented by the vector $(p(0), \ldots, p(n-1))$. A *unary language* is a set $L \subseteq \sigma^*$. L is *n-periodic* if there exists a set $S \subseteq \mathbb{Z}_n$ such that $L = \{k \in \mathbb{N} \mid (k \bmod n) \in S\}$.

11.3 Testing Periodicity on Unary 1qfa's

Let us start by tackling the investigation on the periodicity phenomenon on 1qfa's from the point of view of *testing*. More formally, we consider the following decision problem:

d-PERIODICITY

INPUT: A unary 1qfa A and an integer $d > 0$.
OUTPUT: Is p_A a d-periodic event?

We give an algorithm to decide this problem. Yet, if the 1qfa's in input have rational entries, i.e., *complex numbers with rational components*, the time complexity of our algorithm is polynomial. Notice that 1qfa's with rational entries do not necessarily induce periodic events. For instance, consider the unary 1qfa

$$A = \left((1,0), \begin{pmatrix} \cos\pi\theta & \sin\pi\theta \\ -\sin\pi\theta & \cos\pi\theta \end{pmatrix} = \begin{pmatrix} \frac{3}{5} & \frac{4}{5} \\ -\frac{4}{5} & \frac{3}{5} \end{pmatrix}, (1,0) \right),$$

inducing the event $p_A(n) = (\cos\pi n\theta)^2$, which is periodic if and only if θ is rational. It is known that, for rational θ, the only rational values of $\cos(\pi\theta)$ are $0, \pm\frac{1}{2}, \pm 1$. So, the event induced by A is not periodic. Therefore, restricting d-PERIODICITY to 1qfa's with rational entries is well worth investigating.

To approach d-PERIODICITY, we find it useful to provide a particular representation of 1qfa's. Given a unary 1qfa $A = (\pi, U, \eta)$ with q states, we define its *linear representation* as follows. First, for $1 \leq i \leq q$, we define the q-dimensional boolean vector $\bar{\eta}_i = \eta_i e_i$, with $e_i \in \{0,1\}^{1\times q}$ having 1 at the i-th component, and 0 elsewhere. Thus, $\eta = \sum_{i=1}^{q} \bar{\eta}_i$. The linear representation of A is the tuple $(\tilde{\pi}, \tilde{U}, \tilde{\eta})$, where

- $\tilde{\pi} = \pi \otimes \pi^*$,
- $\tilde{U} = U \otimes U^*$,

- $\tilde{\eta} = \sum_{i=1}^{q} \bar{\eta}_i \otimes \bar{\eta}_i$.

We have that $p_A(n) = \tilde{\pi}\tilde{U}^n\tilde{\eta}$, for every $n \in \mathbb{N}$. In fact:

$$\begin{aligned}\tilde{\pi}\tilde{U}^n\tilde{\eta} &= (\pi \otimes \pi^*)(U \otimes U^*)^n \sum_{i=1}^{q} \bar{\eta}_i \otimes \bar{\eta}_i = \sum_{i=1}^{q} (\pi U^n \bar{\eta}_i) \otimes (\pi^* {U^*}^n \bar{\eta}_i) \\ &= \sum_{\{j \mid \eta_j = 1\}} (\pi U^n)_j (\pi U^n)_j^* = \sum_{\{j \mid \eta_j = 1\}} |(\pi U^n)_j|^2 = p_A(n).\end{aligned}$$

We also need the notion of generating function. Given a function $f : \mathbb{N} \to \mathbb{C}$, its *generating function* is defined as $\sum_{k=0}^{+\infty} f(k) z^k$, for all $z \in \mathbb{C}$ such that $|z| < 1$.

We are now ready to present an algorithm for solving d-PERIODICITY. Let the input be a 1qfa $A = (\pi, U, \eta)$ in linear representation, where π and U have rational entries. Notice that U is a unitary matrix since it is of the form $M \otimes M^*$ for some unitary matrix M. Recall that the event $p_A(n) = \pi U^n \eta$ is d-periodic if and only if $\pi U^n \eta = \pi U^{n+d} \eta$ holds for every $n \in \mathbb{N}$. This condition can be expressed by using the generating function of p_A and, since $(Uz)^n \to \mathbf{0}$ for $z \in \mathbb{C}$ such that $|z| < 1$, we get

$$\sum_{k=0}^{+\infty} (\pi U^k \eta) z^k = \sum_{k=0}^{+\infty} (\pi U^{k+d} \eta) z^k$$

$$\Updownarrow$$

$$\pi \sum_{k=0}^{+\infty} (Uz)^k \eta = \pi U^d \sum_{k=0}^{+\infty} (Uz)^k \eta$$

$$\Updownarrow$$

$$\pi (I - Uz)^{-1} \eta = \pi U^d (I - Uz)^{-1} \eta. \tag{11.2}$$

For any square matrix M, its adjugate matrix $\mathrm{adj}(M)$ is defined as $\mathrm{adj}(M)_{ij} = (-1)^{i+j} \det(M_{[ij]})$, where $M_{[ij]}$ is the matrix obtained from M by erasing its i-th row and j-th column. By recalling that $M^{-1} = \mathrm{adj}(M)^T / \det(M)$, Equation (11.2) becomes

$$\pi \left(\mathrm{adj}(I - Uz)\right)^T \eta = \pi U^d \left(\mathrm{adj}(I - Uz)\right)^T \eta.$$

Notice that both terms of the above equation are rational polynomials of degree less than q, with $q \times q$ being the dimension of U. So, the original problem is reduced to comparing two polynomials with integer coefficients. All the operations involved in this algorithm require polynomial time (see, e.g., [1]). Therefore

Theorem 11.5. *For unary 1qfa's with rational entries, d-*PERIODICITY *is decidable in polynomial time.*

Open problem. It would be interesting to study the decidability, and possibly the complexity, of the variant of d-PERIODICITY, where only a 1qfa A is the input. So, we ask whether p_A is periodic of some period.

11.4 Synthesis of 1qfa's Inducing Periodic Events

We now come to the *synthesis* of succinct 1qfa's exhibiting given periodic events, and we propose an algorithm for solving this task. First, we briefly recall the main steps of this algorithm. Then, we give an *upper and a lower bound* on the number of states of minimal 1qfa's inducing periodic events, both bounds being related to the harmonic structure of the events. Formally, the problem we are to investigate writes as:

SYNTHESIS FROM EVENTS (Syn)

INPUT: An n-periodic event $(p(0), \ldots, p(n-1))$.
OUTPUT: A 1qfa A inducing the event $ap+b$, for some reals $a > 0$, $b \geq 0$, with $a + b \leq 1$.

Thus, we are going to construct 1qfa's inducing not exactly p. However, in Section 11.5 we will show that, from a language recognition point of view, the events p and $ap + b$ are fully equivalent.

Before presenting our synthesis algorithm, we need to recall some formal tools. It is well know that, being periodic, p can be expressed as a linear combination of trigonometric functions by using the discrete Fourier transform and its inverse. More precisely, we have that

$$p(k) = \sum_{j=0}^{n-1} \frac{P(j)}{n} \, \mathrm{e}^{-i\frac{2\pi}{n}kj}, \tag{11.3}$$

with $P(j) = \sum_{k=0}^{n-1} p(k) \, \mathrm{e}^{i\frac{2\pi}{n}kj}$. We let $P = (P(0), \ldots, P(n-1))$ the *discrete Fourier transform of* p. By defining $\mathrm{Supp}(P) = \{j \in \mathbb{Z}_n \mid P(j) \neq 0\}$ the *support set* of P, Equation (11.3) writes as $p(k) = \sum_{j \in \mathrm{Supp}(P)} \frac{P(j)}{n} \, \mathrm{e}^{-i\frac{2\pi}{n}kj}$. The reader is referred to, e.g., [1] (Chp. 7) for more details on the discrete Fourier transform and its relevance from a computational point of view. Here, we just recall that computing the discrete Fourier transform of n-dimensional vectors requires $O(n \log n)$ time.

On the other hand, concerning events induced by 1qfa's, we can show

Lemma 11.6. *Let p be the event induced by a unary 1qfa $A=(\pi, U, \eta)$ with q states. Then, for any $k \in \mathbb{N}$,*

$$p(k) = \sum_{1 \leq s,t \leq q} \mathrm{e}^{ik(\theta_s - \theta_t)} B_{st}, \tag{11.4}$$

where $\mathrm{e}^{i\theta_j}$'s are the eigenvalues of U, and B is an Hermitian positive semidefinite matrix.

Proof. By Equation (11.1) in Section 11.2.2, the event induced by A writes as $p(k) = \sum_{\{j \mid \eta_j = 1\}} |(\pi U^k)_j|^2$, for any $k \in \mathbb{N}$. Since $U \in \mathbb{C}^{q \times q}$ is a unitary matrix, by Proposition 11.1 we can write $U = X \operatorname{diag}(\mathrm{e}^{i\theta_1}, \ldots, \mathrm{e}^{i\theta_q}) X^\dagger$, where X is a unitary matrix and $\mathrm{e}^{i\theta_j}$'s are the eigenvalues of U. So, $U^k = X \operatorname{diag}(\mathrm{e}^{ik\theta_1}, \ldots, \mathrm{e}^{ik\theta_q}) X^\dagger$ and hence

$$p(k) = \sum_{\{j \mid \eta_j = 1\}} |(\pi X \operatorname{diag}(\mathrm{e}^{ik\theta_1}, \ldots, \mathrm{e}^{ik\theta_q}) X^\dagger)_j|^2. \tag{11.5}$$

By letting $\xi = \pi X$ and substituting in (11.5), we get

$$\begin{aligned} p(k) &= \sum_{\{j \mid \eta_j = 1\}} ((\xi_1 \mathrm{e}^{ik\theta_1}, \ldots, \xi_q \mathrm{e}^{ik\theta_q}) X^\dagger)_j \, ((\xi_1 \mathrm{e}^{ik\theta_1}, \ldots, \xi_q \mathrm{e}^{ik\theta_q}) X^\dagger)_j^* \\ &= \sum_{\{j \mid \eta_j = 1\}} \left(\sum_{s=1}^q \xi_s \mathrm{e}^{ik\theta_s} X_{sj}^\dagger \right) \left(\sum_{t=1}^q \xi_t^* \mathrm{e}^{-ik\theta_t} (X_{tj}^\dagger)^* \right) \\ &= \sum_{1 \leq s,t \leq q} \mathrm{e}^{ik(\theta_s - \theta_t)} \sum_{\{j \mid \eta_j = 1\}} \xi_s X_{sj}^\dagger (\xi_t X_{tj}^\dagger)^*. \end{aligned}$$

Now, define the matrix B as

$$B_{st} = \sum_{\{j \mid \eta_j = 1\}} \xi_s X_{sj}^\dagger (\xi_t X_{tj}^\dagger)^*,$$

for $1 \leq s, t \leq q$. It is easy to verify that $B = B^\dagger$, and hence B is Hermitian. To prove that B is positive semidefinite, by Proposition 11.3(i), it is enough to show that $xBx^\dagger \geq 0$, for any $x \in \mathbb{C}^{1 \times q}$:

$$\begin{aligned} xBx^\dagger &= \sum_{1 \leq s,t \leq q} x_s \left(\sum_{\{j \mid \eta_j = 1\}} \xi_s X_{sj}^\dagger (\xi_t X_{tj}^\dagger)^* \right) x_t^* \\ &= \sum_{\{j \mid \eta_j = 1\}} \left(\sum_{s=1}^q x_s \xi_s X_{sj}^\dagger \right) \left(\sum_{t=1}^q x_t \xi_t X_{tj}^\dagger \right)^* \\ &= \sum_{\{j \mid \eta_j = 1\}} \left| \sum_{s=1}^q x_s \xi_s X_{sj}^\dagger \right|^2 \geq 0. \end{aligned}$$

□

In what follows, we use the notion of difference cover [15]. A set $\Delta \subseteq \mathbb{Z}_n$ is a *difference cover in* $\mathbb{Z}_n$ (DC_n, for short) for a set $X \subseteq \mathbb{Z}_n$ if, for each $x \in X$, there exist two elements $a, b \in \Delta$ such that $x = (a - b) \bmod n$.

We are now ready to propose an algorithm for solving Syn. The algorithm consists of two parts: In the first part, θ's and B are computed so that Equation (11.4) exactly reproduces Equation (11.3). In the second part, such a choice is turned into a well formed 1qfa inducing $ap + b$. Here is the *first part*:

```
∘ INPUT $(p(0), \ldots, p(n-1))$
STEP 1: Compute $P = (P(0), \ldots, P(n-1))$, the discrete Fourier transform
    of $p$, and let $\mathrm{Supp}(P) = \{j \in \mathbb{Z}_n \mid P(j) \neq 0\}$.
STEP 2: Find a minimum $\mathrm{DC}_n$ $\Delta = \{d_1, \ldots, d_q\}$ for $\mathrm{Supp}(P)$.
STEP 3: For each $1 \leq t \leq q$, let $\theta_t = -\frac{2\pi}{n} d_t$.
STEP 4: For each $j \in \mathrm{Supp}(P)$, let
        $N(j) = |\{(d_s, d_t) \mid d_s, d_t \in \Delta \text{ and } j = (d_s - d_t) \bmod n\}|$,
    and, for $1 \leq s, t \leq q$, compute
        $B_{st} = \begin{cases} \frac{1}{n}\frac{P(j)}{N(j)} & \text{if } j \in \mathrm{Supp}(P) \text{ and } j = (d_s - d_t) \bmod n \\ 0 & \text{otherwise.} \end{cases}$
```

It is easy to check that $B \in \mathbb{C}^{q \times q}$ *is an Hermitian matrix:* to see that $B_{st} = {B_{ts}}^*$, it is enough to notice that $P(j) = P^*(-j \bmod n)$ and $N(j) = N(-j \bmod n)$, for each $j \in \mathbb{Z}_n$. By plugging θ's obtained at STEP 3 and B obtained at STEP 4 into Equation (11.4), we get exactly $p(k)$ as in Equation (11.3).

We are not going to display the (quite technical) *second part* of the algorithm which produces the desired 1qfa A inducing the event $ap + b$ for some real constants $a > 0$, $b \geq 0$, with $a + b \leq 1$. The reader can find all the details of this part in [27] where, in particular, the explicit evaluation of the coefficients a and b is performed. We just recall that *the number of states of* A *turns out to be* $2q$ *or* $4q+1$, depending on whether the Hermitian matrix B obtained in the first part is positive semidefinite or not, respectively.

Instead, to emphasize the succinctness of returned 1qfa's, we are now going to show that *this algorithm yields a minimal 1qfa — within a constant factor — inducing* $ap + b$. To see this, we first need the following

Lemma 11.7. *Given a* n*-periodic event* p*, let* $ap + b$ *be induced by a unary* q*-state 1qfa, for some reals* $a > 0$*,* $b \geq 0$*,* $a + b \leq 1$*. Then, there exists a* DC_n *for* $\mathrm{Supp}(P)$ *whose cardinality does not exceed* q*.*

Proof. Since $ap + b$ is induced by a 1qfa, it must expand as in Equation (11.4). Yet, being an n-periodic function, it can also be expressed by discrete Fourier transform as in Equation (11.3). By equating these two forms, we get the following equivalence for every $k \geq 0$:

$$\sum_{1 \leq s,t \leq q} \mathrm{e}^{ik(\theta_s - \theta_t)} \rho_{st} \mathrm{e}^{i\alpha_{st}} = \sum_{j=0}^{n-1} \rho_j \mathrm{e}^{i\beta_j} \mathrm{e}^{-i\frac{2\pi}{n}kj} + b,$$

where we choose to let $\rho_{st}\mathrm{e}^{i\alpha_{st}} = B_{st}$, $\rho_j \mathrm{e}^{i\beta_j} = aP(j)/n$, with reals $\rho_{st}, \rho_j, \alpha_{st}, \beta_j$. Hence, for each $j \in \mathrm{Supp}(P)$, there must exist $1 \leq u, v \leq q$ such that

$$\mathrm{e}^{ik(\theta_u - \theta_v)} \mathrm{e}^{i\alpha_{uv}} = \mathrm{e}^{-i\frac{2k\pi}{n}j} \mathrm{e}^{i\beta_j}.$$

Since this equality holds for every $k \geq 0$, it must be that $\theta_v - \theta_u = \frac{2\pi}{n}j$. Therefore, we can set a DC_n for $\mathrm{Supp}(P)$ as $\Delta = \{\lfloor n\theta_t/(2\pi) \rfloor \bmod n \mid 1 \leq t \leq q\}$. $\square$

We are now ready to give a lower bound on the number of states for 1qfa's inducing $ap + b$.

Theorem 11.8. *Let p be a n-periodic event. Every 1qfa inducing the event $ap + b$, for some reals $a > 0$, $b \geq 0$, $a + b \leq 1$, cannot have less states than the cardinality of a minimum* DC_n *for* $\mathrm{Supp}(P)$*.*

Proof. Suppose, by contradiction, that $ap+b$ is induced by a 1qfa with a number of states less than the cardinality κ of a minimum DC_n for $\mathrm{Supp}(P)$. This, by Lemma 11.7, would imply the existence of a DC_n for $\mathrm{Supp}(P)$ of cardinality less than or equal to κ, which is clearly impossible. $\square$

In the next section, we discuss the above synthesis algorithm from a time complexity point of view. We are going to show that, as it is, the algorithm is inefficient. However, we provide a "relaxed version" running in polynomial time, still guaranteeing a quadratic gain on the size of the resulting 1qfa's with respect to equivalent classical devices.

Open problem. As stated by Theorem 11.8, the cardinality of a minimum DC_n for the support set of a n-periodic event p represents a lower bound to the number of states of any 1qfa inducing p. It would be interesting to characterize those (or at least single out families of) periodic events for which such a theoretical lower bound can actually be achieved.

11.4.1 *The Time Complexity of our Synthesis Algorithm*

It is not hard to see that all the steps of our synthesis algorithm can be efficiently performed except STEP 2, where the computation of a minimum difference cover is required. Efficiently solving this task would lead to a feasible synthesis algorithm for succinct 1qfa's inducing periodic events. Unfortunately, we are going to show that computing a minimum difference covers is NP-hard. Formally, we consider the following optimization problem:

MINIMUM DIFFERENCE COVER (MinDC_n)
INPUT: $n \in \mathbb{Z}^+$, $X \subseteq \mathbb{Z}_n \backslash \{0\}$
OUTPUT: $\Delta = A \cup \{0\}$, with $A \subset \mathbb{Z}_n$, such that Δ is a DC_n for X
MEASURE: Cardinality of the DC_n, i.e., $|\Delta|$

By a simple counting argument, we get that the cardinality of a minimum DC_n Δ satisfies $(1+\sqrt{4|X|-3})/2 \leq |\Delta| \leq |X|+1$. Those subsets of $\mathbb{Z}_n$ having a minimum DC_n with cardinality matching the upper bound are called *n-extrema.* We begin by studying the hardness of *testing non-extremity* for subsets of $\mathbb{Z}_n$.

To this aim, we consider the corresponding relaxed problem for subsets of $\mathbb{Z}^+$, with differences performed without mod operation (i.e., difference covers are in $\mathbb{Z}^+$). In this case, we call *extremum* a set $Y \subset \mathbb{Z}^+$ having a minimum difference covers in $\mathbb{Z}^+$ of cardinality $|Y|+1$. The following characterizations of n-extrema and extrema can be proved:

Theorem 11.9. [28] (Thms. 4.1, 4.2)

(i) Let $X = \{x_1, x_2, \ldots, x_m\} \subset \mathbb{Z}_n$, and let $a_1, a_2, \ldots, a_m$ variables on $\{-1, 0, 1\}$. Then X is a n-extremum if and only if

$$\left(\sum_{k=1}^{m} a_k x_k\right) \bmod n = 0 \quad \Leftrightarrow \quad a_k = 0, \ \textit{for every } 1 \leq k \leq m.$$

(ii) Let $Y = \{y_1, y_2, \ldots, y_m\} \subset \mathbb{Z}^+$, and let $a_1, a_2, \ldots, a_m$ be variables on $\{-1, 0, 1\}$. Then Y is an extremum if and only if

$$\sum_{k=1}^{m} a_k y_k = 0 \quad \Leftrightarrow \quad a_k = 0, \ \textit{for every } 1 \leq k \leq m.$$

Example 11.10. Consider the set $E_\rho = \{1, \rho^1, \ldots, \rho^\alpha\}$, for any given integer $\rho \geq 2$. We can prove that E_ρ is an extremum, and a n-extremum for any $n > \frac{\rho^{\alpha+1}-1}{\rho-1}$. First, it is easy to see that $\sum_{k=0}^{\alpha} a_k\rho^k = 0$ if and only if every $a_k \in \{-1,0,1\}$ is 0. Otherwise, we could write $\sum_{\{i \mid a_i=1\}} \rho^i = \sum_{\{i \mid a_i=-1\}} \rho^i$. Since any number has a unique representation in base ρ, we would get a contradiction. So, by Theorem 11.9(ii), E_ρ is an extremum. To see that E_ρ is a n-extremum for $n = \frac{\rho^{\alpha+1}-1}{\rho-1} + 1$, we apply Theorem 11.9(i), noticing that $\sum_{k=0}^{\alpha} \rho^k = n-1$.

Thus, according to Theorem 11.9(ii), deciding whether Y is *not* an extremum is *equivalent* to decide whether there exists a nonzero assignment on $\{-1,0,1\}$ to a_k's, satisfying the equation $\sum_{k=1}^{m} a_k y_k = 0$. We call this latter decision problem $\mathsf{Ass}(-1,0,1)$. To study its hardness, we find it useful to introduce a more general problem $\mathsf{Sys}(-1,0,1)$, where the input is a system of equations

$$\mathcal{S} = \left\{ \sum_{k=1}^{n} w_k^{(t)} x_k = 0 \right\}_{0\le t\le \hat{n}}$$

in the n variables x_k taking values in $\{-1,0,1\}$, with coefficients $w_k^{(t)} \in \mathbb{N}$, and $\hat{n} = n^{O(1)}$. This problem asks whether there exists an assignment in $\{-1,0,1\}$ of x_k's satisfying $\mathcal{S}$ and such that not all x_k's are set to 0.

We let $\le_p$ denote the polynomial time many-one reduction between decision problems (see, e.g., [18]). We show that

Lemma 11.11. $\mathsf{Sys}(-1,0,1) \le_p \mathsf{Ass}(-1,0,1)$.

Proof. We reduce the system of equations $\mathcal{S} = \left\{ \sum_{k=1}^{n} w_k^{(t)} x_k = 0 \right\}_{0\le t\le \hat{n}}$ to a single equation $\mathcal{E}(x_1,\ldots,x_n) = 0$, such that any assignment on $\{-1,0,1\}$ to x_k's is a solution of $\mathcal{E}$ if and only if it is a solution of $\mathcal{S}$. To this purpose, we set $W = 1 + \max_t \sum_{k=1}^{n} w_k^{(t)}$, and define $\mathcal{E}(x_1,\ldots,x_n) = 0$ as

$$\sum_{k=1}^{n} w_k^{(0)} x_k + W \sum_{k=1}^{n} w_k^{(1)} x_k + W^2 \sum_{k=1}^{n} w_k^{(2)} x_k + \cdots + W^{\hat{n}} \sum_{k=1}^{n} w_k^{(\hat{n})} x_k = 0. \tag{11.6}$$

Clearly, any assignment satisfying $\mathcal{S}$ satisfies $\mathcal{E}$ as well. Vice versa, suppose we have an assignment of x_k's satisfying Equation (11.6). For the sake of readability, let $H_i = \sum_{k=1}^{n} w_k^{(i)} x_k$, so that we can rewrite Equation (11.6) as

$$H_0 = -W \left(\sum_{i=1}^{\hat{n}} H_i W^{i-1} \right).$$

Hence, W divides H_0, but since $-W < H_i < W$ for each $0 \leq i \leq \hat{n}$, we must conclude that $H_0 = 0$. By iterating this reasoning, we get that the assignment satisfying Equation (11.6) satisfies each H_i as well. We end by quickly noticing that computing $\mathcal{E}$ from $\mathcal{S}$ is easily seen to be done in polynomial time. $\square$

Now, we need to recall the well known NP-complete problem Partition (see, e.g., [18] (Ch. 3)). Here, we formulate such a problem in a slightly modified but perfectly equivalent version which is more suited to our purposes:

Partition
Input: Finite set $Y = \{y_1, y_2, \ldots, y_m\} \subset \mathbb{Z}^+$.
Output: Is there an assignment on $\{-1, 1\}$ to b_k's s.t. $\sum_{k=1}^{m} b_k y_k = 0$?

In other words, we ask whether Y can be split into two subsets of equal sum.

Lemma 11.12. Partition $\leq_p$ Sys$(-1, 0, 1)$.

Proof. Let $Y = \{y_1, \ldots, y_m\} \subset \mathbb{Z}^+$ be an input instance of Partition. We construct the following system of $m+1$ equations in the $2m+1$ variables $\{b_1, b_2, \ldots, b_m, c_1, c_2, \ldots, c_m, a\}$ ranging on $\{-1, 0, 1\}$:

$$\mathcal{S}_Y = \begin{cases} \sum_{k=1}^{m} b_k y_k = 0 \\ b_1 + 2c_1 + a = 0 \\ b_2 + 2c_2 + a = 0 \\ \vdots \\ b_m + 2c_m + a = 0. \end{cases}$$

Now, notice that any *solution for $\mathcal{S}_Y$ either has all the variables set to 0 (i.e., is the trivial one) or is on* $\{-1, 1\}$ *only.* In fact, take a solution ζ where $b_i = 0$, for a given $1 \leq i \leq m$. Since all the variables range only on $\{-1, 0, 1\}$, the corresponding equation $b_i + 2c_i + a = 0$ has a unique solution for $a = 0$ and $c_i = 0$. In turn, $a = 0$ yields the equations $b_k + 2c_k = 0$, for $1 \leq k \leq m$, giving that ζ must set all the variables to 0. This reasoning shows that any possible nontrivial solution for $\mathcal{S}_Y$ yields a solution in $\{-1, 1\}$ for the equation $\sum_{k=1}^{m} b_k y_k = 0$, and hence represents a partition of Y. Vice versa, it is clear that any possible partition of Y can be immediately transformed into a solution for the corresponding system $\mathcal{S}_Y$. It is enough to add $a \in \{-1, 1\}$, and $c_k = (-a - b_k)/2$ for every $1 \leq k \leq m$. The construction of $\mathcal{S}_Y$ from Y is easily seen to be performed in polynomial time, and this completes the proof. $\square$

We are now ready to prove the NP-completeness of testing non-extremity.

Theorem 11.13. *Deciding whether a set* $Y = \{y_1, y_2, \ldots, y_m\} \subset \mathbb{Z}^+$ *is not an extremum is* NP-complete.

Proof. As above recalled, such a decision problem is equivalent to $\mathsf{Ass}(-1, 0, 1)$ for the equation $\sum_{k=1}^{m} a_k y_k = 0$. A polynomial time nondeterministic algorithm for solving this latter problem simply guesses a nonzero assignment on $\{-1, 0, 1\}$ to a_k's, and then checks in polynomial time whether the assignment satisfies the equation. This shows that $\mathsf{Ass}(-1, 0, 1)$ belongs to NP. From Lemma 11.11 and Lemma 11.12, we get that $\mathsf{Partition} \leq_p \mathsf{Sys}(-1, 0, 1) \leq_p \mathsf{Ass}(-1, 0, 1)$. The result follows from the NP-completeness of $\mathsf{Partition}$. $\square$

This latter result enables us to obtain the NP-completeness even for testing non-extremity in $\mathbb{Z}_n$.

Theorem 11.14. *Deciding whether a subset of* $\mathbb{Z}_n$ *is not a* n*-extremum is* NP-complete.

Proof. By the characterization of n-extrema in Theorem 11.9(i), one may easily design a nondeterministic polynomial time algorithm for testing non-extremity in $\mathbb{Z}_n$, thus setting this problem in NP. To show its completeness, by Theorem 11.13, it is enough to exhibit a reduction from testing non-extremity in $\mathbb{Z}^+$. Our reduction works as follows: given the instance $Y = \{y_1, \ldots, y_m\} \subset \mathbb{Z}^+$, return the instance $Y = \{y_1, \ldots, y_m\} \subset \mathbb{Z}_n$, with $n = 1 + \sum_{i=1}^{m} y_i$. We must show that Y is not an extremum if and only if Y is not a n-extremum. For every assignment on $\{-1, 0, 1\}$ to a_k's, we have $-n < \sum_{k=1}^{m} a_k y_k < n$. Hence $(\sum_{k=1}^{m} a_k y_k) \bmod n = 0 \Leftrightarrow \sum_{k=1}^{m} a_k y_k = 0$. By Theorem 11.9 the result follows. $\square$

As a final step, we are able to establish the complexity of MinDC_n.

Theorem 11.15. MinDC_n *is* NP-hard.

Proof. Theorem 11.14 states that testing non-extremity in $\mathbb{Z}_n$ is NP-complete. Thus, the claimed result can be shown by exhibiting a polynomial time Turing-reduction (see, e.g., [18]) from this decision problem to MinDC_n. It is easy to exhibit a Turing machine that decides in polynomial time whether a given $X \subset \mathbb{Z}_n$ is not a n-extremum by having an oracle for MinDC_n. First, we use such an oracle to compute a minimum DC_n for X, then we check whether its cardinality is less than $|X| + 1$. $\square$

11.4.2 *Efficient Synthesis*

The result in Theorem 11.15 shows that, although providing a minimal, up to a constant factor, 1qfa for periodic events, our synthesis algorithm is in general unfeasible due to the intrinsic difficulty of STEP 2, i.e., computing a minimum DC_n. However, there exist two interesting variants of the algorithm, leading to feasible time complexities:

(1) In [7], nontrivial families of sets for which the computation of minimum DC_n's can be performed in polynomial time are pointed out. Hence, for those periodic events having such sets as supports, the resulting 1qfa's are the smallest possible (up to a constant factor), and constructed in polynomial time.

(2) We can relax STEP 2 by asking a general (not necessarily a minimum) difference cover for the support set of input periodic events. The following result ensures that such a task can be efficiently performed:

Theorem 11.16. [15](Thm. 2.4) *For any $n \geq 0$, there exists a DC_n for $\mathbb{Z}_n$ of cardinality at most $\sqrt{1.5n} + 6$, that can be built in polynomial time.*

As a consequence, we obtain in polynomial time a $O(\sqrt{n})$ state 1qfa inducing any given n-periodic event. Thus, we get time efficiency paying by a possible size increase. However, as we will notice in the next section, the resulting 1qfa's are more succinct than corresponding classical devices.

Open problem. It would be interesting to study the complexity of constructing minimal 1qfa's inducing given periodic events. Our results in Section 11.4.1 suggest the NP-hardness of this task.

11.5 Application to Periodic Languages

In this concluding section, we quickly address an *application* of the above synthesis results to the recognition of periodic languages, i.e., unary languages in the form $L = \{k \in \mathbb{N} \mid (k \bmod n) \in S\}$, for a fixed $S \subseteq \mathbb{Z}_n$.

It is well known that for accepting n-periodic languages by one-way deterministic or nondeterministic automata, n states are necessary and sufficient. Moreover, in some cases, when n is a prime power, even using one-way probabilistic automata (or also two-way nondeterminism) does not help in saving states [30]. Our results in the previous section enable us to show that also in this latter "hard" cases the quantum paradigm leads to a quadratic size improvement:

Theorem 11.17. *Any n-periodic language can be accepted with isolated cut point by a 1qfa with $O(\sqrt{n})$ states.*

Proof. With each n-periodic language $L = \{k \in \mathbb{N} \mid (k \bmod n) \in S\}$, with $S \subseteq \mathbb{Z}_n$, we can associate the n-periodic event p defined, for every $k \geq 0$, as

$$p(k) = \begin{cases} 1 \text{ if } (k \bmod n) \in S \\ 0 \text{ otherwise.} \end{cases}$$

So, p is simply the characteristic function of L. By the results in Section 11.4.2, we can construct in polynomial time a 1qfa A with $O(\sqrt{n})$ states, inducing $ap + b$, for some reals $a > 0$, $b \geq 0$, $a + b \leq 1$. It is easy to see that A accepts L with cut point $(a + 2b)/2$ isolated by $a/2$. $\square$

Thus, the previous theorem settles a general quantum quadratic superiority from a size point of view (both for accepting periodic languages and inducing periodic events). Actually, for certain families of periodic languages and using different techniques, an exponential size decrease can also be reached [2, 10].

Nevertheless, some lower limits to the descriptional power of 1qfa's can be stated. In fact, in what follows we are going to show the existence of periodic languages that cannot be accepted by 1qfa's with less than $\sqrt{n/(3 \log n)}$ states (even dropping the condition of isolation around the cut point).

Theorem 11.18. *There exist n-periodic languages that cannot be accepted by any 1qfa inducing a n-periodic event with less than $\sqrt{n/(3 \log n)}$ states.*

Proof. Let us give an upper bound to the number of different n-periodic languages accepted by q-state 1qfa's inducing n-periodic events. Let p be an n-periodic event induced by a 1qfa with q states. By Lemma 11.6 and Lemma 11.7, there exist a matrix $C \in \mathbb{C}^{q \times q}$ and a set $\Delta = \{a_1, a_2, \ldots, a_q\} \subseteq \mathbb{Z}_n$ such that

$$p(k) = \sum_{1 \leq s,t \leq q} \mathrm{e}^{i \frac{2\pi}{n}(a_s - a_t)k} C_{st}.$$

Without loss of generality, paying by one extra-state [30], we can assume that our 1qfa's accept with cut point 1/2. We can choose Δ in $\binom{n}{q}$ different ways, each one yielding n hyperplanes of the form

$$\sum_{1\le s,t\le q} D_{st}\cos\left(\frac{2\pi}{n}k(a_s-a_t)\right)+E_{st}\sin\left(\frac{2\pi}{n}k(a_s-a_t)\right)=\frac{1}{2},$$

for $0 \le k < n$ and reals D_{st}, E_{st}. These n hyperplanes lay in a $2q^2$ dimensional space S and can divide S in at most n^{2q^2} different regions [12]. The event induced by coefficients D_{st} and E_{st} in the same region define the same language. This implies that the number of n-periodic languages accepted by q-state 1qfa's is bounded above by $\binom{n}{q}n^{2q^2} < n^{3q^2}$.

By noticing that the number of distinct n-periodic languages is 2^n, we must require that $n^{3q^2} \ge 2^n$, in order to accept every n-periodic language by using q-state 1qfa's. Such an inequality is easily seen to yield $q \ge \sqrt{n/(3\log n)}$. $\square$

References

1. Aho, A., Hopcroft, J. and Ullman, J. (1974). *The design and analysis of computer algorithms* (Addison-Wesley).
2. Ambainis, A. and Freivalds, R. (1998). 1-way quantum finite automata: strengths, weaknesses and generalizations, in *Proc. 39th Symposium on Foundations of Computer Science*, pp. 332–342.
3. Ambainis, A. and Watrous, J. (2002). Two-way finite automata with quantum and classical states, *Theoretical Computer Science* **287**, pp. 299–311.
4. Benioff, P. (1982). Quantum mechanical Hamiltonian models of Turing machines, *J. Stat. Phys.* **29**, pp. 515–546.
5. Bernstein, E. and Vazirani, U. (1997). Quantum complexity theory, *SIAM J. Comput.* **26**, pp. 1411–1473.
6. Bertoni, A. and Carpentieri, M. (2001). Regular languages accepted by quantum automata, *Information and Computation* **165**, pp. 174–182.
7. Bertoni, A., Mereghetti, C. and Palano, B. (2003a). Golomb rulers and difference sets for succinct quantum automata, *International Journal of Foundations of Computer Science* **14**, pp. 871–888.
8. Bertoni, A., Mereghetti, C. and Palano, B. (2003b). Lower bounds on the size of quantum automata accepting unary languages, in *Proc. 8th Italian Conf. Theor. Comp. Sci.*, LNCS 2841 (Springer), pp. 86–95.
9. Bertoni, A., Mereghetti, C. and Palano, B. (2003c). Quantum computing: 1-way quantum automata, in *Proc. 7th Int. Conf. Developments in Language Theory*, LNCS 2710 (Springer), pp. 1–20.
10. Bertoni, A., Mereghetti, C. and Palano, B. (2005). Small size quantum automata recognizing some regular languages, *Theor. Comp. Sci.* **340**, pp. 394–407.
11. Bertoni, A., Mereghetti, C. and Palano, B. (2010). Trace monoids with idempotent generators and measure-only quantum automata, *Natural Computing,* to be published.

12. Bertoni, A. and Torelli, M. (1977). *Elementi di matematica combinatoria* (ISEDI).
13. Bianchi, M. and Palano, B. (2009). Events and languages on unary quantum automata, in *Proc. 1st Workshop on Non-Classical Models of Automata and Applications* (Österreichischen Computer Gesellschaft), pp. 61–75.
14. Brodsky, A. and Pippenger, N. (2002). Characterizations of 1-way quantum finite automata, *SIAM J. Comput.* **5**, pp. 1456–1478.
15. Colbourn, C. and Ling, A. (2000). Quorums from difference covers, *Information Processing Letters* **75**, pp. 9–12.
16. Deutsch, D. (1985). Quantum theory, the church-turing principle and the universal quantum computer, *Proc. Roy. Soc. London* **400**, pp. 97–117.
17. Feynman, R. (1982). Simulating physics with computers, *Int. J. Theoretical Physics* **21**, pp. 467–488.
18. Garey, M. R. and Johnson, D. S. (1979). *Computers and intractability. A guide to the theory of NP-completeness* (W.H. Freeman).
19. Golovkins, M., Kravtsev, M. and Kravcevs, V. (2009). On a class of languages recognizable by probabilistic reversible decide-and-halt automata, *Theoretical Computer Science* **410**, pp. 1942–1951.
20. Grover, L. (1996). A fast quantum mechanical algorithm for database search, in *Proc. 28th ACM Symp. Theor. Comp.*, pp. 212–219.
21. Gruska, J. (1999). *Quantum computing* (McGraw-Hill).
22. Hopcroft, J. and Ullman, J. (1979). *Introduction to automata theory, languages, and computation* (Addison-Wesley).
23. Kondacs, A. and Watrous, J. (1997). On the power of quantum finite state automata, in *Proc. 38th Symp. Found. Comp. Sci.*, pp. 66–75.
24. Lenstra, A. K. and H. W. Lenstra, J. (eds.) (1993). *The development of the number field sieve*, *Lecture Notes Math.*, Vol. 1554 (Springer-Verlag).
25. Marcus, M. and Minc, H. (1964). *A survey of matrix theory and matrix inequalities* (Prindle, Weber & Schmidt), reprinted by Dover, 1992.
26. Marcus, M. and Minc, H. (1965). *Introduction to linear algebra* (The Macmillan Company), reprinted by Dover, 1988.
27. Mereghetti, C. and Palano, B. (2002). On the size of one-way quantum finite automata with periodic behaviors, *Theor. Inf. Appl.* **36**, pp. 277–291.
28. Mereghetti, C. and Palano, B. (2006). The complexity of minimum difference cover, *Journal of Discrete Algorithms* **4**, pp. 239–254.
29. Mereghetti, C. and Palano, B. (2007). Quantum automata for some multiperiodic languages, *Theoretical Computer Science* **387**, pp. 177–186.
30. Mereghetti, C., Palano, B. and Pighizzini, G. (2001). Note on the succinctness of deterministic, nondeterministic, probabilistic and quantum finite automata, *Theor. Inf. Appl.* **35**, pp. 477–490.
31. Moore, C. and Crutchfield, J. (2000). Quantum automata and quantum grammars, *Theoretical Computer Science* **237**, pp. 275–306.
32. Nakahara, M., Kanemitsu, S., Salomaa, M. M. and Takagi, S. (eds.) (2006). *Physical realizations of quantum computing* (World Scientific).
33. Nayak, A. (1999). Optimal lower bounds for quantum automata and random access codes, in *Proc. 40th Symp. Found. Comp. Sci.*, pp. 369–376.

34. Nielsen, M. A. and Chuang, I. L. (2000). *Quantum computation and quantum information* (Cambridge).
35. Pin, J. E. (1987). On languages accepted by finite reversible automata, in *14th Int. Coll. Aut., Lang. and Prog.*, LNCS 267 (Springer-Verlag), pp. 237–249.
36. Shor, P. (1997). Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer, *SIAM J. Comput.* **26**, pp. 1484–1509.

Chapter 12

Soliton Circuits and Network-Based Automata: Review and Perspectives

Miklós Bartha
Memorial University of Newfoundland, St. John's, Canada,
E-mail: bartha@mun.ca

Miklós Krész
University of Szeged, Szeged, Hungary,
E-mail: kresz@jgypk.u-szeged.hu

The study of various network related processes has recently become a key issue in many science disciplines including biology, chemistry, and physics, but also in sociology and other areas dealing with intensive communication. Different models are sought to describe such processes, among which automata appear to be successful candidates.

Percolation processes, in particular, are of distinguished interest, and the results reported in this paper deal with a special process of this nature. A state of the network is represented by a matching of the underlying graph, while state transitions are induced by certain alternating walks in the network. This model was introduced in 1990 by Dassow and Jürgensen under the name soliton automaton, with the aim to capture the phenomenon of molecular switching.

Soliton graphs and automata have since been studied in the context of matching theory by the authors in a series of works. The present paper is mainly a review of the most important results originating from this study, and it also outlines some future perspectives and generalizations of this interesting model.

12.1 Introduction

Molecular computing ([1, 61, 65]) is an emerging field in computer science, generating a large number of interesting models trying to capture the main features of this phenomenon both at the level of theory and applications. The idea of molecular memories goes back to Feynman's pioneer paper (cf. [29]), in which he proposes building small machines and then using those machines to build yet smaller machines and so on, down to the molecular level. During the past decades several promising concepts have been worked out for unconventional computing. Among these, nonlinear media that exhibit self-localized mobile patterns in their evolution are potential candidates to serve as universal dynamical computers. The computational model called soliton cellular automaton (cf. [59, 60, 62]) uses soliton interactions in the design of collision-based logic gates. The word soliton (or solitary wave) refers to certain types of waves traveling a relatively large distance with little energy loss. For a survey of unconventional architectures in the framework of molecular computing, see [1].

Other alternatives of molecular computers are based on the design of conventional digital circuits at the molecular level [22]. The idea of this approach is as follows. If the switching elements were built from molecular scale ingredients, then it would be possible to make circuits thousands of times smaller. These circuits would use appropriate molecules as electronic switches and would be interconnected by some sort of ultra-fine conducting wires. One interesting choice of these conductors was proposed by Carter [21], making use of single strands of the electrically conductive plastic polyacetylene, along which soliton waves can travel. For this reason, molecular scale electronic devices constructed from molecular switches and polyacetylene chains are called soliton circuits.

Practical research concerning soliton circuits (see e.g. [36–39]) has evoked the need to develop an applied mathematical arsenal in order to obtain a thorough understanding of the behavior of these circuits. The first mathematical model of soliton circuits, called soliton automata, was introduced in [24], but it was not until the publishing of [7] that matching theory [55] was recognized as the fundamental theoretical background for the study of this model. The underlying object of a soliton automaton represents the topological structure of a molecule or a chain of molecules. It is an undirected graph, called a soliton graph, which comes with a matching that covers all vertices with degree at least two. These, so called internal vertices model groups of carbon and hydrogen atoms, whereas vertices with degree one – called external – represent a suitable chemical interface with the outside world. The matching captures the pattern of single and double bonds within the molecule, which is considered the current state of the corresponding automaton. The state is changed by making an alternating walk from one external vertex to another.

Network analysis [2] is a classical topic of mathematics and computer science, but the in-depth study of network structures and network dynamics has become a central issue only recently [3, 56, 57]. The aim of this study is to describe the common features of network models inspired by various applications (sociological, communication, biological etc. networks).

One of the most important problems in network research is the description of different processes taking place in networks, among which percolation [58] is an interesting special case. During a percolation process, certain vertices and/or edges in the graph representing the network are marked in every state, and the changing of the state results in a new marking situation. Our soliton automaton model can be viewed as a percolation process by which there is a fixed bound (i.e., 1) for the number of marked edges incident with any given vertex.

In order to analyze network processes, one needs an appropriate discrete event system model [23]. Automata are among the most popular tools for this purpose. It is therefore a natural approach to work out automaton-based models for percolation processes and provide a logical description for these automata. Most network models in the literature are closed systems, in the sense that state changes are controlled from within the system. In several applications, however, it is important to distinguish certain interface points, through which information can flow in and out of the system. As main examples, sociological, communication, and biological networks are frequently regarded as open systems in the above sense. Capturing the characteristics of such open systems then becomes a significant issue.

This paper intends to be the starting point of an ambitious research studying open discrete event percolation systems using automata theoretic tools. We have chosen a simple process in which each non-interface vertex of the network is incident with a unique marked edge in every state. In chemistry, groups of atoms covalently bonded with alternating single and double bonds in a molecule of an organic compound are known as conjugated systems. For this reason we adopt the name "open conjugated system" to identify the type of percolation systems we are concerned with.

Matching theory has been used before for the study of chemical conjugated systems matchings (cf. [55]). Graphs representing the topological structure of molecules possessing an alternating pattern of single and double bonds are also known as Hückel graphs. The difference in our approach is the assumption that the graph is open, so that the external (interface) vertices need not be covered by the matchings of interest.

12.2 Basic Concepts

In this section we review the most important definitions related to soliton automata. Since our model builds on a combination of matching theory and automata theory, these two main components will be discussed separately in two subsections.

12.2.1 *Perfect Internal Matchings in Graphs*

Throughout the paper, our notation and terminology with respect to graphs and matchings will be compatible with that of [55].

By a graph we mean a finite undirected graph in the most general sense with multiple edges and loops edges allowed. For a graph G, $V(G)$ and $E(G)$ will denote the set of vertices and the set of edges of G, respectively. The concepts walk, cycle, and path will be meant in the usual way, with their length being the number of edges in them. In notation, a walk is an alternating sequence of vertices and edges $(v_0, e_1, v_1, \ldots, v_n)$, where each edge is incident with the vertex immediately preceding and following it. If $n = 0$, then the walk is called *empty* . Also as usual, a *trail* is a walk in which all edges are distinct.

If $V(G)$ can be partitioned into two disjoint non-empty sets A and B such that all edges of G connect a vertex from A to a vertex from B, then we call G *bipartite* and refer to A, B as the *bipartition* of G.

A vertex $v \in V(G)$ is called *external* if its degree $d(v)$ is one or zero, and *internal* if $d(v) \geq 2$. It is understood that a loop edge around v contributes with both of its endpoints to the count $d(v)$. The sets of external and internal vertices of G will be denoted by Ext_G and Int_G, respectively. *External edges* are those of $E(G)$ that are incident with at least one external vertex, and *internal edges* are those connecting two internal vertices. Graph G is called *open* if it has at least one external vertex, otherwise G is called *closed*. See Figure 12.1 for an open graph with external vertices u and v, and external edges e and f.

Let us fix a graph G for the rest of this subsection. A *matching* M of

G is a subset of $E(G)$ such that no vertex of G occurs more than once as an endpoint of some edge in M. Again, it is understood by this definition that loops are not allowed to participate in M. The endpoints of the edges contained in M are said to be *covered* by M . A *perfect internal matching* is a matching that covers all of the internal vertices, and a *perfect matching* is just a perfect internal matching of a closed graph. For open graphs the concept perfect matching is not defined.

An edge $e \in E(G)$ is *allowed (mandatory)* if e is contained in some (respectively, all) perfect internal matching(s) of G. *Forbidden edges* are those that are not allowed. We shall also use the term *constant edge* to identify an edge that is either forbidden or mandatory. As an example, consider the graph G in Figure 12.1. The set $\{e, h_1, h_2\}$ determines a perfect internal matching of G and, with the only exception of the forbidden edge g, each edge is allowed in G.

By the usual definition, a *subgraph* G' of G is just a collection of vertices and edges of G. For a set $X \subseteq V(G)$, the (full) *subgraph induced by* X is the one having X as vertices and $E(G) \cap (X \times X)$ as edges. Since external vertices play a distinguished role in our model, we do not want to allow that new external vertices (i.e. ones that are not present in G) emerge in G'. Therefore, whenever the degree of vertex $v \in Int_G$ becomes ≤ 1 in G', we shall augment G' by a protective loop edge around v. This augmentation will be understood automatically in all subgraphs of G. If G has a perfect internal matching, then subgraph G' is *nice* if G' also has a perfect internal matching in such a way that every perfect internal matching of G' can be extended to one of G. Finally, for a subgraph G' and matching M of G, $M_{(G')}$ will denote the restriction of M to G'.

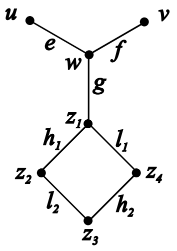


Fig. 12.1 Example graph having a perfect internal matching.

Let M be a perfect internal matching of G. An edge $e \in E(G)$ is said to be *M-positive* (*M-negative*) if $e \in M$ (respectively, $e \notin M$). An *M-alternating path (cycle)* in G is a non-empty path (respectively, even-length cycle) stepping on M-positive and M-negative edges in an alternating fashion. An *M-alternating loop* is an odd-length cycle having the same alternating pattern of edges, except that exactly one vertex has two negative edges incident with it. Let us agree that, if the matching M is understood or irrelevant in a particular context, then it will not be explicitly indicated in these terms.

An *external alternating path* is one that has an external endpoint. If both endpoints of the path are external, then it is called a *crossing*. An *alternating unit* is either a crossing or an alternating cycle. In Figure 12.1, $\gamma = (u, e, w, f, v)$ is an alternating crossing and $\beta = (z_1, l_1, z_4, h_2, z_3, l_2, z_2, h_1, z_1)$ is an alternating cycle with respect to the perfect internal matching $M = \{f, l_1, l_2\}$. An alternating path is *positive* if it is such at its internal endpoints, meaning that the edges incident with those endpoints are positive.

An internal vertex v of G is called *accessible* from external vertex w with respect to M (or simply v is M-accessible from w), if there exists a positive external M-alternating path connecting w and v. Furthermore, an alternating cycle is said to be *M-accessible* from w if at least one of its vertices is accessible from w in M. Generally it is not true that if a vertex is accessible from an external vertex w with respect to a given perfect internal matching, then it is accessible from w with respect to all perfect internal matchings. Nevertheless, as it was proved in [12], accessibility without specifying a concrete external vertex is matching invariant, that is, it holds for one particular perfect internal matching iff it holds for all perfect internal matchings. It is therefore meaningful to say that vertex v is accessible in G without specifying the perfect internal matching M and the external vertex w.

12.2.2 *Finite Automata*

A *non-deterministic finite automaton* is a triple $\mathcal{A} = (S, X, \delta)$, where S is a non-empty finite set, the set of *states*, X is an alphabet, the *input alphabet*, and $\delta : S \times X \to 2^S$ is the *transition function*. An automaton $\mathcal{A} = (S, X, \delta)$ is *deterministic* if for each $s \in S$ and $x \in X$, $|\delta(s, x)| \leq 1$. The function δ is extended to a mapping of $S \times X^*$ into 2^S by the definition $\delta(s, \varepsilon) = \{s\}$ – where ε denotes the empty word – and

$$\delta(s, wx) = \cup(\delta(t, x) \mid t \in \delta(s, w)) \quad \text{for } s \in S,\ w \in X^*, \text{ and } x \in X.$$

The automaton $\mathcal{A}' = (S', X, \delta')$ is a *subautomaton* of $\mathcal{A}$ if $S' \subseteq S$ and δ' is the restriction of δ to S'.

For any automaton $\mathcal{A} = (S, X, \delta)$ and $w \in X^*$, define the relation $\delta_w \subseteq S \times S$ by $(s, s') \in \delta_w$ iff $s' \in \delta(s, w)$. The relation δ_w is said to be *induced* by w. Then the set $T(\mathcal{A}) = \{\delta_w | w \in X^*\}$, together with the usual operation of composition between relations is a monoid $\mathcal{T}(\mathcal{A})$, called the *transition monoid* of $\mathcal{A}$.

Recall from [32] that a deterministic automaton $\mathcal{A} = (S, X, \delta)$ is a *permutation automaton* if $\delta(s, x) \neq \delta(s', x)$ for any $s, s' \in S$ with $s \neq s'$ and for any $x \in X$. The automaton $\mathcal{A}$ is *commutative* if $\delta(s, xy) = \delta(s, yx)$ for all $s \in S$ and $x, y \in X$. We shall also need the concept of full and semi-full automata. By a *full automaton* we mean an automaton $\mathcal{A} = (S, X, \delta)$ such that $X = \{x\}$ is a singleton and $\delta(s, x) = S$ for each $s \in S$. A *semi-full* automaton is the same as a full one, except that it has at least two states, and $\delta(s, x) = S \setminus \{s\}$ for each $s \in S$. A *trivial* automaton is a full one having a single state.

Let $\mathcal{A}_i = (S_i, X_i, \delta_i)$, $i = 1, 2$, be finite automata. A *homomorphism* of $\mathcal{A}_1$ into $\mathcal{A}_2$ is a pair $\psi = (\psi_S, \psi_X)$ of mappings $\psi_S : S_1 \to S_2$ and $\psi_X : X_1 \to X_2$ which satisfies the equation

$$\{\psi_S(s') \mid s' \in \delta_1(s, x)\} = \delta_2(\psi_S(s), \psi_X(x))$$

for every $s \in S_1$ and every $x \in X_1$. If there is a homomorphism $\psi = (\psi_S, \psi_X)$ such that both ψ_S and ψ_X are onto, then $\mathcal{A}_2$ is the *homomorphic image* of $\mathcal{A}_1$. The homomorphism ψ is an automaton *isomorphism* between $\mathcal{A}_1$ and $\mathcal{A}_2$ if ψ_S and ψ_X are bijections. In this case we also say that $\mathcal{A}_1$ and $\mathcal{A}_2$ are *isomorphic*. An isomorphism $\psi = (\psi_S, \psi_X)$ is called *strong* if $X_1 = X_2$, and ψ_X is the identity.

Now we turn to the definition of general products of automata [32]. For a nonnegative integer $k \in N$, let $[k]$ denote the set $\{1, \ldots, k\}$. Consider the automata $\mathcal{A} = (S, X, \delta)$, $\mathcal{A}_t = (S_t, X_t, \delta_t)$, $t \in [k], k \in N$, and let $\phi = (\phi_1, \ldots, \phi_k)$ be a sequence of mappings such that $\phi_t : S_1 \times \ldots \times S_k \times X \to X_t$ for each $t \in [k]$. Automaton $\mathcal{A}$ is the *general product* of $\mathcal{A}_1, \ldots, \mathcal{A}_k$ with respect to *feedback function* ϕ if the following conditions are satisfied:

(a) $S = S_1 \times \ldots \times S_k$;

(b) $\delta((s_1, \ldots, s_k), x) =$
$\delta_1(s_1, \phi_1(s_1, \ldots, s_k, x)) \times \ldots \times \delta_k(s_k, \phi_k(s_1, \ldots, s_k, x))$
for every $x \in X$, $s_j \in S_j$, $j \in [k]$.

The product is called an α_i*-product* ($i \in N$) if each ϕ_t, $t \in [k]$ is independent of its jth argument whenever $j \geq t + i$. Figure 12.2a) and b) show, respectively, the general scheme of α_1- and α_0-products of automata $\mathcal{A}_1$, $\mathcal{A}_2$, and $\mathcal{A}_3$. A *quasi-direct product* is a general product, where ϕ_t only depends on X for each $t \in [k]$.

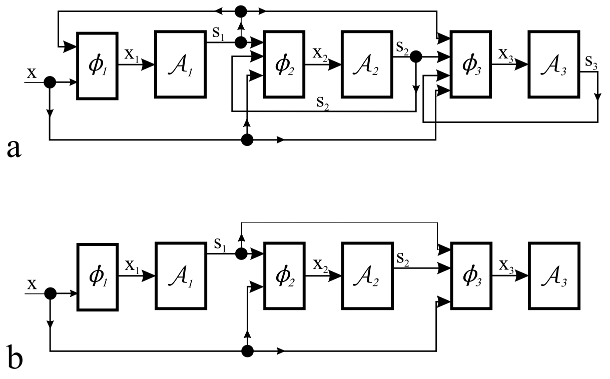


Fig. 12.2 General scheme of the α_i-product with $i \leq 1$.

The *general* ε*-product* of automata $\mathcal{A}_t$, $t \in [k]$ is defined analogously, with the exception that for every $t \in [k]$, $\phi_t(s_1, \ldots, s_k, x)$ is a non-empty subset of $\{y, \varepsilon\}$, where $y \in X_t$. The feedback function will then allow the automaton $\mathcal{A}_t$ to move either on input y, or on ε separately, or on both at the same time (nondeterministically) in state s_t, depending on the general input symbol x. (The standard definition $\delta_t(s_t, \varepsilon) = \{s_t\}$ applies for moving on ε.) We say that ϕ_t is *strict* if it is a function $S_1 \times \ldots \times S_k \times X \to X_t \cup \{\varepsilon\}$, and ϕ is strict if ϕ_t is such for every $t \in [k]$. A *disjoint* ϵ-product is a strict quasi-direct ϵ-product such that X is a superset of the disjoin union $X_1 + \ldots + X_k$ of the inputs X_t, and for every $x \in X$, $\phi_t(s_1, \ldots, s_k, x)$ is different from ε iff $x \in X_t$ – in which case x is mapped into itself by ϕ_t.

Let β be a type of automata products and $\mathcal{K}, \mathcal{F}$ be classes of automata. We say that $\mathcal{K}$ is *homomorphically complete* for $\mathcal{F}$ with respect to β-products if for every automaton $\mathcal{A}$ of $\mathcal{F}$ there exist automata $\mathcal{A}_j \in \mathcal{K}, j = 1, \ldots, n$ such that $\mathcal{A}$ is a homomorphic image of a subautomaton of a β-product of $\mathcal{A}_j, j = 1, \ldots, n$. In particular, the class $\mathcal{K}$ of automata (deterministic automata)is said to be homomorphically complete with respect to β-products if $\mathcal{F}$ above is the class of all automata (respectively, all deterministic automata). Finally, *isomorphic completeness* is defined analogously, replacing the word "homomorphic" by "isomorphic" in the definition above.

12.3 Soliton Graphs and Automata

In this section, following [24], we introduce soliton automata as a mathematical model for electronic switching at the molecular level. The underlying object of a soliton automaton is a graph representing the topological structure of a molecule. Within this structure there are chains of alternating single and double bonds, along which a form of energy, called the soliton, travels in small packets. According to this simple model, vertices correspond to atoms or certain groups of atoms, whereas edges represent chemical bonds or chains of bonds, with no multiplicity assigned to them at this level of "syntax". It is assumed that the molecule consists of carbon and hydrogen atoms only, and that among the neighbors of each carbon atom there exists a unique one to which the atom is connected by a double bond. This assumption is captured by imposing the structure of a perfect internal matching on the underlying graph as "semantics". Naturally enough, the edges contained in the given matching correspond to double bonds. External vertices and edges do not correspond to any particular atoms or chemical bonds, they just represent an interface at which soliton waves are induced and received.

Along these lines, a *soliton graph* is defined as an open graph having a perfect internal matching. Since the collection of perfect internal matchings will constitute the set of states of the corresponding soliton automaton, it is justified to call them *states* of the graph as well.

In more details, a soliton graph G models the intended molecular structure as follows. Each internal vertex v represents a C atom or a C-H couple, depending on whether $d(v)$ is 3 or 2, respectively. A positive (negative) edge (v, w) in a given state represents a double (single) bond between two carbon atoms, or, alternatively, a (CH)-chain with alternating double and single bonds that connects the C atoms at v and w, and which begins and ends with a double (repectively, single) bond. As the actual length of such chains does not affect the mathematical behavior of the model, we just take them as ordinary bonds (edges). (In some circumstances, however, it may be useful to distinguish between chains of different lengths.) Finally, external vertices represent the connection to surrounding molecule structures. Figure 12.3 shows an example soliton graph and its possible chemical interpretation.

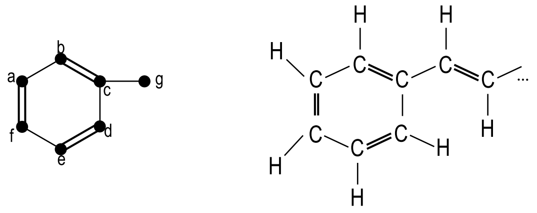


Fig. 12.3 A soliton graph with one of its interpretations.

Due to the chemical laws involved, the degree of each internal vertex in the above model is at most 3. This restriction was incorporated in the original definition of soliton automata by [24]. From the graph theory point of view, however, there is no need to impose this restriction in order for the model to be meaningful. Moreover, it can be proved (cf. [47]) that soliton automata defined on general graphs are equivalent to those comforming to the strict chemical rules.

For the study of the logical aspects of soliton switching we need to give a graph theoretic formalization of the state transitions induced by soliton waves. Ignoring the physico-chemical details, the effect of propagating a soliton along a chain of alternating single and double bonds is to exchange all these bonds. This phenomenon is captured directly by the concept of a soliton walk. Intuitively, a soliton walk is a backtrack-free walk which starts and ends at an external vertex, and alternates on positive and negative edges with respect to the current state. The sign of every edge changes immediately after traversing it, which makes the process interactive and recursive.

Before presenting a precise mathematical definition, we describe the process of switching along a soliton walk through the example in Figure 12.4. This process is also referred to as *making* the soliton walk. Let the current state of the automaton be the matching $M = \{e_1, e_5\}$. Making the walk $\alpha = (v_1, e_1, v_4, e_4, v_5, e_5, v_6, e_6, v_4, e_4, e_2, v_2)$, imitating a possible soliton wave, results in the sequence of matchings shown. In each step, the current position of the soliton is indicated by an arrow. Notice that, even though making one step with the soliton does not necessarily result in a perfect internal matching, by the time the walk is finished, a new state M' of G will have been reached.

The walk starts at vertex v_1, and after traversing edge e_1, the double bond of e_1 is changed to a single one. During the walk, if the soliton is

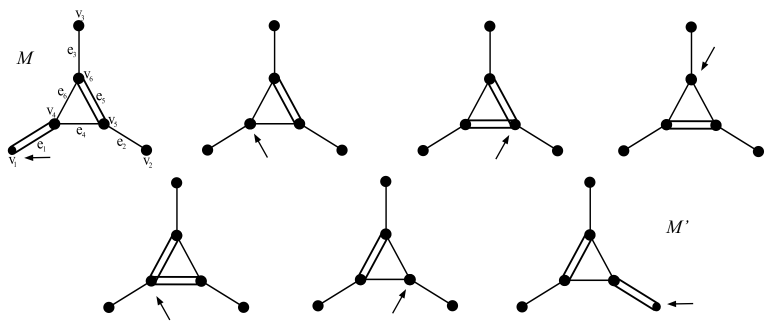


Fig. 12.4 A soliton walk.

about to continue its way on a negative edge (single bond), like in the second step at vertex v_4, then it might have several directions to choose from (e.g. both e_4 and e_6 could be chosen). On the other hand, if there are two positive edges incident with its current position, like in the third step at vertex v_5, then the walk must continue on the one that has not just been passed. In other words, the soliton is not allowed to backtrack.

The formal concept of soliton walks reflects the above heuristics. The collection of *external alternating walks* in G with respect to some state M, and the concept of *switching on* such walks are defined recursively in parallel as follows.

(i) The walk $\alpha = (v_0, v_0)$ is a soliton walk for each isolated external vertex v_0, and switching on α (i.e. making α) in state M results in the matching $S(M, \alpha) = M$.

(ii) The walk $\alpha = v_0 e v_1$, where $e = (v_0, v_1)$ with v_0 being external, is an external M-alternating walk, and $S(M, \alpha) = M\Delta\{e\}$. (The operation Δ is symmetric difference of sets.) Notice that $S(M, \alpha)$ need not be a matching.

(iii) If $\alpha = v_0 e_1 \dots e_n v_n$ is an external M-alternating walk ending at an internal vertex v_n, and $e_{n+1} = (v_n, v_{n+1})$ is such that $e_{n+1} \in S(M, \alpha)$ iff $e_n \in S(M, \alpha)$, then $\alpha' = \alpha e_{n+1} v_{n+1}$ is an external M-alternating walk and

$$S(M, \alpha') = S(M, \alpha)\Delta\{e_{n+1}\}.$$

It is required, however, that $e_{n+1} \neq e_n$, unless $e_n \in S(M, \alpha)$ is a loop.

It is clear by the above definition that $S(M,\alpha)$ is a perfect internal matching iff the endpoint v_n of α is external, too. In this case we say that α is a *soliton walk.*

As an alternative example, consider the graph G of Figure 12.1, and let $M=\{e,h_1,h_2\}$. Then $\gamma=(u,e,w,g,z_1,h_1,z_2,l_2,z_3,h_2,z_4,l_1,z_1,g,w,f,v)$ is a possible soliton walk from u to v with respect to M. Switching on γ then results in $S(M,\gamma)=\{f,l_1,l_2\}$.

Every soliton graph G gives rise to a *soliton automaton* $\mathcal{A}_G=(S_G,X\times X,\delta)$, where the set S_G of states consists of all perfect internal matchings of G. The input alphabet $X\times X$ for $\mathcal{A}_G$ is the set of all (ordered) pairs of external vertices in G, i.e., $X=Ext_G$, and the transition function δ is defined by

$$\delta(M,(v,w))=\{S(M,\alpha)|\alpha \text{ is an } M\text{-alternating soliton walk from } v \text{ to } w\}.$$

Augmenting the above general rule, if no soliton walk exists from v to w in M, then by definition $\delta(M,(v,w))=\{M\}$. This transition function is then extended for words of input $y\in(X\times X)^*$ as defined in Section 12.2.2.

Example. Consider the graph G in Figure12.1. This graph is a soliton graph having states: $s_h^e=\{e,h_1,h_2\}$, $s_l^e=\{e,l_1,l_2\}$, $s_h^f=\{f,h_1,h_2\}$, and $s_l^f=\{f,l_1,l_2\}$. The transitions of $\mathcal{A}_G$ are the following:

$$\begin{aligned}
&\delta(s_h^e,(u,v))=\delta(s_l^e,(u,v))=\{s_h^f,s_l^f\},\\
&\delta(s_h^f,(v,u))=\delta(s_l^f,(v,u))=\{s_h^e,s_l^e\},\\
&\delta(s_h^e,(v,u))=\{s_h^f\},\quad \delta(s_l^e,(v,u))=\{s_l^f\},\\
&\delta(s_h^f,(u,v))=\{s_h^e\},\quad \delta(s_l^f,(u,v))=\{s_l^e\},\\
&\delta(s_h^e,(u,u))=\{s_l^e\},\quad \delta(s_l^e,(u,u))=\{s_h^e\},\\
&\delta(s_h^f,(v,v))=\{s_l^f\},\quad \delta(s_l^f,(v,v))=\{s_h^f\}.
\end{aligned}$$

For example, the transition $s_h^e\to s_l^f$ on input (u,v) is induced by the soliton walk $(u,e,w,g,z_1,h_1,z_2,l_2,z_3,h_2,z_4,l_1,z_1,g,w,f,v)$.

Graph G is called *deterministic* if $\mathcal{A}_G$ is such in the usual sense. If, for every state M and input (v_1,v_2), there exists at most one soliton walk from v_1 to v_2 with respect to M, then G and $\mathcal{A}_G$ are called *strongly deterministic.*

According to the definition in [24], an edge e of a soliton graph G is *impervious* if there is no external alternating walk passing through e in any state of G. We extend this definition word by word for vertices of G. Edge e (vertex v) is then called *viable* if it is not impervious. It is clear that impervious vertices and edges have no impact on the operation of soliton automata. Thus, without loss of generality, we can restrict our attention (regarding soliton automata) to soliton graphs without impervious vertices and edges. From the point of view of graphs, however, it is important when two viable vertices can be connected by an impervious edge. Therefore our objective is in fact to study soliton graphs with viable vertices only. Such graphs will be called *viable soliton graphs.*

12.4 Elementary Decomposition of Soliton Graphs and Automata

In this section we review the main results obtained in [12] on the structure of soliton graphs. Further results on perfect internal matchings can be found in the series of papers [5–7, 10–13, 15, 16].

We begin with a short summary of the matching theoretic concepts involved. The reader can obtain a good understanding of these concepts by following the definitions to come on Figure 12.5 below.

Again, let us fix a soliton graph G for the whole section. According to [7] and [55] G is *elementary* if its allowed edges form a connected subgraph covering all of the external vertices. Note that an elementary graph may contain forbidden edges. If it does not, then it is called 1-*extendable.* A *trivial* elementary soliton graph is either a single external vertex or a single mandatory external edge with a loop around its internal endpoint. Clearly, the automata defined by a trivial elementary soliton graphs are trivial themselves.

In general, the allowed edges of any soliton graph G will span a number of connected components as subgraphs in G. The full subgraphs of G induced by the vertices of these components are called the *elementary components* of G. An elementary component is called *external* if it contains external vertices, and *internal* if this is not the case. A *mandatory elementary component* is a single mandatory edge $e \in E(G)$, which might have a loop around one or both of its endpoints.

On the analogy of the original concept in [55], *canonical equivalence* of elementary graphs was generalized for soliton graphs in [12]. The definition of canonical equivalence is as follows. Let $v, w \in Int_G$ be internal vertices. Then $u \sim v$ if u and v belong to the same elementary component and an extra edge e connecting u and v becomes forbidden in $G{+}e$. It can be proved that $\sim$ is indeed an equivalence relation. The classes determined by $\sim$ are called *canonical classes,* which constitute the *canonical partition* of Int_G. By definition, the restriction of $\sim$ to any particular elementary component C determines the canonical partition of the elementary graph C_h, which is C augmented by the so called hidden edges. A *hidden edge* $e = (u, v)$ of C, usually not originally present in G, emerges from an alternating ear (see [55] or below) connecting u and v outside C. For more details, see [12]. With a slight ambiguity, when referring to an elementary component C of G, we shall in fact mean the elementary graph C_h.

The structure of elementary components in a soliton graph G has been analysed in [12]. To summarize the main results of this analysis, we first need to review some of the key concepts introduced in that paper. An internal elementary component C is *one-way* if all external alternating paths (with respect to any perfect internal matching M) enter C in vertices belonging to the same canonical class of C. This unique class, as well as the vertices belonging to it, are called *principal*. Furthermore, every external elementary component is considered a priori one-way (with no principal canonical class, of course). An elementary component is *two-way* if it is not one-way.

For example, the graph of Figure 12.5 has five elementary components, among which D and E are mandatory external, while C_1, C_2 and C_3 are internal. Component C_3 is one-way with the canonical class $\{u, v\}$ being principal, while C_1 and C_2 are two-way.

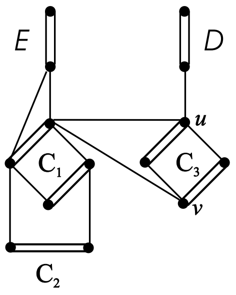


Fig. 12.5 Elementary components in a soliton graph.

Let C be an elementary component of G, and M be a state. An *M-alternating C-ear* is a negative M-alternating path or loop having its two endpoints, but no other vertices, in C. The endpoints of the ear will necessarily belong to the same canonical class of C.

We say that elementary component C' is *two-way accessible* from component C with respect to any perfect internal matching M, in notation $C\rho C'$, if C' is covered by an M-alternating C-ear. It was shown in [12] that the two-way accessible relationship is matching invariant. A *family* of elementary components in G is a block of the partition induced by the smallest equivalence relation containing ρ. A family $\mathcal{F}$ is called *external* if it contains an external elementary component, otherwise $\mathcal{F}$ is *internal.* It was proved in [12] that every family contains a unique one-way elementary component, called the *root* of the family. With a slight ambiguity, we shall identify family $\mathcal{F}$ with the subgraph of G spanned by its members.

Our example graph in Figure 12.5 has three families: $\mathcal{F}_1 = \{E, C_1, C_2\}$, $\mathcal{F}_2 = \{D\}, \mathcal{F}_3 = \{C_3\}$. Families $\mathcal{F}_1$ and $\mathcal{F}_2$ are external, whereas $\mathcal{F}_3$ is internal. The roots of the families are E, D and C_3.

For two distinct families $\mathcal{F}_1$ and $\mathcal{F}_2$, $\mathcal{F}_2$ is said to *follow* $\mathcal{F}_1$, in notation $\mathcal{F}_1 \mapsto \mathcal{F}_2$, if there exists an edge in G connecting any non-principal vertex in $\mathcal{F}_1$ with a principal vertex belonging to the root of $\mathcal{F}_2$. The reflexive and transitive closure of $\mapsto$ is denoted by $\overset{*}{\mapsto}$. One of the main results in [12] is the observation that the relation $\overset{*}{\mapsto}$ is a partial order among the families, by which the external families are minimal elements.

The theorem below is a short summary of results on the elementary structure of viable soliton graphs.

Theorem 12.1. *The following conditions hold for the families of G.*

(i) *In each internal family $\mathcal{F}$, the root of $\mathcal{F}$ has a unique canonical class P, called principal, such that every external alternating path leading to any vertex in $\mathcal{F}$ enters the family at a vertex belonging to P.*

(ii) *There exists a partial order $\overset{*}{\mapsto}$ among the families, which reflects the order in which external alternating paths reach the families. The minimal elements according to $\overset{*}{\mapsto}$ are the external families.*

(iii) *In each family $\mathcal{F}$, every viable forbidden edge is part of an alternating C-ear (in a matching-invariant way) for some $C \in \mathcal{F}$. For the root R of $\mathcal{F}$, the connected components of the subgraph $\mathcal{F} \setminus R$ are adjacent to vertices in R belonging to separate non-principal canonical classes.*

(iv) *An edge e of G is impervious iff e connects two principal vertices belonging to either the same family or two different families.*

For convenience, the inverse of the partial order $\overset{*}{\mapsto}$ between G's families will be denoted by $\leq_G$, or simply $\leq$ if G is understood. Referring to the Hasse diagram of $\leq$, we say that family $\mathcal{G}$ is *below* family $\mathcal{F}$ if $\mathcal{G} \leq \mathcal{F}$, that is, $\mathcal{G}$ follows $\mathcal{F}$. The relation $\leq_G$ suggests a top-down design of the families with the external ones being on top.

Using the elementary structure of soliton graphs described above, the elementary decomposition of soliton automata was worked out in [9] and [50]. As a result of this decomposition, the descriptional complexity of soliton automata can be significantly reduced. The applied technique also provides a basis for the characterization of special classes of automata. These results will be presented in the next section.

12.5 Characterizing Soliton Automata

Determining the computational power of soliton automata is the main objective of their study. The way to achieve this goal is to characterize soliton automata both structurally, using different kinds of products, and by their transition monoids. Since the behavior of these automata is based entirely on the underlying graph structure, their descriptional complexity, too, can be analyzed through this structure.

The characterization of general soliton automata is a very complex tasks, and requires some important special classes to be dealt with first. In this section, chronologically, we review the results by Dassow and Jürgensen (cf. [24–27]) on the transition monoids of certain simple classes of soliton automata first, then we present our own results on deterministic soliton automata, and on the structural decomposition of general and constant soliton automata.

12.5.1 *Special Cases of Deterministic Soliton Automata*

In their introductory paper [24], Dassow and Jürgensen gave a characterization of strongly deterministic soliton graphs in terms of so called chestnut graphs and trees.

Definition 12.2. *A connected graph G is called a* chestnut *if it has a representation in the form $G = \beta + \alpha_1 + \ldots + \alpha_k$ with $k \geq 1$, where β is a cycle of even length and each α_i ($i \in [k]$) is a tree subject to the following conditions:*

(i) $V(\alpha_i) \cap V(\alpha_j) = \emptyset$ for $1 \leq i \neq j \leq k$;

(ii) $V(\alpha_i) \cap V(\beta)$ consists of a unique vertex – denoted by v_i – for each $i \in [k]$;

(iii) v_i and v_j are at even distance on β for any distinct $i, j \in [k]$;

(iv) any vertex $w_i \in V(\alpha_i)$ with $d(w_i) > 2$ is at even distance from v_i in α_i for each $i \in [k]$.

Figure 12.6 shows an example of a chestnut.

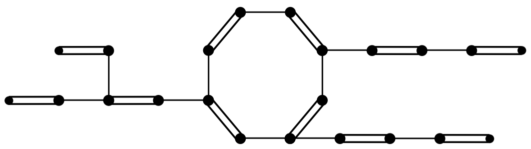


Fig. 12.6 A chestnut.

Theorem 12.3. *Let G be a connected soliton graph. Then G is strongly deterministic if and only if G is a chestnut or a tree.*

The transition monoids of strongly deterministic soliton automata were also characterized in [24] as follows. The reader is referred to [64] for the group theory concepts involved. Some of these concepts are explained below.

Let $\mathcal{G}$ be a permutation group on a set Ω. A subset Ψ of Ω is called a *block* if for each $g \in \mathcal{G}$ the image $g(\Psi)$ either coincides with Ψ or is disjoint from Ψ. The sets $\emptyset$, $\{\omega\}$, for any $\omega \in \Omega$, and Ω are the *trivial blocks.* The group $\mathcal{G}$ is called *primitive* if it is transitive and has only trivial blocks.

Theorem 12.4. *The transition monoid of a strongly deterministic soliton automaton is a direct product of primitive permutation groups which are generated by involutorial elements.*

As a refinement of the above theorem, Dassow and Jürgensen described in [27] the primitive permutation groups which occur as transition monoids of automata associated with a special class of trees.

Theorem 12.5. *Let T be a soliton tree such that any two vertices of degree at least 3 are at even distance from each other. Then the transition monoid of $\mathcal{A}_T$ is a symmetric group.*

Recently, as a further refinement of the above result, in [42] and [45] new bounds on the number of states of a tree-based soliton automaton have been established and a sufficient condition was proved for when the transition monoid of such a soliton automaton consists only of even permutations of the set of states.

The characterization of deterministic soliton automata is far more difficult than that of the strongly deterministic ones. Dassow and Jürgensen have only analyzed two special cases with respect to their transition monoids: deterministic soliton automata with a single external vertex [25] and deterministic soliton automata with at most one cycle [26]. The main result of [25] is stated in Theorem 12.6 below. In this theorem, we rely on the concept of *usable cycle*, which is simply a cycle occuring as a subwalk of some soliton walk in an arbitrary state.

Theorem 12.6. *Let G be a deterministic, connected soliton graph with a single external vertex. If G contains a usable cycle of even length, then G is a chestnut, $\mathcal{A}_G$ has 2 states and its transition monoid $T(\mathcal{A}_G)$ is isomorphic to the symmetric group of order 2. Otherwise, $\mathcal{A}_G$ has a single state only and $T(\mathcal{A}_G)$ is trivial.*

The results of [26] are summarized as follows. Let G be a connected open graph with a single cycle β. Then G has a representation in the form $G = \beta + \alpha_1 + \ldots + \alpha_r$, $r \in N$, such that $\alpha_1, \ldots, \alpha_r$ are pairwise vertex-disjoint trees and for each $i \in [r]$, $V(\alpha_i) \cap V(\beta)$ consists of a single vertex. This decomposition will be referred to as the *tree-decomposition* of G.

Theorem 12.7. *Let G be a connected soliton graph with a single cycle β, having a tree-decomposition $G = \beta + \alpha_1 + \ldots + \alpha_r$. Moreover, let $V_\alpha = \{v_1, \ldots, v_r\}$, where for each $i \in [r]$, v_i denotes the unique common vertex of α_i and of β. Then $T(\mathcal{A}_G)$ is a primitive group of permutations if and only if one of the following conditions fails to hold:*

(a) β is an odd-length cycle.

(b) There are three distinct vertices $v_{i_1}, v_{i_2}, v_{i_3}$ of V_α such that each α_{i_j}, $j = 1, 2, 3$ consists of a single path, and for $s = 1, 2$ the unique odd-length subpath of β connecting v_{i_s} and $v_{i_{s+1}}$ – apart from its endpoints – does not contain vertices of V_α.

Finally, as a new approach, in recent works (cf. [19] and [20]), soliton automata were analyzed as language acceptors. In these studies, the closure properties of the class of all languages accepted by soliton automata have been investigated. It was proved that the class of languages accepted by soliton automata is nearly an *anti-AFL*, that is, not closed under most of the usual operations on languages.

12.5.2 *Deterministic Soliton Automata*

In the analysis of complex systems [23] it is crucial to find those characteristics which make a given system deterministic. Hence the characterization of deterministic soliton automata is a fundamental problem both in theory and practice. In this section we provide a purely syntactical, that is, state-(matching) independent characterization of deterministic soliton graphs. Our first observation concerns the existence (rather non-existence) of alternating cycles in deterministic soliton graphs.

Theorem 12.8. ([14], [47]) *A viable soliton graph G is deterministic iff each connected component G_i of G is either a chestnut or does not contain an alternating cycle with respect to any state of G.*

By Theorem 12.8 it is clear that, with the exception of chestnuts, connected deterministic viable soliton graphs have mandatory internal elementary components only. Therefore – by the very definition of soliton automata – every deterministic soliton automaton, the underlying graph of which does not contain chestnut components, is strongly isomorphic to one having external elementary components only. Indeed, the deletion of internal elementary components, all of them being mandatory, does not affect the transition function of the automaton, not even the self transitions, as they are present anyhow by the definition of δ, provided that the automaton is deterministic. As a consequence, we obtain the following result.

Theorem 12.9. *Every deterministic soliton automaton is strongly isomorphic to a disjoint ϵ-product of chestnut automata and elementary soliton automata having no alternating cycles in their underlying graph.*

The key to a matching-independent characterization of deterministic soliton graphs is a reduction procedure worked out in [8, 14, 46]. We are going to elaborate on this procedure below.

A *redex* r in graph G consists of two adjacent edges $e = (u, z)$ and $f = (z, v)$ such that $u \neq v$ are both internal, and the degree of z is 2. The vertex z is called the *center* of r, while u and v (e and f) are the two *focal* vertices (respectively, *focal* edges) of r.

Let r be a redex in G. *Shrinking* r in G means creating a new graph G_r from G by deleting the center of r and merging the two focal vertices of r into one vertex s (see an example in Figure 12.7). Now suppose that G is a soliton graph. For a state M of G, let M_r denote the restriction of M to edges in G_r. Clearly, M_r is a state of G_r. Notice that the state M can be reconstructed from M_r in a unique way. In other words, the connection $M \mapsto M_r$ is a one-to-one correspondence between the states of G and those of G_r. Then it is easy to prove that the soliton automata $\mathcal{A}_G$ and $\mathcal{A}_{G_r}$ are strongly isomorphic.

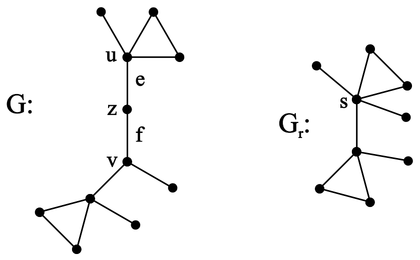


Fig. 12.7 Contracting a redex in graph G.

Another natural simplifying operation on graphs is the removal of a loop from around a vertex v if, after the removal, v still remains internal. Such loops will be called *secondary.* Let G_v denote the graph obtained from G by removing a secondary loop e at vertex v. Clearly, if G is a soliton graph, then so is G_v, and the states of G_v are exactly the same as those of G. The automata $\mathcal{A}_G$ and $\mathcal{A}_{G_v}$, however, need not be isomorphic. This is due to the fact that any external alternating walk reaching v on a positive edge can turn back in G after having made the loop e twice, while the turnaround may not be possible for the same walk in G_v without the presence of e. Nevertheless, it is still true that for every deterministic elementary soliton graph G, $\mathcal{A}_G$ and $\mathcal{A}_{G_v}$ are strongly isomorphic.

For an arbitrary graph G, shrink all redexes and remove all secondary loops in an iterative manner to obtain a reduced graph, that is, a graph free from redexes and secondary loops. Denote the resulting graph by $r(G)$. Observe that this reduction procedure has the so called Church-Rosser property (cf. [4]), that is, if G admits two different one-step reductions to graphs G_1 and G_2, then either G_1 is isomorphic to G_2, or G_1 and G_2 can further be reduced to a common graph $G_{1,2}$. In this context, one reduction step means shrinking a redex or removing a single secondary loop. As an immediate consequence of the Church-Rosser property, the graph $r(G)$ above is unique up to graph isomorphism.

Let G be a connected graph not containing even-length cycles. Clearly, when shrinking each odd-length cycle to one vertex, G turns into a tree.

For this reason we call G a *generalized tree*. It is easy to see that every generalized tree is a viable deterministic soliton graph. The matching-independent characterization of non-chestnut deterministic soliton graphs is now stated as follows.

Theorem 12.10. *For any graph G, if $r(G)$ is a generalized tree, then G is a deterministic soliton graph. Conversely, if G is a deterministic viable elementary soliton graph, then $r(G)$ is a generalized tree.*

Observe that the condition $r(G)$ being a generalized tree alone does not necessarily imply that G is viable or elementary, because reduction might remove some mandatory elementary components (or even impervious parts) from G. Still, G and $r(G)$ are strongly isomorphic as seen before.

Let us call a chestnut graph G a *baby chestnut* if it consists of a pair of parallel edges connecting two vertices (v_1, v_2), and a number of edges or 2-length paths originating from the principal vertex v_1 and leading to different external vertices. See Figure 12.8 for an example. A baby chestnut is *regular* if it does not contain any 2-length paths according to the above description. It is easy to see that every chestnut graph reduces to a baby chestnut. Moreover, the reduction involved must not eliminate any secondary loops, since chestnuts do not contain impervious edges. As a further simplification, each 2-length path from any external vertex can be trimmed down to length one without effecting isomorphism of automata. Thus, we have the following matching-invariant strengthening of Theorem 12.9.

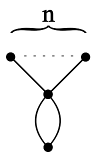


Fig. 12.8 A regular baby chestnut with n external vertices.

Theorem 12.11. *Every deterministic soliton automaton is strongly isomorphic to a disjoint ϵ-product of regular baby chestnut automata and automata defined by generalized trees.*

12.5.3 *Constant Soliton Automata*

By definition, if an external edge e of a soliton graph G is constant, then its external endpoint is contained in a trivial external elementary component. This component is the external endpoint itself if e is negative, and the mandatory edge e if it is positive. Even though the class of soliton automata with only constant external edges is worth studying for itself, the main reason for considering this special class separately is that such constant automata play a central role in the decomposition of general soliton automata. We start with the simplest case, when the underlying graph has a single external vertex. The key issue is the characterization of self-transitions via soliton trails. Recall from [9] that a *self-transition* of any automaton in state s is one on a concrete input symbol (i.e. not on ε) that leaves the automaton in state s.

A *soliton trail* α is an external alternating walk, stepping on positive and negative edges in such a way that α is either a path, or it returns to itself only in the last step, traversing a negative edge. The trail α is a *c-trail* (*l-trail*) if it does return to itself, closing up an even-length (respectively, odd-length) cycle. That is, $\alpha = \alpha_1 + \alpha_2$, where α_1 is a path and α_2 is a cycle. These two components of α are called the *handle* and *cycle*, in notation, α_h and α_c. See Figure 12.9 for an illustration of the above concepts. In this figure, as well as in most of the further ones throughout the paper, double lines indicate edges that belong to a given matching.

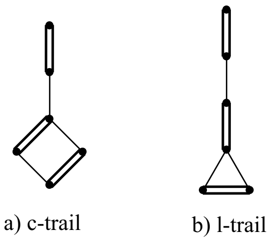


Fig. 12.9 Soliton trails.

An *M-alternating double soliton c-trail* α from external vertex v is a pair of distinct M-alternating soliton c-trails $\alpha = (\alpha^1, \alpha^2)$ from v such that $E(\alpha^1_h) \cap E(\alpha^2_c) = \emptyset$, $E(\alpha^2_h) \cap E(\alpha^1_c) = \emptyset$, and either $\alpha^1_c = \alpha^2_c$ or $V(\alpha^1_c) \cap V(\alpha^2_c) = \emptyset$. Figure 12.10 shows two simple examples of double soliton c-trails. Notice that, in the case $\alpha^1_c = \alpha^2_c$, the difference between the handles $E(\alpha_h^1)$ and $E(\alpha_h^2)$ can be as minor as using two distinct parallel negative edges connecting two given vertices along the handle. In particular, for chestnut graphs this means that the introduction of just one forbidden parallel edge in the graph creates a double soliton c-trail and makes the corresponding automaton nondeterministic for reasons explained below.

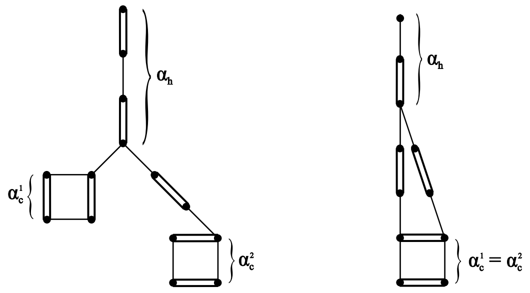


Fig. 12.10 Example for double soliton c-trails.

Theorem 12.12. [9] *For any state M of a soliton automaton $\mathcal{A}_G = ((S_G, (X \times X), \delta)$ and for any external vertex $v \in X$ of G, $M \in \delta(M, (v, v))$ iff one of the following conditions holds:*

(i) G does not contain an M-alternating soliton c-trail from v.

(ii) G contains an M-alternating soliton l-trail from v.

(iii) G contains an M-alternating double soliton c-trail from v.

It was proved in [51] that the property of having either a double soliton c-trail or a soliton l-trail is matching invariant in every soliton graph with a single external vertex. (The trail in hand, however, can be l in one state and double c in another.) Using this observation it is easy to derive the following result from Theorem 12.12.

Theorem 12.13. [51] *Let G be a soliton graph with a single external vertex v. Then $\mathcal{A}_G$ is either a full or a semi-full automaton. Moreover, $\mathcal{A}_G$ is semi-full iff G is a bipartite graph without double soliton c-trails.*

Having characterized the structure of soliton graphs and automata with a single external vertex, their transition monoids can be described in a simple way. Since the deterministic case has already been dealt with in Theorem 12.6, we can assume that our automata are nondeterministic. In order to state our result on the transition monoids of such automata, we need a few simple concepts from the theory of semigroups.

A monoid $(A, \circ)$ with $|A| \geq 2$ is called a *null monoid* if there exists an element $a \in A$ such that $x \circ y = a$ for all $x, y \in A \setminus \{e\}$, where e denotes the identity element. In that case a is called the *zero element.* The null monoid consisting of exactly two elements – i.e. the identity and the zero element – is called the *trivial null monoid.*

Theorem 12.14. *Let G be a soliton graph with a single external vertex v such that $\mathcal{A}_G$ is not deterministic. Then $\mathcal{T}(\mathcal{A}_G)$ is a null monoid with at most 3 elements. Moreover, $\mathcal{T}(\mathcal{A}_G)$ is nontrivial iff G is a bipartite graph without double soliton c-trails.*

Now we turn to the structural characterization of constant soliton automata in general. An *ordered system of elementary soliton automata* (OSA, for short) is a finite *set* of automata $\mathcal{S} = \{\mathcal{A}_t = (S_t, X_t, \delta_t) | t \in [m]\}$, $m \in N$, arranged in a partial order $\leq$, so that the following conditions are met.

1. Every automaton in $\mathcal{S}$ is either full or semi-full, or it is an elementary soliton automaton. Furthermore, an automaton appears as a maximal element in the Hasse diagram of $\leq$ iff it is an elementary soliton automaton. Also, the input alphabets of the soliton automata in $\mathcal{S}$ are pairwise disjoint.

2. If $\mathcal{A}_t$ is semi-full for some $t \in [m]$, then the principal ideal $\{\mathcal{A}'_t \in \mathcal{S} | \mathcal{A}'_t \leq \mathcal{A}_t\}$ determined by $\mathcal{A}_t$ is a chain of semi-full automata.

The OSA $\mathcal{S}$ is called *constant* if every elementary soliton automaton in $\mathcal{S}$ is trivial.

Let $(\mathcal{S}, \leq)$ be an OSA as specified above. An *ordered ϵ-product* of $\mathcal{S}$ is an α_0-ϵ-product $\mathcal{A} = (S, X, \delta)$ of the automata $\mathcal{A}_1, \ldots, \mathcal{A}_m$ equipped with a feedback function $\phi = (\phi_1, \ldots, \phi_m)$ satisfying the four conditions below.

1. The given sequence of the automata $\mathcal{A}_t$ is an extension of $\leq$ to a linear order.

2. For all states $s_j \in S_j$, $j \in [m]$ and $x \in X$, the set

$$I(s_1, \ldots, s_m) = \{t \in [m] | \phi_t(s_1, \ldots, s_m, x) \neq \{\varepsilon\}\}$$

is either empty, or it is a nonempty subset of the (maximal) principal ideal of $\leq$ determined by a soliton automaton $\mathcal{A}_i$. In the latter case:

(i) $i \in I(s_1, \ldots, s_m)$, and

(ii) whenever $t \in I(s_1, \ldots, s_m)$ such that $\mathcal{A}_t$ is full or semi-full, $t' \in I(s_1, \ldots, s_m)$ for all t' such that $\mathcal{A}_{t'} \leq \mathcal{A}_t$.

Moreover, $\phi_t(s_1, \ldots, s_m, x)$ depends only on x and s_i.

3. For every $t \in [m]$, if $\mathcal{A}_t$ is full or semi-full, then ϕ_t is strict.

4. $X = (\cup_i Y_{t_i})^2$, where $\mathcal{A}_{t_1}, \ldots, \mathcal{A}_{t_k}$ is the sequence of elementary soliton automata in $\mathcal{S}$ with $X_{t_i} = Y_{t_i} \times Y_{t_i}$, and, furthermore, $\phi_{t_i}(s_1, \ldots, s_m, x) \neq \{\varepsilon\}$ iff $x \in X_{t_i}$.

Notice that condition 3 above applies for all automata in $\mathcal{S}$, that is, also for the elementary soliton automata in case they are full or semi-full. This can only happen, however, if they are trivial, hence full. Also, if $\mathcal{A}_i$ is trivial in 2 above, then $I(s_1, \ldots, s_m)$ is the whole principal ideal determined by $\mathcal{A}_i$. Consequently, if $\mathcal{S}$ is constant, then the resulting α_0-product is a strict quasi-direct ϵ-product.

Combining ordered products of elementary soliton automata with the elementary decomposition of soliton graphs given in Section 12.4, we obtain the following characterization of constant soliton automata.

Theorem 12.15. [53] *The class of constant soliton automata and the class of automata obtained as ordered strict quasi-direct ϵ-products of constant OSA's coincide up to strong isomorphism.*

12.5.4 *General Soliton Automata*

On the basis of ordered ϵ-products defined in the previous section, we are now ready to present the structural characterization of general soliton automata.

Let $\mathcal{A}_G = (S_G, X \times X, \delta)$ be an arbitrary soliton automaton, and consider the partial order $\leq_G$ of its families $\mathcal{F}_1, \ldots, \mathcal{F}_m$, $m \in N$. We are going to construct an OSA $\mathcal{S}_G$ from these families in the following way. Represent each internal family $\mathcal{F}_i$ in $\mathcal{S}_G$ by a full or semi-full automaton $\mathcal{A}_i$ over the set S_i of states, where S_i is the cartesian product of the sets of states (perfect matchings) of the elementary components in $\mathcal{F}_i$. Automaton $\mathcal{A}_i$ is chosen semi-full iff each family in the principal ideal determined by $\mathcal{F}_i$ consists of a single (necessarily one-way) elementary component free from c-trails and l-trails, and the ideal itself is a chain.

As to an external family $\mathcal{F}_j$, its unique external elementary component C_j contributes the corresponding elementary automaton $\mathcal{A}_j$ to $\mathcal{S}_G$. In case there are internal elementary components in $\mathcal{F}_j$, a number of full automata $\mathcal{A}_j^1, \ldots, \mathcal{A}_j^{n_j}$ are also added to $\mathcal{S}_G$, which correspond to groups of two-way internal elementary components in $\mathcal{F}_j$ originating from different canonical classes of the elementary graph C_j, taking the cartesian product of their sets of states as above. See Theorem 12.1 (iii). The automata $\mathcal{A}_j^i$ are inserted in the partial order $\leq_G$ – which is adopted for the OSA $\mathcal{S}_G$ being created

– in such a way that they come as descendants of $\mathcal{A}_j$ and are minimal in the extended Hasse diagram. Clearly, if at least one $\mathcal{A}_j^i$ is present, and state M and external vertex x in C_j are such that an internal component amalgamated in $\mathcal{A}_j^i$ is accessible from x in M, then every transition of $\mathcal{A}_j$ from x to each external vertex y in C_j may trigger an arbitrary transition of $\mathcal{A}_j^i$. By the same token, there will be a self-transition from x to x in any state of $\mathcal{A}_G$ extending M, even if no such a transition exists in $\mathcal{A}_j$ by itself in state M. If this is not the case, however, then transitions of $\mathcal{A}_j$ from x will stand alone, not affecting the states of $\mathcal{A}_j^i$. In summary, we have the following theorem.

Theorem 12.16. *The soliton automaton $\mathcal{A}_G$ is strongly isomorphic to an ordered ϵ-product of the OSA $\mathcal{S}_G$.*

Unlike for constant automata, however, we cannot say that an arbitrary ordered ϵ-product of any OSA $\mathcal{S}$ is isomorphic to (let alone strong isomorphism) a soliton automaton.

12.6 Complete Systems of Soliton Automata

Having completed the structural analysis of soliton automata in the previous section, it is natural to ask about the computational power of systems of such automata with respect to the general product. The first paper on this issue was [34], where a detailed analysis was given for homomorphically complete systems of strongly deterministic soliton automata. The results of this analysis are summarized in Section 12.6.1. These results were generalized in [48] by characterizing the isomorphically complete systems of soliton automata both in the deterministic and nondeterministic case, which cases will be covered in Sections 12.6.2 and 12.6.3, respectively.

12.6.1 *Homomorphic Representation of Deterministic Automata*

Recall from [55] that, for any $n \in N$, the *n-star* is the bipartite graph $K_{1,n}$ with bipartition (A_1, B_n) such that $|A_1| = 1$, $|B_n| = n$, and the unique vertex of A_1 is adjacent to every vertex of B_n. (See Figure 12.11.)

Theorem 12.17. *The class $\mathcal{G} = \{\mathcal{A}_{K_{1,n}}) \mid n \in N\}$ is homomorphically complete for the class of commutative permutation automata with respect to*

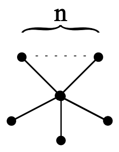


Fig. 12.11 n-star.

the α_0-product. Furthermore, $\mathcal{G}$ is homomorphically complete with respect to the α_1-product.

It is known from [28] that the α_2-product is homomorphically equivalent to the general product. Therefore, homomorphic representations of automata by α_i-products of strongly deterministic soliton automata with a concrete $i \geq 2$ are the same as those by general products of such automata. The final result of [34] describes the homomorphically complete classes of strongly deterministic soliton automata with respect to general products.

Theorem 12.18. *A class $\mathcal{K}$ of strongly deterministic soliton automata is homomorphically (isomorphically) complete with respect to the general product (or the α_i-product for any $i \geq 2$) if and only if $\mathcal{K}$ contains an automaton whose underlying soliton graph G satisfies one of the following three conditions:*

(i) G consists of at least two connected components;

(ii) G is a tree;

(iii) G is a chestnut with at least two external vertices.

12.6.2 *Isomorphic Representation of Nondeterministic Automata*

In this section we characterize the isomorphically complete systems of soliton automata.

Isomorphic representations of nondeterministic automata were first studied in [33], where necessary and sufficient conditions were given for classes of automata to be isomorphically complete with respect to the general product. This characterization was refined in [40] by showing that the α_1-product is isomorphically equivalent to the α_i-product for any $i \geq 2$, thus, to the general product. A further refinement regarding soliton automata is the following statement.

Theorem 12.19. ([48]) *A class $\mathcal{K}$ of nondeterministic soliton automata is isomorphically complete with respect to the α_0-product iff $\mathcal{K}$ is isomorphically complete with respect to the general product (or the α_i-product for any $i \geq 1$).*

Theorem 12.19 implies that it is appropriate to consider general products for the characterization of isomorphically complete systems of soliton automata with respect to any concrete α_i-product. That is, we can speak of isomorphically complete classes of soliton automata in general, without specifying any concrete α_i-product to rely on. The main result on isomorphically complete systems of nondeterministic automata in [48] describes the underlying graph structure of the automata in these systems.

Theorem 12.20. *A class $\mathcal{K}$ of nondeterministic soliton automata is isomorphically complete iff $\mathcal{K}$ contains (not necessarily distinct) automata $\mathcal{A}_{G_t} = (S_{G_t}, X_t \times X_t, \delta_t)$ $(t = 1, 2)$ such that for both $t = 1, 2$ there exists a state $M_t \in S_{G_t}$ with an M_t-alternating cycle γ_t and external vertices $v_t^1 \neq v_t^2 \in Ext_{G_t}$ which satisfy the following conditions:*

(i) *γ_t is M_t-accessible from v_t^1,*

(ii) *there exists either an M_t-alternating soliton l-trail from v_t^1 or an M_t-alternating double soliton c-trail from v_t^1,*

(iii) *γ_1 is M_1-accessible from v_1^2, and there does not exist either an M_1-alternating soliton l-trail from v_1^2 or an M_1-alternating double soliton c-trail from v_1^2,*

(iv) *either there exists an external vertex of G_2 which is not connected with v_2^2 by an M_2-alternating crossing, or one of conditions (i) – (iii) of Theorem 12.12 holds for M_2 and v_2^2 such that γ_2 is not M_2-accessible from v_2^2.*

By the above theorem it is easy to construct an isomorphically complete system of two soliton automata. The graphs G_1 and G_2 are, however, not necessarily distinct in Theorem 12.20, which provides a motivation for finding singleton classes.

Corollary 12.21. *Let $\mathcal{A}_G = (S_G, X \times X, \delta)$ be a soliton automaton, M be a state of $\mathcal{A}_G$, γ be an M-alternating cycle, and $v, w \in X$ be distinct external vertices with the following conditions:*

(i) *γ is M-accessible from v and w,*

(ii) *there does not exist an M-alternating crossing between v and w,*

(iii) there exists either an M-alternating soliton l-trail from v or an M-alternating double soliton c-trail from v.

Then $\{\mathcal{A}_G\}$ is isomorphically complete with respect to the α_0-product.

A simple example graph G that satisfies the conditions of Corollary 12.21 is given in Figure 12.12. Thus, we have the following corollary.

Corollary 12.22. *The class $\{\mathcal{A}_G\}$ is isomorphically complete with respect to the α_0-product.*

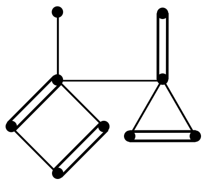


Fig. 12.12 Example soliton graph determining an isomorphically complete system.

12.6.3 *Isomorphic Representation of Deterministic Automata*

In the product hierarchy, it is natural to start studying products of deterministic soliton automata with respect to the α_0-products. It turns out, however, that – similarly to the strongly deterministic case dealt with in [34] – even homomorphic representations by the α_0-product are not powerful enough for any class of deterministic soliton automata to be complete. This follows from the fact that every deterministic soliton automaton is a permutation automaton, considering that subautomata and homomorphic images of permutation automata are also permutation automata, and α_0-products preserve the "permutation property" of automata (cf. [32]).

Theorem 12.23. *There is no class of deterministic soliton automata which is homomorphically complete with respect to the α_0-product.*

Recall that a *bridge* (or cut edge) of a connected graph G is an edge $e \in E(G)$, the deletion of which cuts G into two connected components G_1 and G_2. A generalized tree not containing an internal bridge is called a *generalized star graph.* Explaining the terminology, the procedure of shrinking all odd-length cycles in such a graph into one vertex results in a star graph. In terms of generalized star graphs, our result on isomorphic completeness of classes of deterministic soliton automata is as follows.

Theorem 12.24. *A class $\mathcal{K}$ of deterministic soliton automata is isomorphically complete with respect to the α_i-product ($i \geq 1$) iff for every integer $n \geq 2$, there exists an $\mathcal{A}_{G_n} \in \mathcal{K}$ such that G_n contains a nice subgraph that reduces to a generalized n-star.*

Notice that the graph G_n above cannot be a chestnut containing $m \geq n$ external vertices, because any possible subgraph of a chestnut that reduces to an n-star with $n \geq 2$ fails to be nice.

Taking into account that each component of a strongly deterministic soliton graph is either a chestnut or a tree, we obtain the counterpart of Theorem 12.24 for strongly deterministic soliton automata.

Corollary 12.25. *A class $\mathcal{K}$ of strongly deterministic soliton automata is isomorphically complete with respect to the α_i-product ($i \geq 1$) iff for every integer $n \geq 2$, there exists an $\mathcal{A}_{G_n} \in \mathcal{K}$ such that a non-chestnut component of G_n contains a subgraph that reduces to an n-star.*

Indeed, every subgraph of a tree that reduces to a star is necessarily nice.

By the above results it is easy to construct the simplest isomorphically complete system for deterministic soliton automata.

Corollary 12.26. *The class of soliton automata $\mathcal{A}_{G_n}$ associated with an n-star graphs $n \geq 2$ is isomorphically complete with respect to the α_1-product.*

Our final result shows that, just as in the strongly deterministic case (see Theorem 12.18) homomorphic and isomorphic representations are equivalent with respect to the general product.

Theorem 12.27. *A class $\mathcal{K}$ of deterministic soliton automata is isomorphically (homomorphically) complete with respect to the general product (respectively, the α_i-product with $i \geq 2$) iff $\mathcal{K}$ contains an automaton $\mathcal{A}_G$ such that G satisfies one of the following conditions:*

(i) G has a nontrivial external elementary component;

(ii) G has at least two chestnut components, or a single one with at least two external vertices.

12.7 Algorithms for Soliton Automata

In this section we consider two algorithmic problems related to soliton automata: the automaton construction problem, and the problem of deciding if an arbitrary graph G is a deterministic viable soliton graph.

12.7.1 *The Automaton Construction Problem*

One of the most fundamental algorithmic problems for soliton automata, like in circuit theory in general, is the verification and simulation of soliton circuits. This problem is motivated by the practical demand that a circuit be mathematically verified for its possible use before attempting to build it [35]. Translating the above to the language of soliton automata, the following challenge must be dealt with.

Automaton Construction Problem (ACP): *Given a soliton graph G. Construct the automaton $\mathcal{A}_G$ associated with G.*

The above problem was algorithmically analyzed in [49], while in [52] the complexity of the method has been improved.

In order to solve ACP, a procedure is needed first to enumerate the set S_G of states of G. Then, after having obtained the state set, an algorithm for constructing the transition function must be designed.

The first problem can be solved by adopting an extension of the method suggested in [41] for bipartite graphs with perfect matchings. It is assumed that a random state M of G has been previously found. This can be achieved by a simple modification of any known matching algorithm (see eg. [31]). A concrete alternating unit α with respect to M must also be provided. The algorithm will use the straightforward observation that a perfect internal matching is not unique iff it contains an alternating unit.

The idea borrowed from [41] is to define a procedure

$$NEWSTATES\ (G',M',\alpha',L')$$

for any nice subgraph G' of G, perfect internal matching M' of G', M'-alternating unit α' and perfect internal matching L' of $G \setminus V(G')$. This procedure, when invoked with the initial parameters (G, M, α, $\emptyset$), will recursively generate all the additional perfect internal matchings of G' and states of G by adding the edges in L' to a perfect internal matching of G'. Note that *NEWSTATES* is invoked only when M' is not unique.

After a careful complexity analysis of the method above, the following result is obtained.

Theorem 12.28. [49] *Let G be a soliton graph with $m = E(G)$ and $k = |S_G|$. Then S_G can be constructed in $\mathcal{O}(k \cdot m)$ time.*

We are now left with the problem of effectively constructing the transition function of $\mathcal{A}_G$. The basic idea of the solution is to determine the set of input pairs (v, w) for any states $M_1, M_2 \in S_G$ such that $M_2 \in \delta(M_1, (v, w))$. First consider the case $M_1 \neq M_2$. In this situation we can build on the structure of the symmetric difference $N(M_1, M_2)$ of M_1 and M_2.

Theorem 12.29. [9] *Let M_1, M_2 be distinct states of soliton automaton $\mathcal{A}_G = (S_G, (X \times X), \delta)$. Then for every pair of external vertices $(v, w) \in X \times X$, $M_2 \in \delta(M_1, (v, w))$ holds iff one of the following conditions is satisfied by each M_1-alternating trail α of $N(M_1, M_2)$:*

(a) α is an even M_1-alternating cycle accessible from v in M_1.

(b) α is an M_1-alternating crossing from v to w. In this case $v \neq w$ holds and α is the unique crossing in $N(M_1, M_2)$.

Applying the above result, our problem is reduced to testing the accessibility of alternating cycles in $N(M_1, M_2)$ by M_1-alternating paths starting from v. Using an appropriate efficient alternating path procedure (cf. [55]) for this purpose, we obtain the following result.

Theorem 12.30. *Let M_1 and M_2 be distinct states of $\mathcal{A}_G$, $m = |E(G)|$ and let l denote the number of external vertices. Then the set of input pairs (v, w) for which $M_2 \in \delta(M_1, (v, w))$ can be constructed in $\mathcal{O}(l \cdot m)$ time.*

Having found the transitions between distinct states, we move on to identifying *self-transitions* , which are highlighted by soliton trails in the graph. (See Theorem 12.12.) For an arbitrary external vertex v and state M of G, construct the graph $G[M, v]$ determined by the edges traversed by an M-alternating path or an M-alternating soliton trail starting from v. Then the main point of our algorithm is the following observation.

Theorem 12.31. [9] *For any state M of soliton automaton $\mathcal{A}_G$ and for any external vertex v of G, $M \in \delta(M, (v, v))$ iff one of the following conditions holds:*

(a) $G[M, v]$ is a non-bipartite graph.

(b) $G[M, v]$ is a bipartite graph containing an M-alternating double soliton c-trail from v.

(c) $G[M, v]$ is a bipartite graph not containing an M-alternating even-length cycle.

On the basis of the above theorem it is possible to decide for any state M and external vertex v of G if $M \in \delta(M, (v, v))$ holds. For the concrete implementation one needs an efficient procedure to construct $G[M, v]$ and one to search for alternating cycles with certain properties in $G[M, v]$.

Locating $G[M, v]$ is feasible again by applying certain standard alternating path procedures (cf. [55]), therefore it can be accomplished in linear time. Testing the bipartite property of graphs is also linear by applying a simple breadth-first search, while the existence of an alternating cycle can be checked using the method of [30] having the same complexity. It remains to search for double soliton c-trails. This problem has been solved in [49] by a $\mathcal{O}(|V| \cdot |E|)$-time algorithm using a simple breadth-first procedure, which was later developed into a linear-time algorithm in [52].

In summary, we have obtained the following general result.

Theorem 12.32. [52] *Let G be a soliton graph with $m = |E(G)|$, $k = |S_G|$ and l denoting the number of external vertices. Then ACP can be solved in $\mathcal{O}(k^2 \cdot l \cdot m))$ time.*

12.7.2 *The Deterministic Property of Soliton Graphs*

In this subsection we deal with the problem of deciding if an arbitrary graph G is a deterministic viable soliton graph. As expected, the decision will be based on the characterization of deterministic soliton graphs given in Section 12.5.2. The results reported are quoted from [17] and [18].

In the light of Theorems 12.8 and 12.10, a connected graph G is a deterministic viable soliton graph iff G is a chestnut, or G is a viable soliton graph such that its external elementary components reduce to a generalized tree and the full subgraph induced by the (vertices of the) internal elementary components of G has a unique perfect matching.

It is straightforward to decide if a connected graph G is a chestnut. The simplest way to accomplish this goal is to reduce G, and see if $r(G)$ is a baby chestnut, so that no loops have been eliminated during reduction. The complexity of this trivial algorithm is linear, even if we did not have a general algorithm to perform reduction in linear time. Fortunately, however, we do have one [18], which we are going to apply in the decision of the general problem.

We begin with the outline of this linear-time reduction algorithm. The result is interesting by itself, for reduction can be used as a speed-up in a number of other fundamental graph algorithms as well. See [18] for some of the interesting graph theory applications of reduction.

Two redexes are said to be *connected* if they have at least one focus in common. A collection of connected redexes is called a *redex grove.* A redex grove R is *maximum* if there does not exist another grove R' such that $\cup R \subset \cup R'$. (Note that $\cup R$ is the set of vertices covered by all the individual redexes in R.)

A greedy algorithm to reduce graph G is as follows. Identify all maximum redex groves R in G, and shrink them into appropriate subgraphs G_R. During this process, eliminate all secondary loops on the fly, and continue until $r(G)$ is reached. For an example, see the graph G_0 in Figure 12.13, which contains four redexes centered at vertices 4, 7, 11, and 14. The ones at 11 and 14 determine a maximum redex grove, the shrinking of which gives rise to a new redex a in graph G_1. Shrinking at 7 and a (as a second grove) then yields vertex b, which is again a redex in G_2. Finally, shrinking the grove containing the redexes at b and 4 results in a single line, which is $r(G_0)$.

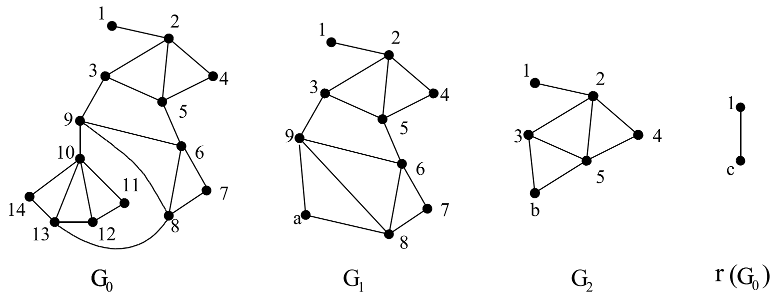


Fig. 12.13 A reduction example.

The problem with this greedy algorithm is that redexes emerge recursively when shrinking maximum redex groves. A maximum redex grove R is called an *implied redex* if the graph G_R is also a redex. In our example the redex grove containing 11 and 14 in G_0, as well as the one containing a and 7 in G_1, is an implied redex. Pursuing an unfortunate scenario, we might shrink ordinary redex groves (ones that are not implied redexes) first, and implied redexes last. The result is a reproduction of the original problem at the "macro" level, whereby the vertices are essentially the centers of the graphs G_R. The complexity of the algorithm then accumulates in a recursive way, resulting in a $\mathcal{O}(n^2)$ algorithm, where n is the number of vertices in G.

In order to bring down the complexity of the algorithm to $\mathcal{O}(m)$, where m is the number of edges in G, we need to locate implied redexes (even recursively emerging ones, like the cascade a, b of redexes in our example) first, without performing any actual shrinking. Then, after replacing each implied redex with a single redex, we come back to the greedy algorithm desribed above to finish the job. An elegant way to find implied redexes is to impose the structure of a depth-first tree on G. One can then design a variation of an attribute grammar [43] to identify implied redexes and replace them by a single redex during a bottom-up sweep of the tree. See [18] for the details. As a result, the following theorem is obtained.

Theorem 12.33. *The graph $r(G)$ can be constructed in $\mathcal{O}(m)$ time.*

As a corollary to Theorem 12.33 one can decide in $\mathcal{O}(m)$ time if graph G is a deterministic elementary soliton graph as follows. Reduce G, and see if $r(G)$ is a generalized tree. If it is not, then G is not an elementary deterministic soliton graph. If it is, then construct a perfect internal matching M for $r(G)$ and reverse the steps of the reduction procedure to check if inverse reduction preserves the elementary property. Graph G is elementary deterministic iff a positive answer is obtained in each step of the inverse reduction process. It is easy to see that the desired matching M can be found for $r(G)$ in $\mathcal{O}(m)$ time, and the check for the elementary property during inverse reduction takes only a constant amount of time in each step. Thus, we have the following result.

Theorem 12.34. *It is decidable in $\mathcal{O}(m)$ time if an arbitrary graph G is a chestnut or an elementary deterministic soliton graph.*

The second major task in deciding if an arbitrary graph G is a deterministic viable soliton graph is to isolate the potential internal components, and check the full subgraph induced by them for the unique perfect matching (UPM) property. We elaborate on this issue following the steps of the construction presented in [17].

Let e be a bridge in a connected graph G (open or closed), separating two connected components G_1 and G_2. The bridge e is called *odd* if G_1 and G_2 are both closed, having an odd number of vertices, *semi-odd* if G_1 is odd closed and G_2 is open, and *open* if both G_1 and G_2 are open. A bridge of an arbitrary graph G is one of a connected component of G.

The following theorem is a generalization of an old result by Kotzig [44] on graphs having a UPM. Kotzig's result says that every graph having a UPM contains a bridge that belongs to that matching. Our generalization follows directly from Theorem 12.1, observing that the mandatory root of any internal family of a non-chestnut connected deterministic soliton graph, situated at the bottom of the Hasse diagram of $\leq$, is necessarily a semi-odd bridge.

Theorem 12.35. *Every non-chestnut connected deterministic soliton graph G having at least one internal family of elementary components contains a semi-odd bridge which belongs to every state M of G.*

Gabow *et al.* [30] gave the following algorithm to decide if a closed graph G has a UPM, and specify that matching M in case of a positive answer. Using the terminology of [30], a 2-*edge component* of graph G is a connected component of the graph remaining from G after the deletion of all bridges.

Algorithm A

```
Initialize $M = \emptyset$ and $R$ to be the set of all bridges of $G$.
While $R \neq \emptyset$ repeat the following steps:
    1. Delete an edge $(x, y)$ from $R$.
    2. If $(x, y)$ is an odd bridge, delete $(x, y)$ from $G$, add $(x, y)$ to $M$, and
       repeat the following steps for each edge $(v, w)$ incident with $x$ or $y$:
        a) Delete $(v, w)$ from $G$, and from $R$ if it is in $R$.
        b) If $v$ and $w$ are still connected but are in different 2-edge compo-
           nents (in the current graph), then:
           Find a path $p(v, w)$ connecting $v$ and $w$, and add every bridge on
           $p(v, w)$ to $R$.
        c) Delete vertices $x$ and $y$.
```

Graph G has a unique perfect matching M iff the final graph becomes empty after running the above procedure. It was shown in [30] that Algorithm A runs in $\mathcal{O}(m \log^4 n)$ time.

The correctness of Algorithm A is based on Kotzig's theorem. We are going to modify Algorithm A to be able to detach semi-odd bridges according to Theorem 12.35. The changes are summarized as follows.

(i) In the initialization, R is set to the collection of all non-open bridges. (Note that a non-open bridge need not be closed, e.g. it can be semi-odd.)

(ii) In Step 1, bridge (x, y) is considered only if it is odd or semi-odd, and in Step 2b only non-open bridges are added to R.

In the light of Theorem 12.35, if G is a non-chestnut connected viable deterministic soliton graph, then the modified algorithm will remove every internal family of G after checking each of these families for the UPM property. Observe that the remainder graph at this point is still a collection of external families, not components. Conversely, if, after running the modified algorithm, the remainder of G consists of a number of open components with open bridges only (if any), then the closed subgraph induced by the deleted vertices has a UPM. It is easy to see that the above changes do not affect the complexity of Algorithm A.

After Algorithm A has successfully removed all potential internal families of G, it will pause for a while before it resumes on a modified version of the remainder graph G'. The pause will last $\mathcal{O}(m)$ time, during which a simplification of G' takes place. We now describe the actions to be taken during the pause.

Recall from [55] that a *factor-critical* graph is a closed graph G for which $G - v$ has a perfect matching for each $v \in V(G)$. Such a matching is called a *near-perfect matching.* Let G be a connected viable deterministic soliton graph different from a chestnut. By *pruning* G we mean deleting all of its external vertices and edges. Let $pr(G)$ denote the resulting closed graph. Furthermore, for each external $e \in E(G)$, let $pr_e(G)$ be the open graph obtained from $pr(G)$ by putting back the edge e only. Adding a loop around the external endpoint of e then results in a closed graph, denoted $cpr_e(G)$.

Theorem 12.36. *If $pr(G)$ does not contain bridges, then it is factor-critical. Moreover, for every external edge e, $cpr_e(G)$ has a UPM.*

Now let G be an arbitrary soliton graph consisting of a single external family, and assume that G does contain internal bridges. By Theorem 12.1, all these bridges must be open. Let $c \in E(G)$ be an internal bridge. By *cutting* c we mean deleting c from G first, then putting it back separately in both of the resulting two connected components as an external edge. Let G_1 and G_2 denote the two connected graphs obtained. Clearly, G_1 and G_2 are still soliton graphs consisting of a single external family, and G is deterministic iff both G_1 and G_2 are such. Cutting all internal bridges of G will then result in a number of soliton graphs $G_1, \ldots, G_k$, each of which is a single external family, so that G is deterministic iff all of $G_1, \ldots, G_k$ are such.

Combining the above argument with Theorem 12.36, we decide if a connected open graph G containing open internal bridges only is a deterministic soliton graph as follows.

Algorithm B

Step 1. Cut all open internal bridges in G to obtain the open graphs $G_1, \ldots, G_k$ as described above.
Step 2. In each G_i still containing internal edges, keep one external edge e_i arbitrarily and delete the rest to obtain a graph $\bar{G}_i$.
Step 3. For each $1 \leq i \leq k$, check if $\bar{G}_i$ has a unique perfect matching, and if so, find that matching M_i by applying Algorithm A.
Step 4. Taking M_i as a state of G_i, find those vertices $X_i \subseteq V(G_i)$ that lie on some crossing with respect to M_i. At the same time, locate all M-alternating ears attached to the subgraph $G_i[X_i]$ induced by X_i and augment this graph by edges connecting the two endpoints of such ears (the hidden edges).
Step 5. Check if $G_i[X_i]$, augmented by the hidden edges, reduces to a generalized tree. Graph G is a deterministic soliton graph iff a positive answer is obtained for each i.

Theorem 12.37. *Algorithm B correctly decides in $\mathcal{O}(m)$ time if a connected open graph G containing open internal bridges only is a deterministic soliton graph.*

Algorithms A and B can now be combined into one algorithm as follows. First apply Algorithm A on G to recursively identify all odd and semi-odd bridges. When this algorithm stops, a pause follows. During the pause apply Steps 1 and 2 of Algorithm B to each connected component of the current graph. At the same time, "freeze" (i.e., memorize) the graphs G_i obtained in Step 1. The pause lasts $\mathcal{O}(m)$ time, during which the graph has simplified.

After the pause add the edges e_i specified in Step 2 to the set R of bridges, and continue running the main loop of Algorithm A all the way until the graph becomes empty or an error occurs. Since the size of the graph after the pause is not greater than that of the original G, the overall complexity of Algorithm A is still $\mathcal{O}(m \log^4 n)$.

At the end, with the matchings M_i obtained, perform the required checks described in Steps 4 and 5 of Algorithm B on the graphs G_i.

At this point we have succesfully decided if G is a deterministic soliton graph with all of its internal elementary components being mandatory. In case of a positive answer, we have also found a state M for G. In total, we have used $\mathcal{O}(m \log^4 n)$ time only, but we still need to check if G is viable. Knowing state M, this check can easily be performed in $\mathcal{O}(m)$ time by the algorithm given in [11] to find the accessible vertices and isolate the families of an arbitrary soliton graph.

As a corollary to the results presented in this subsection, we obtain the following theorem.

Theorem 12.38. *It is decidable in $\mathcal{O}(m \log^4 n)$ time if an arbitrary graph G with n vertices and m edges is a viable deterministic soliton graph.*

In the presence of a perfect internal matching M, the decision if G is a viable deterministic soliton graph can already be done in $\mathcal{O}(m)$ time. Indeed, as outlined in Steps 4 and 5 of Algorithm B, one can detach the external elementary components of G in $\mathcal{O}(m)$ time, and check them for the elementary deterministic property using Theorem 12.34. On the other hand, the remainder graph G', consisting of the internal elementary components of G, can also be checked for the UPM property in $\mathcal{O}(m)$ time, knowing that $M_{G'}$ is a perfect matching. This result was proved in [30].

12.8 Extensions of the Model and Further Research

In this section we present a few ideas leading to possible extensions of the soliton automaton model. These extensions may open up new directions both at the theoretical and application-oriented levels to enhance the power of soliton automata.

12.8.1 *Soliton Automata with Outputs*

As a new feature compared to previous definitions of soliton automata, we extend $\mathcal{A}_G$ with a binary output function ζ defined by

$\zeta(M,(v,w)) = 1$ iff there exists a soliton walk from v to w.

The rationale for this extension is the following. We do not want the automaton $\mathcal{A}_G$ to crash upon receiving an input (v,w) for which there exists no soliton walk from v to w in a given state M. This is reflected by the standard definition of δ. On the other hand, if $v = w$, then there might as well exist a soliton walk α from v to v, the traversal of which does not change the current state M. If all soliton walks from v to v are of this nature, then the only way to distinguish between having or not having a soliton walk is to introduce a binary output. The practical importance of this observation is clear: we can actually build a variety of interesting flip-flops using simple soliton automata.

Example 12.39. (Binary flip-flop) Consider the top graph G in Figure 12.14a with two external vertices. The automaton $\mathcal{A}_G$ has two states, which are shown below G in Figure 12.14a. (Double lines indicate positive edges in the figure.) Without the output feature $\mathcal{A}_G$ would be isomorphic to $\mathcal{A}_{G'}$, where G' is the graph appearing on the top of Figure 12.14b. In $\mathcal{A}_G$, the inputs $(1, 1)$ and $(2, 2)$ can be used to test the current state through the output provided by the automaton. The same test is not possible in $\mathcal{A}_{G'}$.

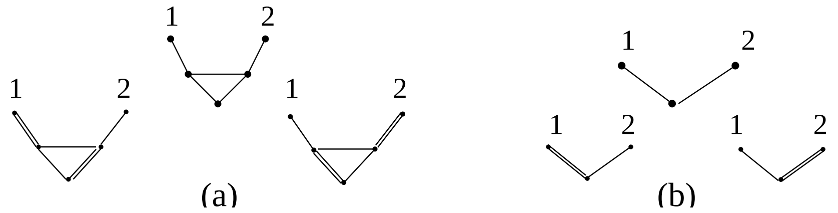


Fig. 12.14 A binary flip-flop.

Example 12.40. (Ternary flip-flop) The underlying graph G, and the four states of the soliton automaton $\mathcal{A}_G$ are shown in Figure 12.15. This automaton comes with a distinguished "initial" state, in which all three external vertices are covered by the corresponding matching. The initial state is recognized by getting a negative output on each of the inputs $(1, 1)$, $(2, 2)$, and $(3, 3)$. The transition function and monoid of $\mathcal{A}_G$ is described in [24].

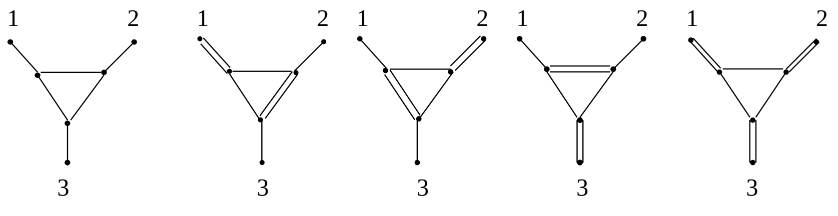


Fig. 12.15 A ternary flip-flop.

Following the pattern of Examples 12.39 and 12.40 one can easily design flip-flops with an arbitrary number of states as soliton automata. In order for this idea to work in practice, one needs an interface sophisticated enough to pick up the output signals provided by soliton automata.

12.8.2 *Generalization of Soliton Automata*

A more careful look at the definition of soliton automata reveals that the restriction requiring the external vertices to have degree zero or one is mainly technical. Therefore, as a generalization, it is natural to consider matching based automata with designated external vertices of an arbitrary degree, and to investigate their relationship to standard soliton automata.

In the generalized model, each internal vertex of the network is incident with a unique marked (positive) edge in every state. We shall use the name “open conjugated system” to identify this model. The analogy is taken from chemistry, where a system of atoms covalently bonded with alternating single and multiple (e.g. double) bonds in a molecule of an organic compound is called conjugated. In any state of a chemical conjugated system, the multiple bonds are considered as ”marked” from the percolation point of view.

Spelling out the above heuristic definition, in the underlying graph of an open conjugated system there is a distinguished subset of vertices corresponding to some interface points, which are referred to as *external vertices.* Analogously to the soliton automaton concept, *internal vertices* are those that are not external. Therefore the underlying graph G is identified as a triple (V, E, T), where $T \subseteq V$ denotes the set of internal vertices. In this context, graph G is called *open*, if $V \neq T$.

A *perfect T-matching* is a matching that covers all of the internal vertices. An open graph $G = (V, E, T)$ having a perfect T-matching is called an *open conjugated system.* The states of an open conjugated system are the perfect T-matchings. Now all the standard definitions relating to soliton automata can be adopted for open conjugated systems in a natural way, substituting the concept soliton walk with that of a *total external alternating walk* [52]. The obvious restriction for such walks is that, whenever the walk continues on an unmarked (negative) edge, then there are no marked (positive) edges available to chose from. Unlike in standard soliton graphs, however, this condition applies at the endpoints of the walk, too. The concept of switching on a total external alternating walk is then adopted without a change, and it is easily verified that switching always results in a new state.

For any open conjugated system $G = (V, E, T)$ it is easy to construct an isomorphic standard soliton automaton. To this end let w be an internal vertex of G having degree 1. It is clear that the unique edge (w, u) is mandatory, while all the other edges incident with u are necessarily forbidden (not contained in any state). Therefore the graph $G - w - u$ defines an isomorphic automaton. After removing all such "false" external vertices iteratively, we end up with an isomorphic open conjugated system $G' = (V', E', T')$ having no internal vertices of degree 1. Now attach an extra edge (v, v') to each external vertex $v \in V \setminus T$ to obtain graph G'' in which the role of being external is taken over by the new vertices v'. It is clear that G'' is still isomorphic to G, and it is a standard soliton graph (having no isolated external vertices). In summary we have proved the following result.

Theorem 12.41. *Every open conjugated system is isomorphic to a soliton automaton.*

Starting from Theorem 12.41, a possible future research objective is to see if any type of network-based automata associated with graph matchings or their generalizations have a stronger computational power than soliton automata. Since the first generalization of matchings was the f-factor concept [63], it is a natural to first extend our model in this direction. Recall that an *f-factor* of a graph is a spanning subgraph having a prescribed degree $f(v)$ at each vertex v. If we drop the above degree-prescription for certain external vertices, then, as a generalization of perfect T-matchings, we obtain the concept of open f-factors. Graphs with open f-factors could serve as a model to such generalized open conjugated systems in which a molecule/atom corresponding to an internal vertex can be connected to its surroundings by several double bonds. The definition of alternating walks between external vertices can be adopted without a change, and switching on these walks will still produce a new state for the graph. These definitions give rise to the model of f-factor automata. Even though the description of this model is still in the preparatory phase, the abundance of results on f-factors promises with great success. The structure theory for f-factors in [54] could be an excellent starting point for the analysis of these automata.

12.9 Summary

We have presented a comprehensive overview of the most important results on soliton graphs and automata. The results have been divided in four major topics: the decomposition of soliton graphs and automata, the characterization of different classes of soliton automata, complete systems, and algorithms.

Regarding the first topic, the results were centered around the decomposition of soliton graphs into elementary components, and the grouping of these components into families arranged in a partial order that reflects the order in which they can be reached by alternating paths starting from the external vertices. The decomposition of soliton graphs then gave rise to that of their automata, using a special restricted form of the α_0-product.

Concerning the second topic, the automata of the following classes of soliton graphs have been characterized through their transition monoids: chestnut graphs, trees, soliton graphs with a single cycle, and constant soliton graphs. In addition, a complete state-independent structural characterization was given for deterministic soliton graphs in general.

As to the third topic, homomorphically and isomorphically complete classes of soliton automata have been discussed with respect to the α_i-products. Classes of particular interest were: commutative permutation automata, deterministic and strongly deterministic soliton automata, and classes containing a single automaton.

The fourth topic was about finding an algorithmic solution to two separate problems: the automaton construction problem, and the decision problem for deterministic soliton graphs. Regarding the first problem, it was shown that the automaton of a soliton graph G can effectively be constructed in $\mathcal{O}(k^2 \cdot l \cdot m)$ time, where m, l, and k are the number of edges, the number of external vertices, and the number of states in G, respectively. Deciding the detrministic property of soliton graphs turned out to be a more complex issue, requiring two sub-ordinate algorithms which are also relevant in their own right. We have presented an $\mathcal{O}(m)$-time algorithm for reducing a graph to one that is free from subdividing vertices. By the help of this algorithm we could decide if an arbitrary graph G is a chestnut or an elementary deterministic soliton graph in linear time. Then we have worked out a modification of the currently fastest unique perfect matching algorithm, and gave a $\mathcal{O}(m \cdot \log^4 n)$-time algorithm to decide the general problem whether graph G is a viable deterministic soliton graph.

Finally, in Section 12.8 we have presented an interesting extension of the soliton automaton model with output, and considered some other possible generalizations concerning open conjugated systems.

References

1. Adamatzky, A. (ed.) (2002). *Collision-Based Computing* (Springer-Verlag).
2. Ahuja, R. K., Magnanti, T. L. and Orlin, J. B. (1993). *Network Flows: Theory, Algorithms, and Applications* (Prentice Hall, Upper Saddle River, New Jersey).
3. Albert, R. and Barabási, A.-L. (2002). Statistical mechanics of complex networks, *Reviews of Modern Physics* **74**, pp. 47–97.
4. Barendregt, H. P. (1984). *The Lambda Calculus: Its Syntax and Semantics*, 2nd edn. (North-Holland, Amsterdam).
5. Bartha, M. (1997). The Gallai-Edmonds algebra of graphs, *Congressus Numerantium* **123**, pp. 205–219.
6. Bartha, M. and Gombás, E. (1991). A structure theorem for maximum internal matchings in graphs, *Information Processing Letters* **40**, pp. 289–294.
7. Bartha, M. and Gombás, E. (1995). On graphs with perfect internal matchings, *Acta Cybernetica* **12**, pp. 111–124.
8. Bartha, M. and Jürgensen, H. (1989). Characterizing finite undirected multigraphs as indexed algebras, Tech. Rep. 252, Department of Computer Science, The University of Western Ontario, London, Canada.
9. Bartha, M. and Krész, M. (2000). Elementary decomposition of soliton automata, *Acta Cybernetica* **14**, pp. 631–652.
10. Bartha, M. and Krész, M. (2001). On the immediate predecessor relationship in soliton graphs, *Congressus Numerantium* **150**, pp. 15–31.
11. Bartha, M. and Krész, M. (2002). Isolating the families of soliton graphs, *Pure Mathematics and Applications* **13**, pp. 49–62.
12. Bartha, M. and Krész, M. (2003). Structuring the elementary components of graphs having a perfect internal matching, *Theoretical Computer Science* **299**, pp. 179–210.
13. Bartha, M. and Krész, M. (2004). Tutte type theorems in graphs having a perfect internal matching, *Information Processing Letters* **91**, pp. 277–284.
14. Bartha, M. and Krész, M. (2006a). Deterministic soliton graphs, *Informatica* **30**, pp. 281–288.
15. Bartha, M. and Krész, M. (2006b). Flexible matchings, *Lecture Notes in Computer Science* **4271**, pp. 313–324.
16. Bartha, M. and Krész, M. (2008). Splitters and barriers in open graphs having a perfect internal matching, *Acta Cybernetica* **18**, pp. 697–718.
17. Bartha, M. and Krész, M. (2009a). Deciding the deterministic property for soliton graphs, *Ars Mathematica Contemporanea* **2**, pp. 121–136.
18. Bartha, M. and Krész, M. (2009b). A depth-first algorithm to reduce graphs

in linear time, in *Proceedings of the 11th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing* (IEEE Computer Society Press), pp. 273–281.

19. Boldt, O. (1995). *Abschlusseigenschaften von Solitonschprachen*, Master's thesis, Universitate Magdeburg.
20. Boldt, O. and Jürgensen, H. (2007). Soliton languages are nearly an anti-AFL, *Int. J. Found. Comput. Sci* **18**, pp. 1161–1165.
21. Carter, F. L. (1984). The molecular device computer: Point of departure for large scale cellular automata, *Physica D* **10**, pp. 175–194.
22. Carter, F. L., Schultz, A. and Duckworth, D. (1987). Soliton switching and its implications for molecular electronics, in F. L. Carter (ed.), *Molecular Electronic Devices II* (Marcel Dekker Inc.), pp. 149–182.
23. Cassandras, C. G. and Lafortune, S. (2008). *Introduction to Discrete Event Systems*, 2nd edn. (Springer).
24. Dassow, J. and Jürgensen, H. (1990). Soliton automata, *J. Comput. System Sci.* **40**, pp. 154–181.
25. Dassow, J. and Jürgensen, H. (1991). Soliton automata with a single exterior node, *Theoretical Computer Science* **84**, pp. 281–292.
26. Dassow, J. and Jürgensen, H. (1993). Soliton automata with at most one cycle, *Journal of Computer and System Sciences* **46**, pp. 155–197.
27. Dassow, J. and Jürgensen, H. (1995). The transition monoids of soliton trees, in *Mathematical Linguistics and Related Topics. Papers in Honour of Solomon Marcus on his 70th Birthday* (Editura Academiei Romane, Bucuresti), pp. 76–87.
28. Ésik, Z. (1985). Homomorphically complete classes of automata with respect to the α_2-product, *Acta Sci. Math.* **48**, pp. 135–141.
29. Feynman, R. P. (1961). There's plenty of room at the bottom, in H. D. Gilbert (ed.), *Miniaturization* (Reinhold, New York), p. 282.
30. Gabow, H. N., Kaplan, H. and Tarjan, R. E. (2001). Unique maximum matching algorithms, *Journal of Algorithms* **40**, pp. 159–183.
31. Gabow, H. N. and Tarjan, R. E. (1991). Faster scaling algorithms for general graph-matching problems, *J. Assoc. Comput. Mach.* **38**, pp. 815–853.
32. Gécseg, F. (1986). *Products of Automata* (Akademie-Verlag, Berlin).
33. Gécseg, F. and Imreh, B. (1995). On completeness of nondeterministic automata, *Acta Math. Hungar.* **68**, pp. 151–159.
34. Gécseg, F. and Jürgensen, H. (1990). Automata represented by products of soliton automata, *Theoretical Computer Science* **74**, pp. 163–181.
35. Groves, M. P. (1988). Towards verification of soliton circuits, in F. L. Carter, R. E. Siatkowski and H. Wohltjen (eds.), *Molecular Electronic Devices* (North-Holland, Amsterdam), pp. 287–302.
36. Groves, M. P. (1997). Soliton circuit design using molecular gate arrays, in *Proceedings of the 20th Australasian Computer Science Conference*, pp. 245–252.
37. Groves, M. P., Carvalho, C. F., Marlin, C. D. and Prager, R. H. (1993). Using soliton circuits to build molecular memories, *Australian Computer Science Communications* **15**, pp. 37–45.

38. Groves, M. P., Carvalho, C. F. and Prager, R. H. (1995). Switching the polyacetylene soliton, *Materials Science and Engineering* **C3**, pp. 181–185.
39. Groves, M. P. and Marlin, C. D. (1995). Using soliton circuits to build molecular computers, *Australian Computer Science Communications* **17**, pp. 188–193.
40. Imreh, B. and Ito, M. (1997). On α_i-product of nondeterministic automata, *Algebra Colloquium* **4**, pp. 195–202.
41. Itai, A., Rodeh, M. and Tanimoto, S. (1978). Some matching problems for bipartite graphs, *Journal of the ACM* **25**, pp. 517–525.
42. Jürgensen, H. and Kraak, P. (2007). Soliton automata based on trees, *J. Found. Comput. Sci* **18**, pp. 1257–1270.
43. Knuth, D. E. (1968). Semantics of context-free languages, *Math. Systems Theory* **2**, pp. 127–145.
44. Kotzig, A. (1959). On the theory of finite graphs with a linear factor i, *Mat.-Fyz. Casopis Slovensk. Akad. Vied* **9**, pp. 73–91.
45. Kraak, P. (2005). *Solitonautomaten*, Master's thesis, Universitate Potsdam.
46. Krész, M. (2002). Alternating cycles in soliton graphs, *WSEAS Transactions on Mathematics* **1**, pp. 165–170.
47. Krész, M. (2004). *Soliton automata: a computational model on the principle of graph matchings*, Ph.D. thesis, University of Szeged, Szeged, Hungary.
48. Krész, M. (2005a). Isomorphically complete systems of soliton automata, in *Proceedings of the 11th International Conference on Automata and Formal Languages*, pp. 150–163.
49. Krész, M. (2005b). Simulation of soliton circuits, *Lecture Notes in Computer Science* **3845**, pp. 347–348.
50. Krész, M. (2007a). Graph decomposition and descriptional complexity of soliton automata, *Journal of Automata, Languages and Combinatorics* **12**, pp. 237–263.
51. Krész, M. (2007b). Nondeterministic soliton automata with a single external vertex, in *Proceedings of the 1st International Conference on Languages, Automata Theory and Applications*, pp. 319–330.
52. Krész, M. (2008a). Automata associated with open conjugated systems, in *Proceedings of the 12th International Conference on Automata and Formal Languages*, pp. 232–244.
53. Krész, M. (2008b). Soliton automata with constant external edges, *Information and Computation* **206**, pp. 1126–1141.
54. Lovász, L. (1972). On the structure of factorizable graphs II, *Acta Math. Acad. Sci. Hungar.* **23**, pp. 465–478.
55. Lovász, L. (1986). *Matching Theory* (North Holland, Amsterdam).
56. Newman, M. E. J. (2000). Models of the small world, *Journal of Statistical Physics* **101**, pp. 819–841.
57. Newman, M. E. J. (2003). The structure and function of complex networks, *SIAM Review* **45**, pp. 167–256.
58. Newman, M. E. J. and Watts, D. J. (1999). Scaling and percolation in the small-world network model, *Phys. Rev. E* **60**, pp. 7332–7342.
59. Papatheodorou, T. S., Ablowitz, M. J. and Saridakis, Y. G. (1988). A rule

for fast computation and analysis of soliton automata, *Stud. Appl. Math.* **79**, pp. 173–184.
60. Park, J. K., Steiglitz, K. and Thurston, W. P. (1986). Soliton-like behaviour in automata, *Physica D* **19**, pp. 423–432.
61. Sienko, T., Adamatzky, A., Rambidi, N. and Conrad, M. (eds.) (2003). *Molecular Computing* (The MIT Press, London).
62. Steiglitz, K., Kamal, I. and Watson, A. (1988). Embedded computation in one-dimensional automata by phase coding solitons, *IEEE Trans. Comput.* **37**, pp. 138–145.
63. Tutte, W. T. (1952). The factors of graphs, *Canad. J. Math.* **4**, pp. 314–328.
64. Wielandt, H. (1964). *Finite permutation groups* (Academic Press, New York).
65. Yin, Y. M. and Lin, X. Q. (2001). Progress on molecular computer, *Progress in Chemistry* **13**, pp. 337–342.

This page is intentionally left blank

Chapter 13

Inferring Leadership Structure from Data on a Syntax Change in English

W. Garrett Mitchener
*College of Charleston Math Department,
66 George St. Charleston, SC 29424 USA,
E-mail:* `MitchenerG@cofc.edu`

In a typical human population, some features of the language are bound to be in flux. Variation in each individual's usage rates of optional features reflects language change in progress. Sociolinguistic surveys have determined that some individuals use new features to a greater degree than the population average, that is, they seem to be leading the change. This article describes a mathematical model of the spread of language change inspired by a model from population genetics. It incorporates the premise that some individuals are linguistic leaders and exert more influence on the speech of learning children than others. Using historical data from the rise of *do*-support in English, a maximum likelihood calculation yields an estimate for the influence ratio used in the model. The influence ratio so inferred indicates that 19 of the 200 simulated individuals account for 95% of the total influence, confirming that language change may be driven by a relatively small group of leaders. The model can be improved in any number of ways, but additional features must be selected carefully so as not to produce a computationally intractable inference problem. This project demonstrates how data and techniques from different subfields of linguistics can be combined within a mathematical model to reveal otherwise inaccessible information about language variation and change.

13.1 Introduction

The purpose of this chapter is to introduce a sociolinguistic mathematical model of a structured population of speakers and integrate it with historical data. The project began with a conversation between the author and mathematical biologist Martin Nowak, in which the question at hand was originally posed as: Given data from a language change, is it possible to infer a population size, and what would that number actually mean? The idea of inferring a population size might suggest trying to estimate how many people lived in medieval England based on the writings of Chaucer, but this is not what that question is meant to ask. Rather, the intent is to formulate a mathematical model with some parameter that represents the number of individuals relevant to some feature of the overall population, such as the size of an average town or friendship network, then to estimate the value of that parameter from data. There are many potentially applicable mathematical models in the population genetics literature in which several genetic variants of a particular species are present, but one of them eventually takes over the entire population. This is called *fixation*. It seems reasonable to interpret variants of a language analogously to genetic variants of a species and to investigate what data about one variant's route to fixation might say about the underlying population. Specifically, the original question of inferring a population size is better posed as follows: Is the population homogeneous, or are some individuals more important than others in driving the change? Is there perhaps a small core of leaders that switch to a new language variant, and the rest of the population simply learns from them and reflects their speech patterns? Inferring a size from language change data would presumably reflect the size of this core. The central task of this chapter is to explain how such a calculation can be carried out, thereby reconstructing the linguistic leadership structure of medieval English society from data on a change in syntax.

The model developed here adapts a genetic model to incorporate sociolinguistic observations. It is then fit to historical data from a syntax change in English. The inferred parameter yields an estimate for the size of the leadership core. This project is part of a growing body of syntheses of linguistic fields and mathematical modeling methods that were generally not combined until the 1990s or so. Some remarks are in order about the challenges and potential of such syntheses.

Mathematics has traditionally been applied with great success to linguistic sub-fields related to formal grammar and the idealized speaker.[1] Statistical studies are essential for understanding language variation and change. Linguists typically use well established tools such as hypothesis tests, VARBRUL, or ANOVA, but these tools have their limits. Recent studies have taken advantage of ever more sophisticated statistical models for analyzing historical data [24, 57]. This chapter introduces a new kind of tool to this collection and shows how it can deduce unexpected information from well known data.

The use of biological models as bases for linguistic models has been very useful in recent studies of the biological evolution of language,[2] and the study of language change on historical time scales.[3] For example, the logistic sigmoid function, a well known model of the S-curve characteristic of language change, has its roots in the study of population growth in a constrained environment. Many introductory textbooks on differential equations teach logistic growth in conjunction with census data; for example there is a project on the subject in [12]. It should be noted that the statistical curve fitting method in that project is somewhat suspect, but since it is within a textbook for a first course on differential equations, there is justification for not choosing a more robust method that might distract students from the primary topic. However, this textbook project highlights a cultural difficulty within mathematics. The mathematical subfield of dynamical systems focuses on the precise and the nonlinear. Statistical inference on the other hand must deal with noisy discrete data, and is frequently limited to linear methods. Combining these two fields correctly can be difficult, and as in the textbook, circumstances often dictate that one field be sacrificed in favor of the other. A better resolution is to use tools from the areas where dynamical systems theory and statistics overlap: Markov chains, and maximum likelihood inference methods.

In addition to mathematical cultural difficulties, this project attempts to combine historical linguistics and sociolinguistics in an unusual way. Sociolinguists have established that social networks contribute to the spread of language changes [26–28]. Present-day investigations can partially identify the relevant social structures from inteviews that reveal details about the friendship networks and speech patterns of many individuals. Such studies that produce a snapshot of the state of a language at a single moment in

[1]See for example [7–9, 14, 18, 25, 53, 55].
[2]See for example [6, 21–23, 29, 30, 32–34, 41–47, 49, 56]
[3]See for example [4, 5, 10, 16, 17, 19, 20, 35–39, 48, 58]

time are called *synchronic*. For example, one might spend a few months recording spontaneous speech by one hundred individuals of varying ages, then estimate the formant frequencies in their vowels or how often they use certain syntactic alternatives. Assuming that adult speech changes very little, the age variation indicates speech patterns going back in time several decades in what is called *apparent time*. The interviews might also include information about socio-economic class and friendships, indicating how one person's speech might influence another's. Unfortunately, the data acquired via interviews takes so much time to gather that the data sets are often far sparser than statisticians would like. Furthermore, interviews and social networking data are generally not available for studying language changes older than the present oldest generation.

Studies of linguistic data across several decades or centuries are called *diachronic* because they compare language use from two or more separated time periods. Corpora consisting of written documents from across a wide range of time are essential to such studies. Sociolingustic studies sometimes include follow-up interviews years or decades after an initial study, but such projects cannot span the centuries that corpus studies can. Unfortunately, corpora are analogous to fossils or archaeological discoveries in that present-day scientists have no control over the content of ancient documents, or which documents survive to be included in a corpus. Furthermore, the written record contains plenty of linguistic information, but the written language is often distinct from the spoken language in ways that cannot be confirmed centuries later.

The data set of interest for this project is from the change in late Middle English syntax from verb-raising to *do*-support [13]. In verb-raising syntax, main verbs are raised from a low position in the syntax tree to various high positions. This means that the main verb raises above the subject, yielding inverted questions, and the main verb raises above negation, so it appears before *not*:

(13.1) Know you what time it is?

(13.2) I know not what time it is.

In *do*-support syntax, the main verb is restricted to a low position in the syntax tree, so when a verb is needed to fill a high position, the auxiliary verb *do* must be inserted:

(13.3) Do you know what time it is?

(13.4) I don't know what time it is.

Affirmative declarative statements have the same surface form under both grammars; the insertion of *do* in this case is actually forbidden under the *do*-support grammar:

(13.5) I know what time it is.

(13.6) *I do know what time it is.

The * indicates an ungrammatical utterance. The second example can be made grammatical by stressing the *do*, which changes the meaning to indicate that the speaker is contradicting a previously made statement. Without that stress, the *do* is ungrammatical. Oddly, the insertion of *do* is also ungrammatical for affirmative subject questions without stress:

(13.7) Who knows what time it is?

(13.8) *Who does know what time it is?

A previous study of the *do*-support data by Kroch [24] fit a logistic sigmoid

$$y = \frac{1}{1 + e^{-a(t-t_{1/2})}} \tag{13.9}$$

to the S-curve of the usage rate y of *do*-support over time t. The notation $t_{1/2}$ refers to the fact that when $t = t_{1/2}$, $y = 1/2$. Such a function may be grounded in a logistic population model where the rate of spread of a feature is jointly proportional to the fraction of people who have it and the fraction who do not, as in the logistic differential equation

$$\frac{dy}{dt} = ay(1-y)$$

to which the function (13.9) is the general solution. The logistic model assumes an infinite, unstructured, homogeneous population. The point of the calculations in [24] was to infer the rate constant a and demonstrate that the rise of *do*-support in all different kinds of sentences was governed by the same rate constant, although the time offsets $t_{1/2}$ differ. Kroch names this result the *constant rate effect*. Infelicities of the logistic model, such as the fact that it admits populations with a fractional number of people, were not important, nor was the overall population size.

In the interest of inferring a population size from the *do*-support data, we turn to mathematical population genetics. One of the simplest and most flexible tools for working with finite populations is the Moran model [40]. The population consists of a finite number of agents, each of which is in one of a fixed number of states. An agent is removed to simulate death, and a new one is created by cloning a randomly selected agent thereby simulating birth. Since births and deaths are paired, the overall population size is fixed.

The mathematical framework at work is the discrete time, finite state space Markov chain: At each time step, the population is in one of a large but finite number of possible states. The dynamics specify that given the current state, the population changes randomly to a new state at the next time step, but the probability of selecting each possible new state is a deterministic function of its current state. It is possible, though often computationally infeasible, to use a vector to represent the probability that the population is in each possible state at a given time step. A stochastic matrix can be used to represent the transition process, and multiplying the distribution vector by the transition matrix gives the distribution vector for the next time step. A computer program with a random number generator can implement the transition process and output a stream of states, that is, a sample trajectory of the model.

The variable influence model in this project starts with the Moran model, but assumes that individuals have different degrees of influence on the speech of others. The cloning step is therefore modified to take influence into account. Initially, most of the agents are in a state representing the old language. A few influential agents start in a different state representing the new language. New agents are more likely to be cloned from the more influential agent, so the new language will spread and is likely to take over. The state of each individual agent must be recorded, which makes for much more complex calculations compared to the original Moran model and the logistic model, in each of which the population state is a single number. The main difficulty is that there are so many possible population states that the vector-and-matrix representation of the Markov chain is computationally infeasible. Instead, the only way to investigate its behavior is to accumulate many sample trajectories and take some sort of average. Therefore, several simplifying assumptions are necessary to formulate a model for which the calculations are feasible. A further complication is determining when enough samples have been collected, so the analysis of the samples will be done two different ways. One is a straightforward average and the other estimates the same average from an approximate density. Based on samples collected over a year of computer time, the two calculations agree, which suggests that we have enough samples. So, despite the numerical difficulties, it is possible to fit the model to the *do*-support data. The result is an estimate of an influence ratio that indicates the extent to which influence is concentrated in a few individuals.

In the rest of this chapter, we formulate the variable influence model, then test a range of values of the influence ratio to determine which is most harmonious with the Middle English data. The calculations strongly support the conclusion that the population is distinctly skewed, specifically, that a leadership core of around 19 individuals out of the 200 in the simulation account for 95% of the total influence.

13.2 The Available Data

The available data consists of counts of sentence types from clusters of Middle and Modern English manuscripts and the approximate dates of those clusters [13, 24]. The sentences of interest are different kinds of questions and negative statements, as these clearly show whether the speaker is generating them with a verb-raising grammar or a *do*-support grammar.

Do-support replaced verb-raising in several stages, affecting some types of sentence before others. The cleanest data is for transitive affirmative questions, as in 'Do you want sugar?' and this subset of the data will be the focus of the remainder of this chapter. See Figure 13.1 for a graph of this data.

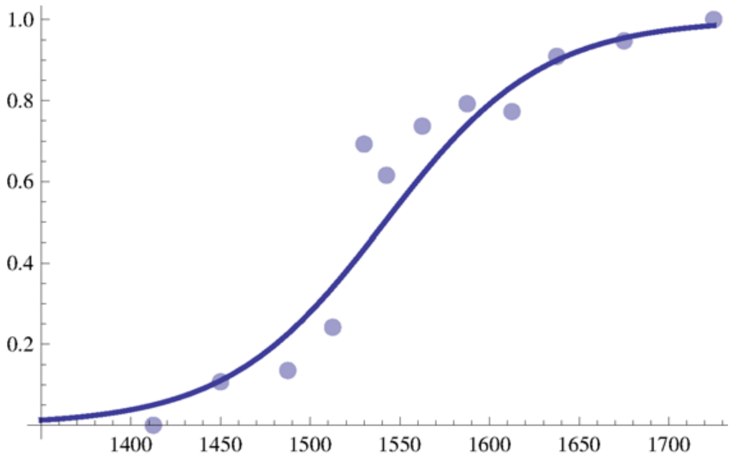


Fig. 13.1 Occurrence rate of *do*-support as a fraction of total sentences, for transitive affirmative questions. The curve is a logistic sigmoid fit to this data via maximum likelihood.

13.3 Formulation of the Variable Influence Model

The simulated population consists of n individual agents, with $Y_i(t)$ representing the type of the i-th agent at time step t, $i \in \{0, 1, \ldots, n-1\}$. We fix $n = 200$. True speech shows context- and individual-dependent variation, but at this stage of the modeling process, a simplification is computationally necessary. We therefore make the simplifying assumption that there are 2 relevant language variants in use, numbered 0 and 1, representing the old verb-raising and new *do*-support grammars, respectively. Thus $Y_i(t) = k$ means that at time t, agent i uses variant k exclusively.

The Markov chain has 2^n possible states because the type of each individual must be tracked separately. This means that if the simulated population is at all large, even 30 individuals, the transition matrix will be too large to compute with directly.

Let $X_k(t)$ be the number of individuals of type k at time t. Thus, if a speaker is selected uniformly at random and asked to produce a sentence, then the sentence is generated by language variant k with probability

$$S_k(t) = \frac{X_k(t)}{n} \tag{13.10}$$

and for each t, $S_0(t) + S_1(t) = 1$.

Let Δt be the real time associated with a unit change it t. The transition function from time step t to $t + 1$ involves examining each agent. With probability $\beta\Delta t$, the agent is replaced, otherwise it remains unchanged, that is $Y_i(t+1) = Y_i(t)$. To replace it, another agent is selected at random and its type is used as $Y_i(t + 1)$. This operation simulates the birth of a new individual who chooses a language variant based on the speech of a single adult. For the calculations in this chapter, Δt is taken to be one year, and $\beta = 1/40$ so that each agent survives for a geometrically distributed random lifetime with a mean of 40 years.

To model a population in which all individuals have equal influence on learning, we would choose the agent to copy in the replacement step uniformly at random from among the whole population. For variable influence, we can assign a score to the i-th slot in the population and choose the agent to copy with probability proportional to that influence score. We will use the function b^i, where b is the *influence ratio*, $0 < b \leq 1$. That is, each individual is a factor b less influential than the next most influential individual. If b is close to 1, then influence is spread through a large part of the population, but if b is even a bit less than 1 then influence is concentrated in a few individuals.

In the initial state, most agents are in state 0 to indicate that the population was dominated by verb-raising initially, but a few agents are in state 1 to trigger the transition to *do*-support. For this chapter, the initial state is that the four most influential agents are in state 1 and the rest are in state 0. The initial time is interpreted as the year 1410, which is about the time of the first data point in the corpus. In the long run, the population will end up in one of two absorbing states, either all state 0 or all state 1. Historically, the English converged to all state 1.

Some sample trajectories are displayed in Figures 13.2 to 13.4. They were hand selected from a small random sample to illustrate the impact of b, and exclude trajectories in which the new language went extinct. For smaller values of b as in Figure 13.2, the usage rate of the new language increases too quickly and overshoots the data. For larger values of b as in Figure 13.4, the population is more influentially uniform, and the trajectory behaves more like a symmetric random walk, almost as likely to go down as up. An intermediate value as in Figure 13.3 fits the data better.

The average shapes of trajectories are shown in Figures 13.5 to 13.7. These display quartiles of samples of many trajectories, and approximately indicate the distribution of the population as a function of time for several different values of b.

To understand why the trajectories have the shape that they do, consider first the beginning of the change, where only a few of the most influential agents are in state 1. Each time step replaces approximately βn agents, and many of the new ones will be clones of influential agents and therefore type 1. This yields an approximately linear growth in the usage rate of the new grammar, but slightly concave-up. The curvature happens because as more influential agents are replaced, the fraction of new agents of type 1 each step will increase. It is often very slight and does not match the distinct initial upward curve of the usual sigmoid trajectory of language changes, which suggests that some modifications should be made to the model in future experiments.

The downward curve at the top of the simulated trajectories is at least qualitatively in agreement with the downward curve typical of language changes. This curvature happens because once most of the population has switched to type 1, a large fraction of the agents that get replaced were already type 1, so the net change of the usage rate of the new grammar is slower.

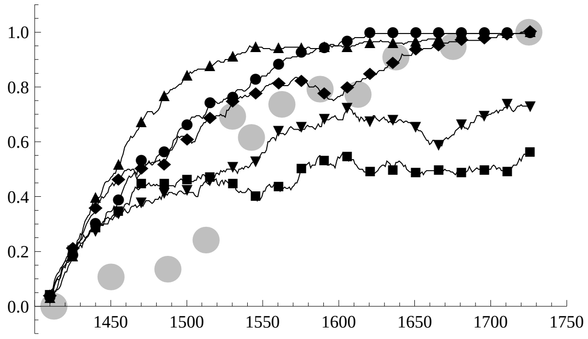


Fig. 13.2 Trajectories, shown as the fraction $S_1(t)$ of the population in state 1 as a function of time. For these runs, $b = 0.8$. Different runs are marked by different symbols. Big gray dots mark the *do*-support usage rate from the corpus.

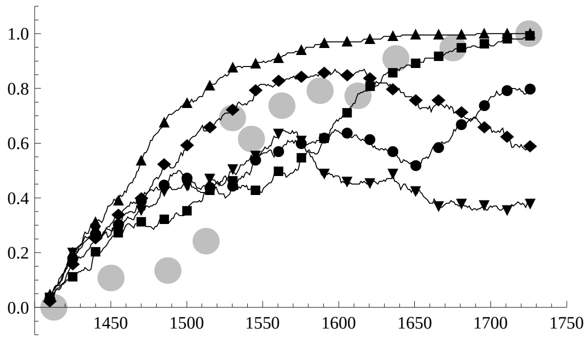


Fig. 13.3 Trajectories, shown as the fraction $S_1(t)$ of the population in state 1 as a function of time, for $b = 0.85$.

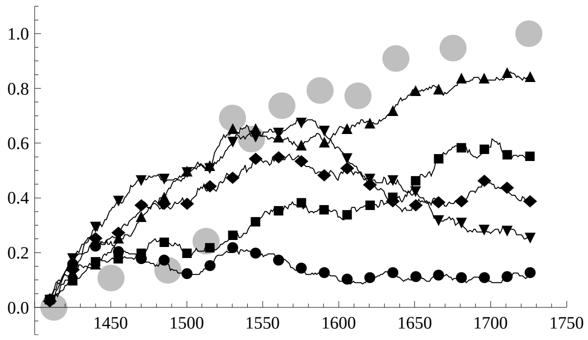


Fig. 13.4 Trajectories, shown as the fraction $S_1(t)$ of the population in state 1 as a function of time, for $b = 0.9$.

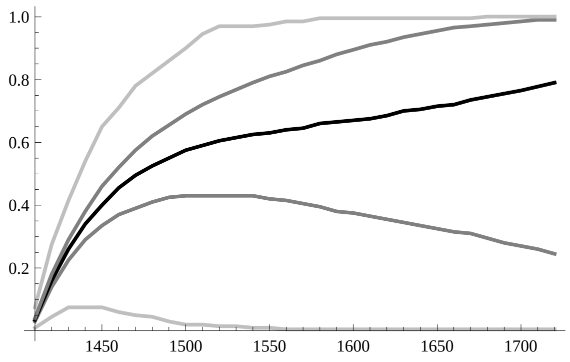


Fig. 13.5 An ensemble envelope of $S_1(t)$ as a function of time. The curves show the minimum, first quartile, median, third quartile, and maximum at each time step over 5000 sample trajectories, for $b = 0.8$.

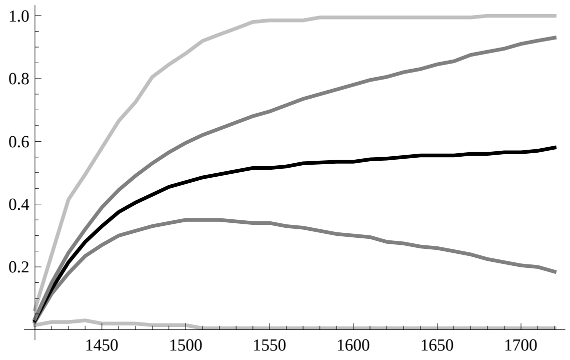


Fig. 13.6 An ensemble envelope of $S_1(t)$ as a function of time, for $b = 0.85$.

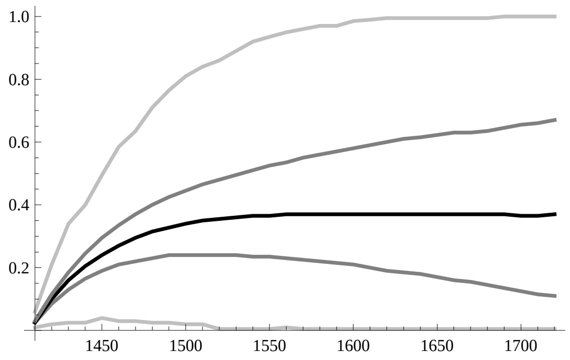


Fig. 13.7 An ensemble envelope of $S_1(t)$ as a function of time, for $b = 0.9$.

13.4 Fitting the *Do*-support Data

13.4.1 *The Maximum Likelihood Method*

To represent the data, let $t_1, t_2, \ldots$ be the times at which clusters of manuscripts are available, and define $m_k(j)$ be the number of sentences of type k found in the manuscripts at time t_j.

Any model of linguistic dynamics can be tuned to the manuscript count data through maximum likelihood. The idea comes from Bayesian inference [15]. We are really interested in the value of a parameter b. Probability theory is a mathematical way to express partial information about unknown quantities. So we will treat B as a random variable for the parameter b, and our confidence that B has a particular value is represented by a probability distribution $\mathbb{P}\,(B \in db) = \mathrm{p}\,(b)\,db$. Until we obtain data, our information about B is some *prior distribution* that incorporates any assumptions we might need to bring to the model. We assume $0 \leq B \leq 1$ but any value in this range is equally likely, so the prior is the uninformative uniform distribution on the interval $[0,1]$, that is $\mathrm{p}\,(b) = \mathbf{1}\,(0 \leq b \leq 1)$.

The addition of data, also treated as a random variable, causes us to have more confidence in some values than others. This modified knowledge is represented by a *posterior distribution* $\mathrm{p}\,(b \mid m)$.[4] Bayes's formula gives

$$\mathrm{p}\,(b \mid m) = \frac{\mathrm{p}\,(m \mid b)\,\mathrm{p}\,(b)}{\mathrm{p}\,(m)}$$

The distribution $\mathrm{p}\,(m \mid b)$ is called the *likelihood*, meaning the probability of observing the data m given a particular value of the b. Since we are taking the prior distribution to be the uniform distribution, and since $\mathrm{p}\,(m)$ is the same no matter what b is, we can treat these as unknown constants and use the form

$$\mathrm{p}\,(b \mid m) \propto \mathrm{p}\,(m \mid b)\,.$$

The obvious choice of b is one in which we have high confidence after examining the data, that is, a b for which the posterior is high. Since the posterior differs from the likelihood by a constant factor, we can choose b to be the value that maximizes the likelihood. The point of this is that the likelihood can be computed based on the model of which b is a parameter.

[4]In classical frequentist statistics, one might assume that B has a normal distribution with mean μ and variance σ^2 and express this information as a confidence interval. That is, $\mathrm{p}\,(b \mid m) \propto e^{-(b-\mu)^2/\sigma^2}$. The Bayesian approach allows the posterior to be more general.

By computing it for many values of b, one can then sketch the posterior distribution up to a scale factor and select the maximum.

Given estimates $s_k(t)$ of the population-wide usage rate of variant k at time t, the likelihood comes from the binomial distribution

$$\mathrm{p}\,(m \mid s) = \prod_j \binom{m_0(j) + m_1(j)}{m_1(j)} s_0(t_j)^{m_0(j)} s_1(t_j)^{m_1(j)} \tag{13.11}$$

A straightforward calculation gives an upper bound on the likelihood of the transitive affirmative *do*-support data. For each time point t_j, there is a value of $\hat{s}_j$ that maximizes

$$\hat{s}_j^{m_0(j)} (1 - \hat{s}_j)^{m_1(j)}$$

Putting those values of $\hat{s}_j$ in for $s_0(t_j)$, using $s_1(t_j) = 1 - \hat{s}_j$, and taking the product yields the upper bound. It should be noted that a model can only achieve that bound by over-fitting the data. For the transitive affirmative question data, the upper bound is 5.48×10^{-10}. Let ρ be the natural logarithm of this upper bound, so $\rho = -21.3248$. For reference, the likelihood achieved by the logistic curve in Figure 13.1 is 2.04×10^{-17}.

Logistic dynamics yield a fairly simple explicit formula for the likelihood in terms of two unknown parameters. The curve in Figure 13.1 was drawn by assuming the form $s_1(t) = 1/(1 + \exp(-a(t - t_{1/2})))$ and solving for a and $t_{1/2}$.

In contrast, it is not possible to use an explicit formula for the likelihood with the variable influence Markov chain. Instead, the likelihood must be computed by conditioning on the population's complete history. Let $\mathcal{H}$ be the set of all possible histories $y_i(t)$ of the population. If y is given, then the type counts $x_k(t)$ and the overall usage rates $s_k(t)$ are known in terms of y. Thus

$$\begin{aligned} \mathrm{p}\,(m \mid b) &= \sum_{y \in \mathcal{H}} \mathrm{p}\,(m \mid s)\,\mathrm{p}\,(y \mid b) \\ &= \mathbb{E}\,(\mathrm{p}\,(m \mid S)) \end{aligned} \tag{13.12}$$

Unfortunately, the summation over $\mathcal{H}$ is computationally infeasible. For any reasonable population size, such as the modest $n = 200$ used in this project, there are too many possible histories. A Monte Carlo method that averages over a random sample $\mathcal{S}(b)$ of possible histories generated with a particular value of b is the obvious alternative:

$$\mathrm{p}\,(m \mid b) \approx \frac{1}{|\mathcal{S}(b)|} \sum_{y \in \mathcal{S}(b)} \mathrm{p}\,(m \mid s) \tag{13.13}$$

An important property of (13.11) is that if either language variant goes extinct too early in the trajectory, then for some j, $s_1(t_j)$ will be one or zero, thereby zeroing out the entire product. The syntactic change is known to have taken place, and the old language persisted for some time. Therefore any sample trajectory in which that change is impossible will be discarded, that is, we condition on the fact that the Markov chain must be absorbed into the state of all 1s but not before the old syntax disappears from the written record.

It should be noted that this calculation is different from what is normally meant by the terms *Markov chain Monte Carlo*, in which the goal is to concoct a Markov chain whose stationary distribution matches some desired distribution and to then sample from it. Rather, for this model the Markov chain itself is the random process of primary interest. We are not interested in a stationary distribution but in trajectories themselves, starting from a particular starting point and moving toward absorption.

13.4.2 *The Monte Carlo Calculation*

Although the average (13.13) is computationally feasible, it turns out to require a huge sample size to achieve acceptable results. The core difficulty is that the trajectories y for which $\mathrm{p}(m \mid s)$ are largest are relatively uncommon, but the corresponding values of $\mathrm{p}(m \mid s)$ are several orders of magnitude larger than the likelihood values contributed by bulk of the samples. In other words, the average (13.13) is dominated by rare events. Figure 13.8 shows a histogram of $\ln \mathrm{p}(m \mid S)$ for $500,000$ samples using $b = 0.85$. That is, the horizontal scale is logarithmic. For reference, the smallest positive number representable in the standard 64-bit floating point format is about 2×10^{-308} or about e^{-708}. To avoid hardware underflow errors, the *logarithm* of the likelihood has to be computed all along. An important reason to condition on the fact that the change took place is that none of the likelihood samples can be zero, for which the logarithm would be undefined.

The author wrote a computer program and ran it sporadically on a shared computer cluster over the course of a year to generate $170,000,000$ samples of the log-likelihood for each of 12 different values of b. On this cluster, the program takes approximately 9.5 hours to produce 500,000 sample runs for each of the 12 values of b, which means that the full data set required about 3200 hours or about 134 days of cluster computing time. These samples were then processed in several different ways as described in the following subsections.

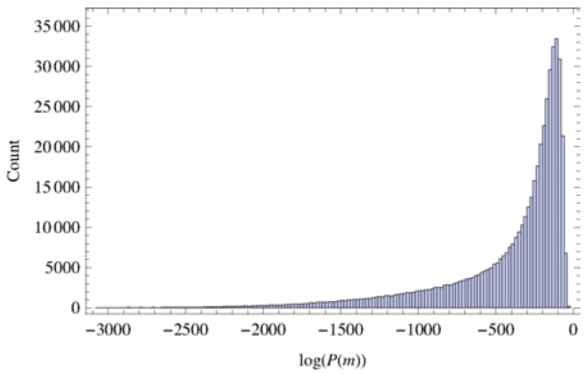


Fig. 13.8 Histogram of 500,000 samples of $\ln \mathbb{P}(m \mid S)$ with $b = 0.85$.

13.4.3 *Direct Average*

The obvious approximation to the likelihood is a direct average of the samples as in (13.13). The safest way to compute it is to use a program like Maple or Mathematica to read the log-likelihood samples, apply $\exp(\cdot)$ using arbitrary precision arithmetic rather than hardware floating point arithmetic, and take the average. Likelihood ought to be a smooth function of b, but it takes many millions of samples to produce a reasonably clean plot. The results are shown in Figure 13.9. As is typical of Monte Carlo methods, the accuracy of the result depends on the square-root of the sample size, so the error bars are fairly large even with $170,000,000$ samples. Nevertheless, the likelihood increases dramatically as b decreases from 1 (evenly distributed influence) to $b = 0.85$, which indicates that influence is concentrated in a few members of the population.

The confidence intervals shown in Figure 13.9 are drawn by computing a sample standard deviation $\bar{s}$ for the set of likelihood samples, then plotting $\pm 2\bar{s}/\sqrt{|\mathcal{S}|}$, assuming there is sufficient data to invoke the central limit theorem. Interpreting the confidence intervals, there is enough data to assert that the maximum is at no more than 0.9 and most likely at 0.85

13.4.4 *Fitting the Density*

It is useful to process the samples a second way to confirm that enough data has been collected. An alternative to the raw average is to fit a curve to the log-likelihood histogram and use an integral to compute the

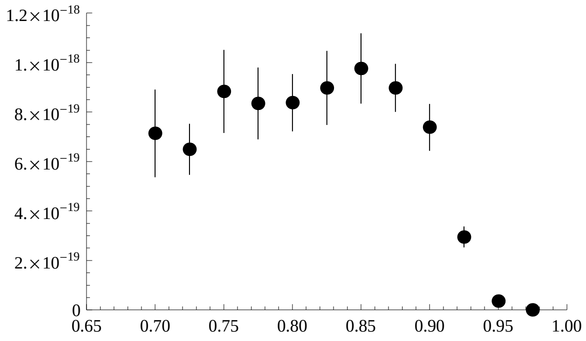


Fig. 13.9 Estimates of $\mathrm{p}\,(m \mid b)$ from a direct average. Whiskers indicate a 95% confidence interval.

likelihood. To make this process numerically simpler, the log-likelihood samples are transformed by subtracting them from the log-upper-bound ρ, thereby yielding all positive values that can theoretically go all the way down to 0. That is, given a random history Y and the corresponding X and S, set $Z = \rho - \ln \mathrm{p}\,(m \mid S)$. A histogram of Z can be obtained from Figure 13.8 by reflecting it about the vertical axis and shifting it horizontally. Let $\mathrm{p}\,(z \mid b)$ be the density function for Z and let $\bar{f}(z; b)$ be an estimate of $\mathrm{p}\,(z \mid b)$ obtained through a curve fit. Then

$$\begin{aligned} \mathrm{p}\,(m \mid b) &= \mathbb{E}\,(\exp Z) \\ &= \int_0^\infty e^{\rho - z}\,\mathrm{p}\,(z \mid b)\,dz \\ &\approx \int_0^\infty e^{\rho - z}\bar{f}(z; b) dz \end{aligned} \tag{13.14}$$

This method does not escape from all the difficulties of the direct average method. The integral is very sensitive to the density near $z = 0$ because that is where most of e^{-z} is concentrated, but that is precisely where there is the least data and the most uncertainty. The curve fit effectively smooths the histogram in that area.

The algorithm used to fit the density for each value of b is as follows. Sample histories are transformed into samples for Z, which are then grouped into bins of width 1. Empty bins are discarded. Each bin B_n contains numbers $z_1, z_2, \ldots \in [n, n+1)$ which are mapped to the point

$$(u_n, v_n) = \left(\frac{1}{|B_n|} \sum_i z_i, \frac{|B_n|}{|\mathcal{S}|} \right) \tag{13.15}$$

where $|\mathcal{S}|$ is the total number of samples in all the bins. The left-most bins usually contain fewer points than the others because high likelihood estimates are rare. The average of the numbers in B_n is used for the horizontal coordinate of the point rather than the midpoint of the bin because it better reflects the off-center numbers in those left-most bins, which yields more stable likelihood estimates.

Although the shape of the density of Z consists of a smooth hill and a long tail, it does not seem to be well represented by the commonly occurring gamma or extreme-value distributions. Instead, a fairly general form was chosen for the fit function based on trial and error and asymptotic considerations. A polynomial $a_0+a_1\lambda+a_2\lambda^2$ is fit to the log-log points $(\ln u_n, \ln v_n)$ with $u_n \leq 40$ using the method of least squares, with each point weighted by the number of samples in the corresponding bin B_n. The transformation to Z ensures that all the u_n are positive so $\ln u_n$ is defined. The weighting is important because it continues the trend of points not quite at the extreme left where more data is available, while not ignoring the points derived from sparser data at the extreme left. This process yields a fit to a function of the form

$$\begin{aligned}\bar{f}(z) &= e^{a_0+a_1\lambda+a_2\lambda^2} \text{ where } \lambda = \ln z \\ &= c_0 z^{c_1} e^{-c_2(\ln z)^2}\end{aligned} \tag{13.16}$$

with

$$c_0 = e^{a_0} > 0, c_1 = a_1 > 0, c_2 = -a_2 > 0,$$

This form takes advantage of the fact that the graph of $(\ln u_n, \ln v_n)$ is fairly smooth, and that we need only fit the left side of the hill. It does not match the tail of the density, but it does not need to. Only the shape of the density for small z matters. See Figure 13.10.

A seemingly better fit to the log-log points can be found by including higher powers of λ. However, for some b values, doing so yields negative coefficients on the $(\ln z)^3$ terms and therefore a singularity as $z \to 0$. The correct asymptotic behavior at 0 requires that the coefficients on $(\ln z)^2$ be negative and those on $(\ln z)^3$ be positive. Then, as $z \to 0$, $-(\ln z)^2 \to -\infty$ and $(\ln z)^3 \to -\infty$, so $\bar{f}(z) \to 0$.

Given those approximate densities, the integral approximations of the likelihoods are shown in Figure 13.12. The fitting procedure gives 95% confidence intervals for the parameters, which are mapped into confidence intervals for the likelihood estimates. The results are essentially the same as from using the raw average, as in Figure 13.9, with the maximum at $b = 0.85$.

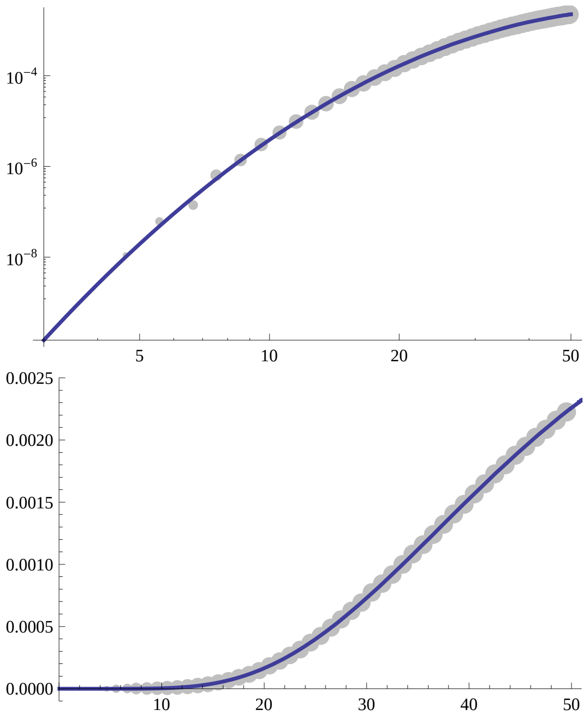


Fig. 13.10 Top: Log-log plot of (u_n, v_n) and $\bar{f}$ fit to the points with $u_n \leq 40$, for $b = 0.85$. Bottom: Same functions on normal scale. Areas of dots are proportional to $\ln(|B_n| + 1)$.

This process is particularly sensitive to the value of ρ. The calculations were carried out once with an incorrect ρ, which resulted in a very noisy likelihood graph with much larger error bars. Therefore, the form (13.16) is probably not optimal. However, the error bars with the correct ρ are very small, and the overall shape agrees well with the likelihood estimates from the direct average.

13.5 Results and Discussion

To begin, we should compare the likelihood estimates from the two methods. See Figure 13.13. Both methods indicate that the maximum satisfies $0.8 \leq b \leq 0.85$. They are largely in agreement, suggesting that $170,000,000$ is nearly enough samples to estimate $\mathrm{p}(m \mid b)$ across the range of interest.

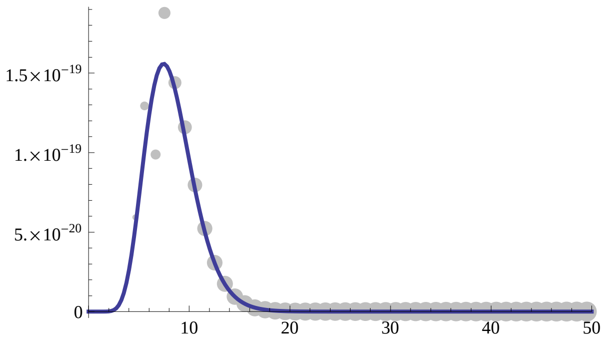


Fig. 13.11 Plot of $(u_n, e^{\rho - u_n} v_n)$ and $e^{\rho - z}\bar{f}(z)$ for $b = 0.85$.

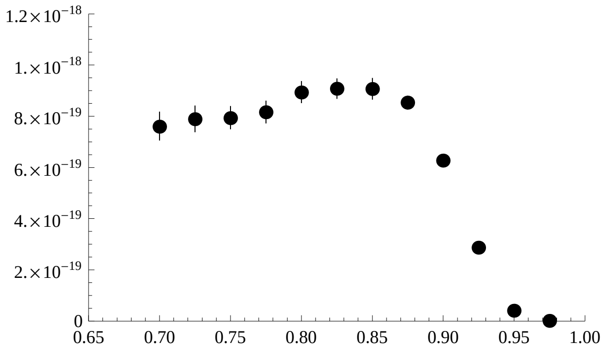


Fig. 13.12 Estimates of $\mathrm{p}\,(m \mid b)$ from integrating against a fitted density. Whiskers indicate a 95% confidence interval.

Let us be overly conservative and consider $b = 0.9$. Recall that within the simulation, the influence score of individual i is b^i. The ratio of the net influence of the first m individuals to the total influence across the population indicates the degree to which influence is concentrated among the most influential individuals. This ratio is plotted in Figure 13.14. The 29 most influential individuals account for 95% of the total influence for $b = 0.9$. For $b = 0.875$, the 23 most influential account for 95%. The b that maximizes the likelihood appears to be at most $b = 0.85$, for which the 19 most influential individuals account for 95%.

Even though more samples would help to pin down the correct value of b, the collected data is definitely more consistent with a variable-influence population than a flat population: A leadership core of approximately 19

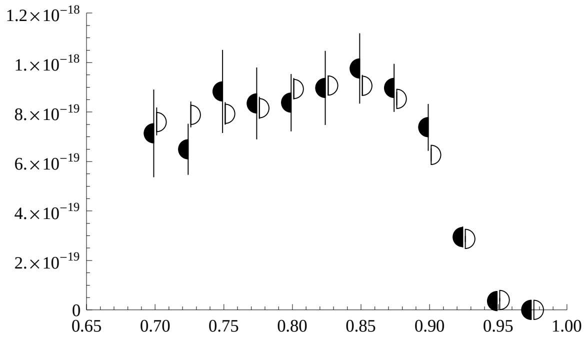


Fig. 13.13 Estimates of $\mathrm{p}(m \mid b)$. For each value of b, the black half dot is the estimate from the direct average and the white half dot is the estimate from integrating against a fitted density. Whiskers indicate a 95% confidence interval.

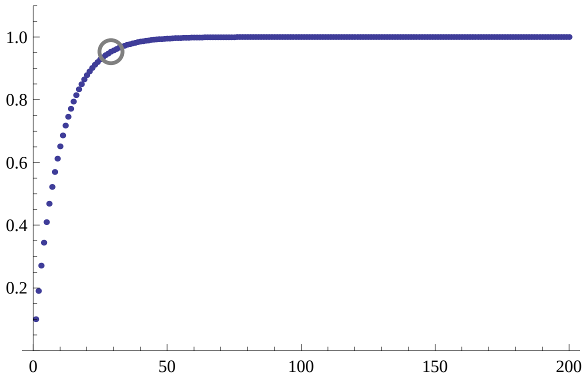


Fig. 13.14 Ratio of the net influence of the first h individuals to the total influence across the population, as a function of h, for $b = 0.9$. The gray circle is centered at $h = 29$, which accounts for 95%.

people account for most of the total influence. This suggests that if it were possible to survey a large number of people and somehow determine who was most influential on each of their speech patterns, we should expect 19 or so linguistic leaders for each community within the overall population.

13.6 Future Directions

This project focuses on inferring a single macroscopic feature of medieval English society from data about syntax. However, the results suggest many improvements, further questions, and other applications of this mathematical modeling technique.

There is plenty of room for improvement in the variable influence model, but at a cost. Many of its features as presented here are fixed at some reasonable value so that b is the only parameter that has to be inferred. The inescapable difficulty is that adding detail will add variables, which must then either be set arbitrarily, like the choice of $\beta = 1/40$, or inferred from samples of the Markov chain. There is no reason to suppose that the unknown variables can be inferred independently in seeking to maximize the likelihood. That is, if we seek to maximize the likelihood allowing some new unknown c to vary, the maximum might occur at some $b_{\max}$ and $c_{\max}$ where $b_{\max}$ is not 0.85 as found here (although it ought to be close). To maximize the likelihood will require many samples at a much larger set of values of (b, c), which will require a lot of computer time. Likelihood provides a metric for how important a particular variable is to the model. If adding the variable increases the likelihood significantly without overfitting the data, then it is important and the increase in likelihood quantifies how much. Otherwise, it can probably be omitted, and computer time can be better spent on some other feature. On a practical note, the sampling program could be written more efficiently, but any improvements will probably not be sufficient to allow for inferring more than two or three interdependent variables in a reasonable amount of time. New mathematical tools for dealing with Markov chains such as this model are therefore needed.

To give a specific comparison, the logistic sigmoid in Figure 13.1 fits the *do*-support data better than the variable influence model, in that the likelihood of the data given the sigmoid model (2.04×10^{-17}) is higher than the likelihood given the variable influence model with the best choice of b (10^{-18}). This difference is to be expected because the logistic model has two variables, a and $t_{1/2}$, but the variable influence model has only one, b. The difference should not be interpreted as a failure of the variable influence model, because the two models give different information. Specifically, the logistic model does not give any information about heterogeneity of the population. Based on Figure 13.3, the main drawback to the variable influence model is that its trajectories do not match the slow growth of the change near its beginning, so refinement should focus first on this feature to increase the likelihood.

With these concerns in mind, the question naturally arises of whether the extra effort required by models of this kind is worthwhile. Alternatively, one could try to run analyses based around a series of hypothesis tests. For example, we might consider as a null hypothesis that children learn from all adults in their neighborhood equally, then ask whether a certain data set allows us to reject this hypothesis at some confidence level. This approach has the advantage of being computational easier, and the statistics are potentially conventional and well understood. However, such a project gives no indication of the degree to which individuals with varying influence might drive language change. A hypothesis test may give the result that certain models are statistically distinct, but it gives no information about what the difference means, or whether it might be statistically significant but subordinate to some other stronger force. A VARBRUL-based analysis might initially seem to be a reasonable alternative. It gives relative strengths of various factors on the probability of an overall binary outcome, but it can only be used if the factors in question are also binary. Estimating the size of a leadership core is not possible with VARBRUL, for example.

In contrast, the variable influence model includes a continuous parameter b that indicates the strength of the effect. In the calculations, it ranges from no effect at $b = 1$ to concentrating influence in 19 individuals at $b = 0.85$. The likelihood calculation allows us to estimate how probable each value of b is given the data, so we are essentially testing a whole range of hypotheses rather than two as in a standard hypothesis test.

An obvious improvement would be a more realistic representation of language usage and the learning process. The variable influence model currently assumes that children exactly copy one other individual's speech, when they should learn from several, including adults and peers. Furthermore, an individual's state should include more possibilities than using one language variant or another exclusively. Recognizing that the discrete categorial tools of formal grammars and idealized speakers are insufficient for representing the intricate variations of language, there is increasing interest in using probability in conjunction with traditional formalisms to understand and represent language [2, 52, 31]. These features could be incorporated into the current model and would likely be worth the computational cost.

An additional improvement would be in the interpretation of the manuscript data. The likelihood formula (13.11) implicitly models the creation of the corpus by selecting individuals uniformly at random and asking for a sentence, which is rather naïve. The corpus contains collections of manuscripts written at estimated times by relatively few speakers in a variety of genres, and these are the ones that happen to have survived the centuries and been cataloged by linguists. There is clearly room for an improved model of corpus formation, but it, too, would introduce additional parameters that must be fixed or inferred.

The present model does not try to account for how leaders arise. Studies reported in [27, 26] suggest that leadership in language change is determined more by personality, attitude, gender, and friendship networks than any conventional notion of economic or political power. Furthermore, is not clear how to properly scale the results of Section 13.5 to larger populations. Simply increasing the population size n is not likely to affect b significantly because the less influential bulk of the population effectively copies the proportions of the leadership core. Larger populations will have more complex social structure, in which some people are very influential but over distinct subsets of the overall population. The ordered influence structure used here was simple and gave reasonable results, but it would be more realistic to represent the population as graph. Agents would be vertices, edges would indicate linguistic influence, and new agents could be incorporated through some form of preferential attachment process.[5] For example, each new agent might be linked or not to each existing agent with probability determined by the number of links the existing agent already has. Another alternative would be to retain the flat structure but use a function other than b^i for the influence of the i-th individual. However, each of these potential improvements might make it more computationally demanding to fit the model to the corpus data.

Labov and his collaborators have accumulated considerable data on phonetic changes in cities, including information about specific informants who seem to be leading these changes. It should be possible to fit the variable influence model to that phonetic data, in which case its conclusions about leadership structure can be compared to the collected sociological information. Such a project would provide an additional means of verifying this method of statistical analysis.

The model developed in this chapter began as a population genetics model, although the application is sociological and no explicit use is made of natural selection. However, it should be possible to adapt the variable influence model for use in studying biological evolution. Consider for example a model of the evolution of imitation posed by Boyd and Richerson [3]. Their underlying model was a set of individual agents who choose a behavior based either on their observation of the environment, or by copying a randomly selected individual when their observation is inconclu-

[5]See [1, 54] and forthcoming articles by Swarup.

sive. The mathematics is greatly simplified by assuming that all agents are essentially interchangeable, and by focusing on the dynamics of average properties of the population. Leadership structure breaks the assumption of interchangeability, and may rule out the reduction of the dynamics to average properties. It would be informative to revisit the Boyd-Richerson model with leadership structure added in, and see which of their results still hold and which are modified.

13.7 Conclusion

In conclusion, corpus data concerning the rise of *do*-support at the expense of verb-raising, in conjunction with an agent-based population model, is consistent with the hypothesis that influence is distributed unevenly through the population. The maximum likelihood method, computed two different ways from sample runs of the variable influence Markov chain model, yields an estimate of the influence ratio. That estimate asserts that approximately 19 out of the 200 people within the simulation account for 95% of the total linguistic influence. This project provides an important mathematical tool for combining sociolinguistics with historical methods and sophisticated mathematical models, but there is plenty of room for improvement.

The author gratefully acknowledges that this project was supported by a grant from the National Science Foundation (DMS #0734783).

Appendix: Probability and Notation

The notation for probability distributions can be confusing, particularly when mixing continuous and discrete distributions and when conditional probability is involved. I provide this appendix to assist readers who may not be as familiar with some of the concepts and my preferred notation.

Whenever possible, a capital letter (as in X) is used for a random variable and the corresponding lower case letter (as in x) is used for a non-random value that it might take. For instance, the density for X would be written in terms of x, and a calculation involving random samples would be written in terms of X.

The notation

$$\mathbb{P}\left(X \in dx\right) = f(x)dx$$

indicates that X has a *continuous distribution* with *probability density function* f, so that

$$\mathbb{P}(a < X < b) = \int_a^b f(x)dx$$

For a random variable N with a *discrete distribution*, we can write

$$\mathbb{P}(N = n) = g(n)$$

to indicate that g is the *probability mass function* for N. A mass function can also be expressed using delta measures. The symbol $\delta_z(x)$ is special notation for using an integral $\int$ to pick out discrete values of a function:

$$\int \phi(x)\delta_z(x)dx = \phi(z)$$

so the mass function for N can also be expressed as

$$\mathbb{P}(N \in dx) = \sum_n g(x)\delta_n(x)dx.$$

The bar notation indicates conditioning. That is, $X \mid Y$, read "X given Y," means that we modify the distribution of X by assuming Y is known. The basic property of conditioning is that

$$\mathbb{P}(X \in dx \text{ and } Y \in dy) = \mathbb{P}(X \in dx \mid Y = y)\,\mathbb{P}(Y \in dy).$$

Since you can also condition on X,

$$\mathbb{P}(X \in dx \text{ and } Y \in dy) = \mathbb{P}(Y \in dy \mid X = x)\,\mathbb{P}(X \in dx).$$

Combining the two gives Bayes's formula,

$$\mathbb{P}(X \in dx \mid Y = y) = \frac{\mathbb{P}(Y \in dy \mid X = x)\,\mathbb{P}(X \in dx)}{\mathbb{P}(Y \in dy)}$$

which expresses the distribution of X given Y in terms of the distribution of Y given X. (The dy's effectively cancel; the details involve the Radon-Nikodym derivative and are well beyond the scope of this chapter.)

Since it's usually more convenient to work with densities (or to use delta measures to pretend that discrete distributions have densities), the notation $\mathrm{p}(x)$ is often used to indicate the density of the random variable X tied by convention to the same letter in lower case,

$$\mathbb{P}(X \in dx) = \mathrm{p}(x)\,dx.$$

Conditional densities are expressed as

$$\mathbb{P}(X \in dx \mid Y = y) = \mathrm{p}(x \mid y)\,dx.$$

The conditioning formula becomes

$$\mathrm{p}\,(x, y) = \mathrm{p}\,(x \mid y)\,\mathrm{p}\,(y)$$

and Bayes's formula becomes

$$\mathrm{p}\,(x \mid y) = \frac{\mathrm{p}\,(y \mid x)\,\mathrm{p}\,(x)}{\mathrm{p}\,(y)}.$$

Since the $\mathrm{p}\,(\cdot)$ notation is compact and expressive, it is normally overloaded to indicate the mass function of a discrete random variable as well, as in $\mathrm{p}\,(n)$. The interested reader must use context to determine the correct rigorous interpretation for $\mathrm{p}\,(\cdot)$.

The *expected value* or *mean* or *first moment* of a continuous random variable X is

$$\mathbb{E}\,(X) = \int x\,\mathbb{P}\,(X \in dx) = \int x\,\mathrm{p}\,(x)\,dx.$$

For a discrete random variable N, the formula is the same except that the integral becomes a sum:

$$\mathbb{E}\,(N) = \int x\,\mathbb{P}\,(N \in dx) = \int x \sum_n \mathrm{p}\,(n)\,\delta_n(x) dx = \sum_n n\,\mathrm{p}\,(n)\,.$$

The notation for an *indicator function* is

$$\mathbf{1}\,(\text{condition}) = \begin{cases} 1 & \text{if the condition is true} \\ 0 & \text{if the condition is false} \end{cases}$$

References

1. Apolloni, A., Kumar, V. A., Marathe, M. V. and Swarup, S. (2009). Computational epidemiology in a connected world, *Computer* **42**, pp. 83–86, doi:10.1109/MC.2009.386.
2. Bod, R., Hay, J. and Jannedy, S. (eds.) (2003). *Probabilistic Linguistics* (MIT Press, Cambridge, MA).
3. Boyd, R. and Richerson, P. J. (1988). An evolutionary model of social learning: The effects of spatial and temporal variation, in T. R. Zentall and J. Bennett G. Galef (eds.), *Social Learning* (Lawrence Erlbaum Associates, Inc., Hillsdale, New Jersey), pp. 29–48.
4. Briscoe, E. J. (2000). Grammatical acquisition: Inductive bias and coevolution of language and the language acquisition device, *Language* **76**, 2, pp. 245–296.
5. Briscoe, E. J. (ed.) (2002). *Linguistic Evolution through Language Acquisition: Formal and Computational Models* (Cambridge University Press).

6. Cangelosi, A. and Parisi, D. (eds.) (2002). *Simulating the Evolution of Language* (Springer-Verlag).
7. Chomsky, N. (1965). *Aspects of the Theory of Syntax* (MIT Press, Cambridge, MA).
8. Chomsky, N. (1972). *Language and Mind* (Harcourt Brace Jovanovich, New York).
9. Chomsky, N. (1988). *Language and Problems of Knowledge* (MIT Press).
10. Cucker, F., Smale, S. and Zhou, D.-X. (2004). Modeling language evolution, *Foundations of Computational Mathematics* **4**, 3, pp. 315–343.
11. di Sciullo, A. M. (ed.) (2005). *UG and External Systems: Language, brain and computation*, no. 75 in Linguistics Today (John Benjamins).
12. Edwards, C. H. and Penney, D. E. (2008). *Differential Equations and Boundary Value Problems: Computing and modeling*, 4th edn. (Pearson Prentice Hall).
13. Ellegård, A. (1953). *The Auxiliary do: The Establishment and Regulation of Its Use in English*, *Gothenburg Studies in English*, Vol. II (Almqvist and Wiksell).
14. Fong, S. (2005). Computation with probes and goals: A parsing perspective, in [11], pp. 311–333.
15. Gelman, A., Carlin, J. B., Stern, H. S. and Rubin, D. B. (2004). *Bayesian Data Analysis*, 2nd edn. (Chapman & Hall/CRC).
16. Gibson, E. and Wexler, K. (1994). Triggers, *Linguistic Inquiry* **25**, pp. 407–454.
17. Gold, E. M. (1967). Language identification in the limit, *Information and Control* **10**, pp. 447–474.
18. Joshi, A. and Schabes, Y. (1997). Tree-Adjoining grammars, in *Handbook of Formal Languages 3: Beyond Words*, chap. 2, no. 3 in Handbook of Formal Languages (Springer-Verlag), ISBN 978-3-540-60649-9, pp. 69–120, doi:10.1.1.30.502.
19. Kirby, S. (2001). Spontaneous evolution of linguistic structure: an iterated learning model of the emergence of regularity and irregularity, *IEEE Transactions on Evolutionary Computation* **5**, 2, pp. 102–110.
20. Kirby, S. and Hurford, J. R. (2002). The emergence of structure: An overview of the iterated learning model, in [6], pp. 121–148.
21. Komarova, N. L., Niyogi, P. and Nowak, M. A. (2001). The evolutionary dynamics of grammar acquisition, *Journal of Theoretical Biology* **209**, 1, pp. 43–59.
22. Komarova, N. L. and Nowak, M. A. (2001a). The evolutionary dynamics of the lexical matrix, *Bulletin of Mathematical Biology* **63**, 3, pp. 451–485.
23. Komarova, N. L. and Nowak, M. A. (2001b). Natural selection of the critical period for language acquisition, *Proceedings of the Royal Society of London, Series B* **268**, pp. 1189–1196.
24. Kroch, A. (1989). Reflexes of grammar in patterns of language change, *Language Variation and Change* **1**, pp. 199–244.
25. Kroch, A. S. and Joshi, A. K. (1985). The linguistic relevance of tree adjoining grammar, Tech. Rep. MS-CIS-85-16, University of Pennsylvania, URL

`http://repository.upenn.edu/cis_reports/671/`.
26. Labov, W. (1994). *Principles of Linguistic Change: Internal Factors*, Vol. 1 (Blackwell, Cambridge, MA).
27. Labov, W. (2001). *Principles of Linguistic Change: Social Factors*, Vol. 2 (Blackwell, Cambridge, MA).
28. Labov, W. (2007). Transmission and diffusion, *Language* **83**, 2, pp. 344–387.
29. Mitchener, W. G. (2003a). Bifurcation analysis of the fully symmetric language dynamical equation, *Journal of Mathematical Biology* **46**, pp. 265–285, doi:10.1007/s00285-002-0172-8.
30. Mitchener, W. G. (2003b). *A Mathematical Model of Human Languages: The interaction of game dynamics and learning processes*, Ph.D. thesis, Princeton University.
31. Mitchener, W. G. (2005). Simulating language change in the presence of non-idealized speech, in *Proceedings of the Second Workshop on Psychocomputational Models of Human Language Acquisition* (Association for Computational Linguistics), pp. 10–19.
32. Mitchener, W. G. (2007). Game dynamics with learning and evolution of universal grammar, *Bulletin of Mathematical Biology* **69**, 3, pp. 1093–1118, doi:10.1007/s11538-006-9165-x.
33. Mitchener, W. G. and Nowak, M. A. (2003). Competitive exclusion and coexistence of universal grammars, *Bulletin of Mathematical Biology* **65**, 1, pp. 67–93, doi:10.1006/bulm.2002.0322.
34. Mitchener, W. G. and Nowak, M. A. (2004). Chaos and language, *Proceedings of the Royal Society of London, Biological Sciences* **271**, 1540, pp. 701–704, doi:10.1098/rspb.2003.2643.
35. Niyogi, P. (1998). *The Informational Complexity of Learning* (Kluwer Academic Publishers, Boston).
36. Niyogi, P. (2006). *The Computational Nature of Language Learning and Evolution* (MIT Press, Boston).
37. Niyogi, P. and Berwick, R. C. (1996). A language learning model for finite parameter spaces, *Cognition* **61**, pp. 161–193.
38. Niyogi, P. and Berwick, R. C. (1997a). A dynamical systems model for language change, *Complex Systems* **11**, pp. 161–204, URL `ftp://publications.ai.mit.edu/ai-publications/1500-1999/AIM-1515.ps.Z`.
39. Niyogi, P. and Berwick, R. C. (1997b). Evolutionary consequences of language learning, *Linguistics and Philosophy* **20**, pp. 697–719.
40. Nowak, M. A. (2006). *Evolutionary Dynamics: Exploring the equations of life* (Harvard University Press).
41. Nowak, M. A. and Komarova, N. L. (2001). Towards an evolutionary theory of language, *Trends in Cognitive Sciences* **5**, 7, pp. 288–295.
42. Nowak, M. A., Komarova, N. L. and Niyogi, P. (2001). Evolution of universal grammar, *Science* **291**, 5501, pp. 114–118.
43. Nowak, M. A., Komarova, N. L. and Niyogi, P. (2002). Computational and evolutionary aspects of language, *Nature* **417**, 6889, pp. 611–617.
44. Nowak, M. A. and Krakauer, D. C. (1999). The evolution of language, *Proceedings of the National Academy of Sciences, USA* **96**, pp. 8028–8033.

45. Nowak, M. A., Krakauer, D. C. and Dress, A. (1999a). An error limit for the evolution of language, *Proceedings of the Royal Society of London, Series B* **266**, pp. 2131–2136.
46. Nowak, M. A., Plotkin, J. and Jansen, V. A. A. (2000). Evolution of syntactic communication, *Nature* **404**, 6777, pp. 495–498.
47. Nowak, M. A., Plotkin, J. and Krakauer, D. C. (1999b). The evolutionary language game, *Journal of Theoretical Biology* **200**, pp. 147–162.
48. Pearl, L. and Weinberg, A. (2007). Input filtering in syntactic acquisition: Answers from language change modeling, *Language Learning and Development* **3**, 1, pp. 43–72.
49. Plotkin, J. and Nowak, M. A. (2000). Language evolution and information theory, *Journal of Theoretical Biology* **205**, pp. 147–159.
50. Retoré, C. (ed.) (1997). *Logical Aspects of Computational Linguistics*, *Lecture Notes in Computer Science*, Vol. 1328 (Springer-Verlag), ISBN 978-3-540-63700-4.
51. Ritt, N., Schendl, H., Dalton-Puffer, C. and Kastovsky, D. (eds.) (2006). *Medieval English and its Heritage: Structure, meaning and mechanisms of change*, *Studies in English Medieval Language and Literature*, Vol. 16 (Peter Lang, Frankfurt), proceedings of the 13th International Conference on English Historical Linguistics.
52. Shannon, C. E. and Weaver, W. (1949). *The Mathematical Theory of Information* (University of Illinois Press, Urbana, Illinois).
53. Stabler, E. (1997). Derivational minimalism, in [50], pp. 68–95, URL `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.50.3127`.
54. Swarup, S. and Gasser, L. (2007). Unifying evolutionary and network dynamics, *Phys. Rev. E* **75**, 6, p. 066114, doi:10.1103/PhysRevE.75.066114.
55. Tesar, B. and Smolensky, P. (2000). *Learnability in Optimality Theory* (MIT Press).
56. Trapa, P. E. and Nowak, M. A. (2000). Nash equilibria for an evolutionary language game, *Journal of Mathematical Biology* **41**, pp. 172–1888.
57. Warnow, T. (1997). Mathematical approaches to comparative linguistics, *Proceedings of the National Academy of Sciences, USA* **94**, pp. 6585–6590.
58. Yang, C. D. (2002). *Knowledge and Learning in Natural Language* (Oxford University Press, Oxford).

This page is intentionally left blank

Chapter 14

Weighted Automata Modeling for High Density Linkage Disequilibrium Mapping

Tran Trang

Centre Intégré de Bioinformatique, Faculté de Médecine, Université de Lille 2, France,
E-mail: `trang.tran@univ-lille2.fr`

In genetic epidemiology, modeling Linkage Disequilibrium (LD) between Single Nucleotide Polymorphisms (SNP) has become an important problem for disease genes mapping. On the fine scale of high-density maps, the complex diseases are often related to effects of combinations of multiple SNPs while the main effects of the pairwise LD measure may be small or absent. The major challenge of association study, especially at the whole-genome level, is the lack of multilocus LD analysis. This chapter introduces Variable Length Finite Automata (VLFA) for describing multilocus LD. Given a set of haplotype-phenotype data, we propose a novel method and algorithm to infer the structure of VLFA. This VLFA consists of an inhomogeneous variable length Markov chain in which the length of memory of the model depends on the nature of stochastic data. The model will have a longer memory if the SNP markers are in the regions of high LD, whereas, the memory will be short in the regions of low LD. Having a stochastic structure, the VLFA reflects the multilocus LD, the dependence structure of haplotypes, the historical recombination events and the location of disease mutations. Through the experimental results on real published SNP data set, we show that our approach is flexible and robust for genetic association studies.

14.1 Introduction

The areas of bio-informatics and computational biology have strongly developed in recent years. There have been many applications of combinatorial optimization and combinatorial structures in those fields. In particular, discovering exactly or approximately patterns (motifs) are one of most interesting problems of combinatorial optimization and have been shown remarkably effective applications in computational biology. In computational biology, motif finding problems are widely studied from genomic to post-genomic. Most frequently, motif discovery applications arise when identifying shared overrepresented regulatory motifs (binding sites) within DNA sequences (promoter region) or regulatory protein (transcription factors) sharing functional and structure elements within DNA or protein sequences [2, 7, 11]. In the last decade, the combinatorial structures based on finite state automata, and more particularly stochastic finite state automata (probabilistic automata), hidden Markov model and different variants of graphical model have been used quite successfully to address several complex pattern discovery and matching pattern problems in computational biology [16]. These structures, however, typically built in this context are subtended by uniform, homogeneous and fixed-memory length Markov models.

This chapter will consider another essential application of the pattern finding to be used for characterizing and predicting bio-sequence families. Given different families of sequences, the goal is to find all frequent patterns that are significantly more abundant in a specific family than in others [2]. Here, we investigate this kind of pattern discovery to explore single nucleotide polymorphism (SNP) data analysis, timely topics of computational molecular biology, for identifying likely location of disease susceptibility (DS) genes. Currently, haplotype analysis of disease chromosomes can help identify probable historical recombination events and localize DS genes. In particular, the haplotype pattern or haplotype block, known as multi-locus SNP analysis, have been successfully applied to the identification of the DNA variations and are now considered the most promising method for localizing DS genes in complex disease [8, 10, 23, 26]. We are interested in problem of identification of *risk haplotype pattern*. The basic idea in disease-haplotype association is to search for genetic patterns that are more common within affected haplotypes than control haplotypes. This consists in identifying subset of individual haplotypes and subset of contiguous SNP markers, where the haplotypes are strongly correlated behaviours and strongly significant associated with the disease, called *disease-haplotype association*. The method directly explores the sharing of haplotype segments in affected haplotypes from the common ancestral chromosome in the region where the functional mutation occurred in the past that are rarely present in normal haplotypes (Identical By Descent, IBD). If most affected individuals in a population share the same mutant allele at a causative locus, it is possible to narrow the genetic interval around the disease locus by detecting *linkage disequilibrium* (LD) between nearby markers and the disease locus [1, 14, 15].

Since the biological problems addressed by motif finding in this context are complex and itself NP-complete, no existing methods can solve them completely. Most available methods performed pattern discovery without considering the dependent linkage between variables. On the fine scale of high-density maps, the complex diseases are often related to effects of combinations of multiple SNPs while the main effects of the pairwise LD measure may be small or absent. The major challenge of association study, especially at the whole-genome level, is the lack of multilocus LD analysis. Several methods used sliding-window approaches [8, 10, 30], graphical models [28, 22], tree-based recursive partitioning methods [29] and data mining technique [23] taking multilocus LD into account to perform motif finding problem in computational haplotype. These models, however, are not structurally rich and advantageous. If the memory length of process and the window size are too small, information is lost, whereas, if the memory and the window size are too large, excessive noise is introduced. Moreover, growing very large number of genetic markers (refer to HapMap project, *http://hapmap.ncbi.nlm.nih.gov/*) offers new opportunities, but also amplifies the challenging statistical and computational complexity. In fact, a limit of gene-disease association is the relatively large number of observed haplotype sequences that increase the degrees of freedom and then decrease the power for a specific statistic test. Thus, large degrees of freedom reduce the power of sequence association analyses and also limit the modelling capacity for incorporating other factors [8, 10, 29].

To overcome the challenges mentioned above, we introduce in this chapter a combinatorial optimization based on discrete structure of variable length finite automata (VLFA) for motif finding with a new mathematical programming approach and divers novel applications. From a structural point of view, the class of proposed VLFA models is structurally richer than existing models. It includes and generates the class of stationary Markov chains. Unlike the structure of automata widely studied in the literature, the memory of these models are allowed to be of variable length in which the length of memory of the process depends on the context. In the region where the variables are strongly dependent, the models will have a longer memory, whereas, in the regions of low dependence, the memory of the models will be short. In particular, if the variables are independent, the models have a structure of the homogeneous Markov chain. Having a stochastic structure, the VLFA model is more suitable to describe the dependence structure among several variables characterizing the observed haplotypes such as ancestral haplotypes, recombination events, and location of disease mutations. From an algorithmic view, the methods will adapt to the linkage dependence within variables and do not need to choose the length segment and window size parameters. So the algorithms are computationally very fast for large dataset. The proposed algorithms combine and optimize large statistical models represented by VLFA. These algorithms find optimal solution underlying optimization problem with minimal complexity and automatically balance degrees of freedom and number of statistical tests for extracting maximal information. The experimental results show that VLFA model tends to be more effective and powerful for genetic association study comparing with current methods. As a basic pattern discovery, the method described in this chapter will follow four steps:

(1) Choosing the language (formalism) to represent the patterns in a big search space,
(2) Choosing the rating for patterns, to tell what is better than others,
(3) Developing an algorithm that finds the best patterns from a pattern class,
(4) Evaluating the complexity and the efficacy of the developed algorithms.

Next section briefly reviews some basic genetic research backgrounds on gene mapping and computational haplotype analysis. Given a haplotype-phenotype training data set, we generally formulate a combinatorial optimization problem via linear programming formulation adapting to problem of risk haplotype pattern discovery. Section 14.3 reviews some fundamental discrete structures that are basic structures and theoretical elements for studying our contributed models. The new data structure, construction algorithm and the applications of VLFA will be found in Section 14.4. Section 14.5 reports some experimental results through real SNP marker data. The comparative study with related work is also reported. We finish with some conclusions and ideas for future works.

14.2 Biological Problem and Formulation

14.2.1 *Genetic Association Studies*

14.2.1.1 *Genetics Terminologies*

In genomic research, the fundamental unit of population analysis is the *single nucleotide polymorphism* (SNP). A SNP is a mutation occurring when a single nucleotide in the genome differs between paired chromosomes in an individual. It simply describes a single base pair change that is variable across the population at a frequency of at least 1%. A locus of SNP marker is a short identifiable sequence that has a known location, and this sequence varies between individual chromosomes. Variants of markers are called *alleles*. The number of alleles per marker is small, typically, less than 10 for micro-satellite markers or 2 for SNP markers. A *genotype* at a marker is the pair of alleles occurring at that locus on two chromosomes of a diploid individual. A genotype with two identical alleles is called *homozygote*, otherwise, it is called *heterozygote*. An *haplotype* is an ordered sequence of alleles in contiguous SNPs along a region of chromosome. For example, Figure 14.1 represents five mutations at two identical DNA fragments of an individual chromosome that define five associated SNP loci. In this case, one says that the first SNP locus has two alleles, C and A; and second one has alleles A and C, *etc*; the set of genotypes is {C/A, A/C, T/G, T/C, G/T}; CATTG and ACGCT are haplotypes.

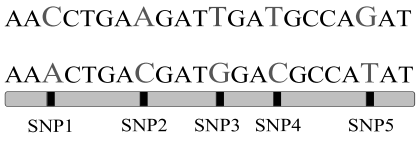


Fig. 14.1 SNP markers, genotypes and haplotypes.

A *trait* is a distinct variant of a phenotypic character of an organism. A binary trait refers to binary variable defined as one that can take on two values of disease status: affected chromosome (diseased) or unaffected chromosome (not diseased). A *phenotype* is defined as a physical attribute of a trait and in the context of association analysis generally refers to a measure of disease progression.

The *linkage disequilibrium* (LD) is defined as an association in the alleles present at each of two sites on a genome. When alleles at different loci in an haplotype are not independent (none randomly associated in gametes), they are said to be in LD and when alleles in a haplotype are independent, they are said to be in *linkage equilibrium* (LE). Formally, let $A|a$ and $B|b$ be the possible alleles at two loci. Denote P_A and P_B the allele frequencies of alleles A and B, and P_{AB} the frequency of the haplotype AB combining from alleles A and B. If the alleles of two loci are in LE, then $P_{AB} = P_A P_B$. In contrast, when the alleles of two loci are in LD to each other, then $P_{AB} \neq P_A P_B$. In this case, the difference between the frequencies of allelic combinations in LD and the frequencies of allelic combinations in LE, noted $D_{AB} = P_{AB} - P_A P_B = D_{ab} = D$ and $D_{Ab} = P_{Ab} - P_A(1 - P_B) = D_{Ba} = -D$, is the measure of the LD between the nearby loci on the chromosome. In practice, the Pearson's correlation coefficient r^2 or the measurement D' value, less sensitive to the allelic frequencies than D value, were used to quantify the LD:

$$\begin{aligned} r^2 &= \frac{D^2}{P_A P_B (1 - P_A)(1 - P_B)}, \\ D' &= \begin{cases} \dfrac{D}{\max(P_A P_B, (1 - P_A)(1 - P_B))} & \text{if } D < 0, \\ \dfrac{D}{\min(P_A(1 - P_B), (1 - P_A)P_B)} & \text{if } D > 0. \end{cases} \end{aligned} \tag{14.1}$$

If $|D'| = 1$ (resp. $r^2 = 1$) then the two SNPs are in complete LD which means knowing one of them is directly predictive of the other, and $|D'| = 0$ (resp. $r^2 = 0$) indicates that the two SNPs are independent. In the process of gamete production, a pair of paternal homologous chromosomes exchanges genetic material (segment of DNA) with each other during the meiosis is called *recombination event* (see left graph of Figure 14.2). As a result, a chromosome transmitted from a parent to an offspring is not an exact copy of either parental chromosomes, but a mosaic of them. The recombination event ensures that the variation is maintained in each generation between individual. This is very basic phenomenon in genetics. The recombination event directly influences LD across generations. The relationship between recombination event and LD in each generation t is given by:

$$D'_t = (1 - \theta)^t D'_0, \tag{14.2}$$

where $0 \leq \theta \leq 1$ is the recombination rate [1, 14]. The right graph of Figure 14.2 gives an illustration of this relationship for $\theta = 0.5, \theta = 0.2$ and $\theta = 0.1$. Thus, we can see that the recombination event occurring between SNPs reduces the LD between them and SNPs close together are less likely

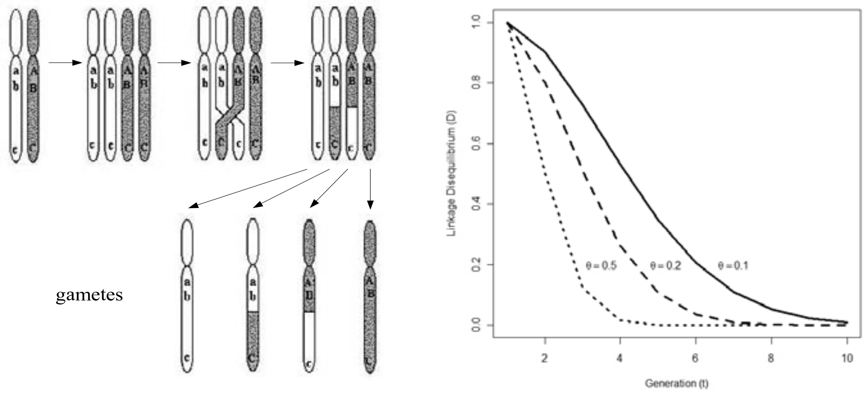


Fig. 14.2 Link between recombination event and LD over generations.

to be affected by recombination than two SNPs far away. Historical recombination between pair of chromosomes results in the decay of D' toward zero along the time axis at a rate depending on the magnitude of the recombination rate θ between the two loci. Consequently, LD blocks, the key for gene mapping analysis, are regions in which the standard measure D' of pair-wise LD is consistent (or nearly consistent) with no recombination, $|D'| \approx 1$ ($r^2 \approx 1$), for all (or nearly all) pairs of markers in the region.

14.2.1.2 *Association Analysis*

In general, the exact location of disease susceptibility (DS) gene is unknown. The goal of gene mapping analysis is to identify a set of DNA variations (SNPs) for localizing DS genes in complex disease. *Association analysis* is a method potentially useful for identifying a set of SNP associated with target disease (also called marker-trait associations) based on LD measure. This method was therefore sought for narrowing the interval in which a disease gene might lie, and one of these was by the analysis of LD [1, 13–15]. If most affected individuals in a population share the same mutant allele at a causative locus, it is possible to narrow the genetic interval around the disease locus by detecting disequilibrium between nearby markers and disease locus. The approach makes use of the many opportunities for crossovers between markers and the disease locus during the large number of generations since the first appearance of the mutation (Figure 14.3). Since the candidate gene generally involves multiple SNPs, the location of candidate gene depends on the identification of block of SNPs in high LD.

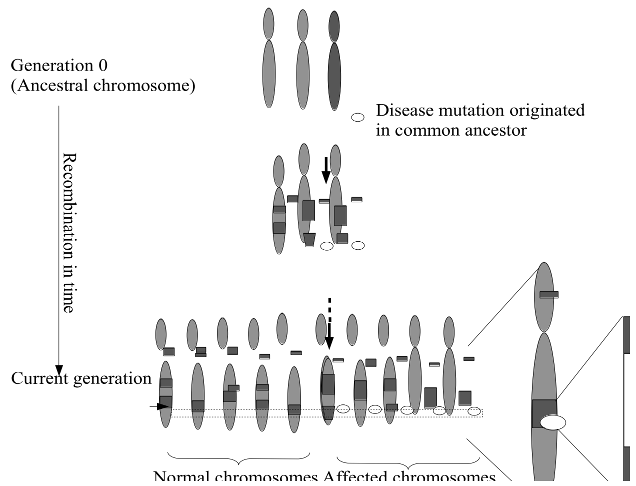


Fig. 14.3 Association analysis via LD around an ancestral mutation. The mutation is indicated by a white region. Chromosomal stretches derived from common ancestor of all mutant chromosomes are shown in dark region, and new stretches introduced by recombination are shown in light grey.

Therefore, the approaches must be referred to as multi-locus methods and are specifically designed to find multiple disease loci (Figure 14.4).

Single-SNP, genotype and haplotype information can be used to study the genetic variations associated with a specific disease. However, haplotype information are more informative and have advantages compared with the single-SNP analysis and genotype analysis [26]. In fact, the haplotype sequences can be used to capture regional LD information and identifying haplotype blocks is one way of the studying LD patterns[6, 14]. The LD causes the associations between markers and disease even for markers that are part of a disease-causing gene. These studies examine the association between a particular set of haplotype patterns and disease. In particular, the use of haplotypes also capture information about common patterns that may be descended from ancestral haplotypes (IBD). As illustrated by Figure 14.3, around the DS gene region, it is expected that affected haplotypes should share segments from the common ancestral haplotype where the mutation occurred a long time ago in the past. It captures the sharing of haplotype segments due to historical recombination events and incorporates the mutations. Instead, haplotypes from affected individuals are expected to be more similar at the disease gene location than those from controls that are assumed to be random samples.

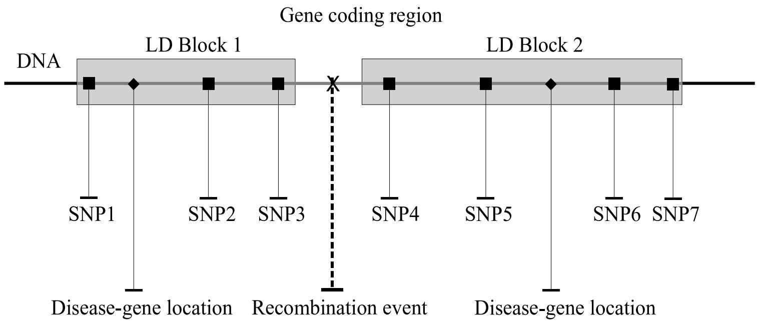


Fig. 14.4 Block of linkage disequilibrium and location of disease causing variant.

14.2.2 *Haplotype Pattern Discovery Problem*

In computational haplotype analysis, haplotype-disease association aims at identify a set of risk haplotype patterns associated with target disease. The input data consist of case ($\mathcal{L}^+$) and control ($\mathcal{L}^-$) haplotypes on a map of loci markers along chromosome. In a population-based study, case haplotypes are obtained from affected individuals; controls are from unaffected individuals. In a family-based analysis, case haplotypes can be parental haplotypes transmitted to offspring; control haplotypes are those non-transmitted haplotypes. Table 14.1 represents the input haplotype-phenotype data mapping from n loci markers in the physical order along chromosome. Given a large number of mapping genetic markers and a collection of affected and control haplotypes, the task is to identify likely location of risk haplotype patterns in high LD to predict the location of the disease susceptibility (DS) gene on the map. The basic idea is to search haplotype patterns that are more common within cases than controls. In each model, a group of case haplotypes is compared with a group of control haplotypes in terms of haplotype frequencies. If most affected individuals in a population share the same mutant allele at a causative locus, it is possible to narrow the genetic interval around the disease locus by detecting LD between nearby markers and the disease locus.

An haplotype sequence can be viewed as string over an alphabet of size 2 for SNP markers and less than 10 for micro-satellite markers. Without loss of generality, consider a family $\mathcal{L} = \{s_1, \ldots, s_m\}$ of m strings of length n over alphabet $\mathcal{A} = \{x_1, x_2, \ldots, x_k\}$ of size k. Let us denote $\mathcal{L}^+$ and $\mathcal{L}^-$ two distinct subfamilies of related strings of $\mathcal{L}$ ($\mathcal{L}^+ \cup \mathcal{L}^- = \mathcal{L}$). Given such data structure, we address in this work a novel combinatorial problem, called

Table 14.1 Input haplotype-phenotype training dataset.

Haplotype string	Locus 1	Locus 2	$\dots$	Locus n	Phenotype
s_1	$s_{1,1}$	$s_{1,2}$	$\cdots$	$s_{1,n}$	$+$
s_2	$s_{2,1}$	$s_{2,2}$	$\cdots$	$s_{2,n}$	$-$
$\vdots$	$\vdots$	$\vdots$		$\vdots$	
s_m	$s_{m,1}$	$s_{m,2}$	$\cdots$	$s_{m,n}$	$+$

protective linkage patterns , of the motif discovery on strings:

Given two subsets of positive strings $\mathcal{L}^+$ and negative one $\mathcal{L}^-$, find a set of common patterns $\mathcal{P}$ such that each element $M \in \mathcal{P}$ has significantly more occurrences in $\mathcal{L}^+$ than in $\mathcal{L}^-$ and all elements within M are closely related (dependent linkage or causal dependence).

There are basic common questions concerning this problem when searching for unknown protective linkage patterns associated with a specific family of strings: Are there sharing patterns between strings? Where these patterns are located in? Is there the dependency linkage between elements within pattern (Markov property)?

14.2.3 *Combinatorial Optimization Formulation*

Let $X \in \mathcal{M}_{m,n}(\mathcal{A})$ be symbolic matrix data over alphabet $\mathcal{A}$ which corresponds to $\mathcal{L} = \{s_1, \dots, s_m\}$ of m strings of length n, each row of X is a string of $\mathcal{L}$ along n columns. Let $\mathcal{L}^+$ and $\mathcal{L}^-$ be two distinct subfamilies of related strings of $\mathcal{L}$. Suppose that we are given $m^+ = \text{Card}(\mathcal{L}^+)$ and $m^- = \text{Card}(\mathcal{L}^-)$ $(m^+ + m^- = m)$. A pattern of X is a subset of rows that are identical string over alphabet $\mathcal{A}$ across a subset of arbitrary columns (called submatrix, motif, bi-cluster) [17, 25]. In this work we are interested in *contiguous pattern.* A contiguous pattern is a subset of rows that are identical string over alphabet $\mathcal{A}$ across a subset of contiguous columns. Formally, given subset of indices $I = \{i_1, i_2, \dots, i_\ell\} \subseteq [1, \dots, m]$ of rows and subset of consecutive indices $J = \{t, t+1, \dots, t+h\} \subseteq [1, \dots, n]$ of columns $(h > 0)$, the submatrix $X_{I,J}$ is called *coherent contiguous pattern* (CCP) of matrix X if

$$X_{i_1,j} = \cdots = X_{i_\ell,j} \quad \text{for all } j \in J \text{ and } i_1, \dots, i_\ell \in I. \tag{14.3}$$

In other words, the pattern $X_{I,J}$ is then represented by an identical string of length $|J|$. In general, there are 2^m and 2^n possibilities for choosing two subsets I and J in the sets of m rows and n columns, respectively.

Thus, there are $2^m \times 2^n$ possible patterns in the matrix X. Here, we are interested in problem of identification of *maximal patterns* (largest patterns) that are shared by almost rows and columns of X. The problem consists in identifying I and J such that $X_{I,J}$ has the largest size given by the map

$$\begin{aligned} f : \mathcal{M}_{2^m,2^n}(\mathcal{A}) &\to \mathbb{N} \\ X_{I,J} &\mapsto f(X_{I,J}) := |I| + |J|. \end{aligned} \tag{14.4}$$

From this definition, the function $f(.)$ is the size or the frequency of the submatrix $X_{I,J}$ and considering as the *objective function* of the problem. The set $\mathcal{M}_{2^m,2^n}(\mathcal{A})$ is the search space of the size $2^m \times 2^n$. The maximization of the objective function $f(.)$ is a combinatorial optimization problem in which we search the maximal value of $f(.)$ for all submatrices verifying the coherent constraint (14.3) in the search space $\mathcal{M}_{2^m,2^n}(\mathcal{A})$. The extracted matrix $X_{I,J}$ with $f(X_{I,J})$ maximized is called *maximum frequent coherent pattern* of the matrix X. An optimal solution $\widehat{X}_{I,J} \in \mathcal{M}_{2^m,2^n}(\mathcal{A})$ is called global maximum if for all $X_{I,J} \in \mathcal{M}_{2^m,2^n}(\mathcal{A}), f(\widehat{X}_{I,J}) \geq f(X_{I,J})$.

For each pattern $X_{I,J}$, with I and J defined as above, the dependent linkage between variables in the pattern (so-called *linkage pattern*) is given by the conditional probability defined as follows (see Section 14.3 for detail)

$$P(X_{t+h} \mid X_1, X_2, \ldots, X_n) = P(X_{t+h} \mid X_t, X_{t+1}, \ldots, X_{t+h-1}). \tag{14.5}$$

We are interested in identifying CCP associated with a specific family of strings by adding the following constraint

$$\forall C \geq 1, \quad P(X_{I,J} \in \mathcal{L}^+) \geq CP(X_{I,J} \in \mathcal{L}^-), \tag{14.6}$$

where $P(X_{I,J} \in \mathcal{L}^+)$ and $P(X_{I,J} \in \mathcal{L}^-)$ are respectively the appearing probabilities of the pattern $X_{I,J}$ in $\mathcal{L}^+$ and in $\mathcal{L}^-$ and the coefficient C is called *degree of association.* In practice, the standard $\pm\chi^2$-score is used to measure degree of association between string pattern and family of strings of interest [10, 23]. Let m_P, m_{AP} and m_{CP} denote the string pattern count, the number of positive strings and the number of negative strings in the pattern $X_{I,J}$, respectively. Consider the following 2x2 contingency table of the number of positive strings and negative strings in $X_{I,J}$,

	Match Pattern	Non Match Pattern	$\sum$
Positive string	m_{AP}	m_{AN}	m^+
Negative string	m_{CP}	m_{CN}	m^-
$\sum$	m_P	m_N	m

where $m_{AN} = m^+ - m_{AP}, m_{CN} = m^- - m_{CP}$ and $m_N = m - m_P$. This contingency table can be constructed the $\pm\chi^2$-score of association of $X_{I,J}$ as follows:

$$\pm\chi^2(X_{I,J}) = \frac{\sqrt{m}(m_{AP}.m_{CN} - m_{AN}.m_{CP})}{\sqrt{m^+.m^-.m_P.m_N}}. \tag{14.7}$$

The measure $\pm\chi^2$ of a pattern is the standard χ^2 measure where the sign is positive if the relative frequency of the pattern is higher in $\mathcal{L}^+$ than in $\mathcal{L}^-$ ($m_{AP}/m^+ > m_{CP}/m^-$) and negative otherwise. The string patterns satisfying (14.7) with $\pm\chi^2 > 0$ is called the *protective patterns.* And the large positive $\pm\chi^2$ value means strong association between the pattern and the family $\mathcal{L}^+$ since many positive strings in $\mathcal{L}^+$ share the same factor. Given a positive association threshold $\beta > 0$, we say that the pattern $X_{I,J}$ is statistically significant associated with family $\mathcal{L}^+$ if $\pm\chi^2(X_{I,J}) \geq \beta$. Denote $\mathcal{P}^+ = \{X_{I,J} \subset X | \pm\chi^2(X_{I,J}) \geq \beta\}$ the set of all strongly associated patterns.

Theorem 14.1. *Given a threshold $\beta > 0$ of lower bound for $\pm\chi^2(X_{I,J})$. If the pattern $X_{I,J}$ is associated with family $\mathcal{L}^+$, we have*

$$m_{AP} \geq \frac{m.m^+.\beta}{m.m^- + m^+.\beta}.$$

The proof of the theorem was reported in [23]. The threshold β is provided by the user and discussed in the experimental section. Biologically, the pattern in $\mathcal{P}$ with highest $\pm\chi^2$-score means that many affected chromosomes of present generation share a common genomic region. The marker positions associated with this pattern is taken as the prediction for predicted DS genes.

Combining (14.3), (14.4), (14.5) and (14.7), the problem of finding *protective linkage patterns* is formulated by the following combinatorial optimization problem:

$$\begin{cases} \text{Maximize} \quad f(X_{I,J}) := |I| + |J| \\ \text{subject to} \\ \text{(a) } X_{i_1,j} = \cdots = X_{i_\ell,j}, \forall i_1, \ldots, i_\ell \in I \ \text{ and } \ j \in J \\ \text{(b) } P(X_{t+h} \mid X_1, \ldots, X_n) = P(X_{t+h} \mid X_t, X_{t+1}, \ldots, X_{t+h-1}), \\ \text{(c) } \pm\chi^2(X_{I,J}) \geq \beta, \quad \forall\beta > 0, \\ \text{for } \ I = \{i_1, i_2, \ldots, i_\ell\}, J = \{t, t+1, \ldots, t+h\}, \forall h > 0. \end{cases} \tag{14.8}$$

The extracted matrix $X_{I,J}$ identified by (14.8) is called *maximum protective linkage pattern* of the input matrix X. The two subsets I and J are called an optimal solution. The first constraint (a), called *coherent constraint*, ensures that the identified pattern is coherent (identical string) on the consecutive locus columns. The second one (b) is called *linkage constraint* that guarantees that the variables in the pattern are related to each other. This constraint means that the pattern follows the structure of inhomogeneous high order Markov chains. Finally, the *association constraint* (c) ensures that the identified pattern is significantly more overrepresented in $\mathcal{L}^+$. Note that the problem (14.8) returns a local optimization (local maximum values). The computational complexity of the problem (14.8) depends on the objective function f. In general, seeking an optimal solution for combinatorial optimization problem formulated as (14.8) is NP-complete [17]. Since the linkage pattern depends on the context and the size of search space is very large, we search a model that has variable length memory to model variant linkage patterns (sometime called learning linkage patterns) and an efficient algorithm to maximize the contiguous patterns.

Biologically, the problem (14.8) is based on the simple observation that LD with DS gene locus is likely to be where strongly disease-associated haplotype segments are. In this approach, the LD is examined by looking for haplotype patterns that consists of a set of consecutive markers. This method makes use of the many opportunities for crossovers between markers and the disease locus during the large number of generations since the first appearance of the mutation. It directly explores the sharing of haplotype segments, due to historical recombination events, in affected haplotypes deriving from the common ancestral chromosome in the region where the functional mutation occurred in the past that are rarely present in normal haplotypes.

Example 14.2. Let $\mathcal{L}^+ = \{s_1, s_4, s_5, s_7\}$ and $\mathcal{L}^- = \{s_2, s_3, s_6, s_8\}$ be respectively the sets of case and control haplotypes from 5 bi-allelic SNP markers (each marker has only two alleles, major and minor alleles) coded by binary strings over the alphabet $\mathcal{A} = \{1, 2\}$ (the first allele at a marker is coded by symbol 1 and the second one is coded by symbol 2) and represented as Table 14.2. The submatrix $X_{[2,3,5,8],[1,2]} = \begin{Bmatrix} 1\,1\,1 \\ 2\,2\,2 \end{Bmatrix}^T$, composed by four haplotypes $[2, 3, 5, 8]$ sharing the same alleles $[\mathbf{1,2}]$ at two consecutive markers $[1, 2]$, is a pattern of size $f(X_{[2,3,5,8],[1,2]}) = 6$ representing haplotypic string pattern **12**. This pattern has the number of case count $N_{AP} = 1$ and control count $N_{CP} = 3$. We have then $P(\mathbf{12} \in \mathcal{L}^+) = \frac{1}{4} < P(\mathbf{12} \in \mathcal{L}^-) = \frac{3}{4}$. Thus, the pattern **12** at markers $[1, 2]$ is not associated

Table 14.2 Example of haplotype-disease data.

Haplotype	SNP 1	SNP 2	SNP 3	SNP 4	SNP 5	Phenotype
s_1=AGCAG	1	1	**1**	**1**	**2**	+
s_2=ATCAG	**1**	**2**	**1**	**1**	**2**	−
s_3=ATTCA	**1**	**2**	2	2	1	−
s_4=AGCAG	1	1	**1**	**1**	**2**	+
s_5=ATCAG	**1**	**2**	**1**	**1**	**2**	+
s_6=TTTCA	2	2	2	2	1	−
s_7=TGCAG	2	1	**1**	**1**	**2**	+
s_8=ATTCA	**1**	**2**	2	2	1	−

with the disease. While the submatrix $X_{[1,2,4,5,7],[3,4,5]} = \left\{\begin{matrix} 1\,1\,1\,1\,1 \\ 1\,1\,1\,1\,1 \\ 2\,2\,2\,2\,2 \end{matrix}\right\}^T$, composed by five haplotypes $[1, 2, 4, 5, 7]$ sharing the same alleles [**1,1,2**] across over three consecutive markers $[3, 4, 5]$, is a largest pattern with the size $f(X_{[1,2,4,5,7],[3,4,5]}) = 8$ representing haplotype pattern **112**. This pattern has the number of case count $N_{AP} = 4$ and control count $N_{CP} = 1$. We have then $P(\mathbf{112} \in \mathcal{L}^+) = 4/5 > P(\mathbf{112} \in \mathcal{L}^-) = \frac{1}{5}$. That deduces $\pm\chi^2(\mathbf{112}) = 2.01$. The pattern **112** at markers $[3, 4, 5]$ is a significant pattern in risk disease and considered a risk haplotype pattern. The pair of indices $I = [1, 2, 4, 5, 7]$ and $J = [3, 4, 5]$ is the optimal solution of combinatorial maximization problem.

14.3 Fundamentals of Discrete Structures

14.3.1 *Strings and Languages*

Let $\mathcal{A} = \{x_1, \dots, x_k\}$ be an alphabet of the size k. A *string* (or *word*) s is a sequence of elements of $\mathcal{A}$. The empty string is denoted by ε. The set of all possible strings over $\mathcal{A}$ is denoted by $\mathcal{A}^*$. The length of the string s is noted by $|s|$, and $|\varepsilon| = 0$. For any $h \geq 0$, we note $\mathcal{A}^h = \{s \in \mathcal{A}^* \mid |s| = h\}$.

A string $u \in \mathcal{A}^*$ is called a *factor* (resp. a *prefix*) of a string s if there exist strings $x, y \in \mathcal{A}^*$ such that $s = xuy$ (resp. $s = uy$). For $i < j$, denote $s[i : j] = s_i s_{i+1} \dots s_j$ a factor of the string s starting at i and ending at j. The prefix of length i of a string is also denoted by $s[1, i]$. A *longest common factor* (LCF) of two string u and v beginning at the same position i is defined as a longest factor a such that $u = u[1 : i - 1]au'$ and

$v = v[1:i-1]av'$ and noted by $a = \mathrm{Fact}_i(u,v)$. In the case $u[1:i-1] = v[1:i-1] = \varepsilon$, the factor a is the longest common prefix of u and v [24].

Let $\mathcal{L} = \{s_1, \ldots, s_m\} \subseteq \mathcal{A}^*$ be a finite language containing m strings such as, for $j = 1,..,m, |s_j| \in \mathcal{A}^n$. A string u is called a LCF of $\mathcal{L}$ at the position i, noted by $u = \mathrm{Fact}_i(\mathcal{L})$, if for all $s \in \mathcal{L}$, $s = s[1:i-1]uv$ and $|u|$ is maximal. In this context, for $u \in \mathcal{A}^*$, we define

$$\mathcal{L}_u := \{s \in \mathcal{L} \mid \exists u \in \mathcal{A}^*, s = s[1:i-1]uv\} \tag{14.9}$$

the subset of strings of $\mathcal{L}$ having the LCF u. The common factor u defined by this way represents a CCP (coherent contiguous pattern) of $\mathcal{L}$. The set of all LCFs of $\mathcal{L}$ is denoted by

$$\mathrm{Fact}(\mathcal{L}) = \bigcup_{i=1}^{n} \mathrm{Fact}_i(\mathcal{L}). \tag{14.10}$$

For $u \in \mathcal{A}^*$, let us consider $N_u = \mathrm{Card}(\mathcal{L}_u)$. In particular $N_\varepsilon = m$. Thus, $\forall u \in \mathcal{A}^h$, one has

$$N_u = \sum_{x \in \mathcal{A}} N_{ux} \text{ and for any } 0 \leq h \leq n, m = \sum_{u \in \mathcal{A}^h} N_u. \tag{14.11}$$

Definition 14.3. Let $\mathcal{L}$ be a language and let $\mathrm{Fact}(\mathcal{L})$ be a set of all LCFs of $\mathcal{L}$. The objective function (14.4) of the combinatorial optimization formulated as (14.8) is rewritten by

$$\begin{aligned} f : \mathrm{Fact}(\mathcal{L}) &\to \mathbb{N} \\ u &\mapsto f(u) := |u| + N_u. \end{aligned}$$

Using the notation (14.11), for $u \in \mathcal{A}^h$ and $x_i \in \mathcal{A}$, let us consider the following ratios

$$P_u = \frac{N_u}{m} \text{ and } P_{ux_i} = \frac{N_{ux_i}}{N_u}, i = 1, \ldots, k. \tag{14.12}$$

Since $\sum_{x_i \in \mathcal{A}} N_{ux_i} = N_u$ then the ratios P_{ux_i} define the *discrete probability* over $\mathcal{A}^*$, $0 \leq P_{ux_i} \leq 1$ and for all $u \in \mathcal{A}^h, \sum_{x_i \in \mathcal{A}} P_{ux_i} = 1$. The probability of string u conditionally dependent with string v, denoted by $P(u \mid v)$, is defined by

$$\forall v \in \mathcal{A}^*, \quad P(u \mid v) = \frac{N_{vu}}{N_v}. \tag{14.13}$$

Definition 14.4. A language $\mathcal{L}$ over an alphabet $\mathcal{A}$ is a probabilistic language over $\mathcal{A}^*$ if there exists a distribution function $P : \mathcal{A}^* \to [0,1]$ satisfying $\sum_{s \in \mathcal{A}^*} P(s) = 1$. And two probabilistic languages $\mathcal{L}_1$ and $\mathcal{L}_2$ are identical if for all $s \in \mathcal{A}^*, P(s|\mathcal{L}_1) = P(s|\mathcal{L}_2)$.

Definition 14.5. Given three string u,v and w, we say w is *conditionally independent* of u given v if and only if the conditional probability of w given u and v does not depend on the value of v,

$$P(w \mid u, v) = P(w \mid v).$$

If we consider the joint distribution of string w and string u conditioned on string v, this definition can be expressed in a slightly different way

$$P(w, u \mid v) = P(w \mid v)P(u \mid v). \tag{14.14}$$

The definition of conditional independence can be also interpreted as follows: learning about w has no effect on knowledge information of u given knowledge concerning v, and vice versa.

The basic measure between pairwise of strings frequently used in data mining is the Hamming distance. It is only defined on words of equal length. The Hamming distance takes into account the different symbols at the same position within two strings.

Definition 14.6. The Hamming distance between any pair of strings of $\mathcal{L}$ is the map $d_H : \mathcal{L} \times \mathcal{L} \to \mathbb{N}$ such that $\forall (s_i, s_j) \in \mathcal{L} \times \mathcal{L}, d_H(s_i, s_j) := \sum_{h=1}^{n} \mathbf{1}_{\{s_{i_h} \neq s_{j_h}\}}$, where $\mathbf{1}_{x \neq y}$ is an indicator function which return 1 if $x \neq y$ and 0 otherwise.

Another measure based on string similarity by counting all identical symbols at all positions within two strings is often used. The idea is that the similarity is greater if the strings are closely related.

Definition 14.7. The similarity function which measures the similarity between any pair of strings of $\mathcal{L}$ is defined as d_S given by the map $d_S : \mathcal{L} \times \mathcal{L} \to [0, 1]$ such that $\forall (s_i, s_j) \in \mathcal{L} \times \mathcal{L}, d_S(s_i, s_j) = \sum_{h=1}^{n} \mathbf{1}_{\{s_{i_h} = s_{j_h}\}}/n$, where $\mathbf{1}_{x=y}$ is an indicator function.

Remark 14.8. Note that $1 - d_S(s_i, s_j)$ is then normalized Hamming distance. In biological term, the similarity is greater if the haplotypes across SNP loci markers are closely related to a specific family and share more of the genome IBD. Thus, the similarity function d_S and Hamming distance can be used to distinguish between different disease gene mutation carriers and/or to measure a relationship between haplotypes in family. Most existing clustering algorithms use the Hamming metric based on the mutation model [8, 10, 23]. It allows us to quantitatively measure a (genetic) similarity between any pair of haplotypes taken from the set $\mathcal{L}^+$ and $\mathcal{L}^-$.

14.3.2 *Elements of Probabilistic Automata*

A *Probabilistic Finite Automaton* (PFA) is a finite state automaton where the transition and/or emission functions are probabilistic functions. The probabilistic language over an alphabet $\mathcal{A}$ can be generated by a PFA [20]. PFAs are finite state machines that generate strings of finite length in the probabilistic manner as follows. A PFA has a set of designated starting states and final states. Starting from an initial state, at each step an edge outgoing from the current state is chosen according to the transition probability assigned to that edge, and the symbol drawn from $\mathcal{A}$ labelling the edge is emitted. We associate for each edge a probability, where the sum of all outgoing probabilities and final probability of a state is equal to one. The analytic structure of PFA is defined as follows.

Definition 14.9. A PFA is a 6-tuples $\mathfrak{A} = (\mathcal{A}, \mathcal{Q}, I, F, T, P)$, where

- $\mathcal{A}$ is a finite alphabet,
- $\mathcal{Q}$ is a finite set of states,
- $I : \mathcal{Q} \to [0,1]$ is the probability of initial states,
- $F : \mathcal{Q} \to [0,1]$ is the probability of terminal states,
- $T : \mathcal{Q} \times \mathcal{A} \to \mathcal{Q}$ is the transition function, $\forall x \in \mathcal{A}, \forall q_i, T(q_i, x) = q_{i+1}$,
- $P : \mathcal{Q} \times \mathcal{A} \to [0,1]$ is the next symbol probability function,

such that the applications I, P, F verify the following constraints:

$$\sum_{q\in\mathcal{Q}} I(q) = 1, \quad \forall p \in \mathcal{Q}, \quad F(p) + \sum_{x\in\mathcal{A}} P(p,x) = 1.$$

The transition function T can be extended to be defined recursively on $\mathcal{Q} \times \mathcal{A}^*$ as well: $\forall q \in \mathcal{Q}$ and for all $s = x_{i_1} \dots x_{i_h} \in \mathcal{A}^*, T(q, x_{i_1} \dots x_{i_h}) = T(T(q, x_{i_1} \dots x_{i_{h-1}}), x_{i_h})$. The mechanism generating finite string of a PFA is in the following manner. Beginning from an initial state $I(q_0)$, until a final state $F(p)$ is reached, if q is the current state, the next symbol x is probabilistically chosen according to $P(q,x)$. If x is the symbol generated, the next state is $T(q,x) = r$. Thus, if $\mathfrak{A}$ is a *probabilistic non-deterministic finite automata* (PNFA), the probability of a string $s = x_{i_1} \dots x_{i_n} \in \mathcal{A}^*$ generated by $\mathfrak{A}$, denoted by $P_{\mathfrak{A}}(s)$ is computed by as follows:

$$P_{\mathfrak{A}}(s) = \sum_{q_{i_0}, q_{i_1}, \dots, q_{i_n} \in \mathcal{Q}} I(q_{i_0}) \prod_{h=1}^{n-1} P(q_{i_h}, x_{i_{h+1}}) F(q_{i_n}). \tag{14.15}$$

A *probabilistic deterministic finite automaton* (PDFA) is a PFA $\mathfrak{A} = (\mathcal{A}, \mathcal{Q}, I, F, T, P)$ where for each pair of state and input symbol $x \in \mathcal{A}$

there is one and only one transition to a next state, i.e. $\forall p \in \mathcal{Q}$, the transition $T(p, x) = q$ is unique. For each string, there is only one path from the initial state to a final state labelled by this string. Thus, the probability of a string $s = x_{i_1} \dots x_{i_n} \in \mathcal{A}^*$ generated by $\mathfrak{A}$, denoted by $P_{\mathfrak{A}}(s)$, is computed by as follows:

$$P_{\mathfrak{A}}(s) = I(q_0) \prod_{h=1}^{n-1} P(q_{i_h}, x_{i_{h+1}}) F(q_{i_n}). \tag{14.16}$$

In this work, we consider the class of PDFA. The language recognized by a PDFA is defined by

$$\mathcal{L} = \{s \mid P_{\mathfrak{A}}(s) \neq 0 \text{ and } \sum_s P_{\mathfrak{A}}(s) = 1\}.$$

A PDFA recognizing the language $\mathcal{L}$ is described as follows. The states are the nonempty sets of the string with form $u^{-1}\mathcal{L}$ for $u \in \mathfrak{A}^*$, where $u^{-1}\mathcal{L} = \{v \in \mathcal{A}^* | xv \in \mathcal{L}\}$. The initial state corresponds to empty string ε and the final states are the sets $u^{-1}\mathcal{L}$ with $u \in \mathcal{L}$. The transition from the state $u^{-1}\mathcal{L}$ to the state $(xu)^{-1}\mathcal{L}$ is labelled by letter $x \in \mathcal{A}$.

Adapting the linear representation to probabilistic automaton, the computations are done simply by using linear algebra [19, 20]. The following definition gives the relationship between the morphism of monoid over $\mathcal{A}^*$ and the monoid of square matrices with coefficients in probabilistic space.

Definition 14.10. Let $\mathbb{K} = [0, 1]$ a probabilistic space. Let $\mathfrak{A} = (\mathcal{A}, \mathcal{Q}, I, F, T, P)$ be a PFA over $\mathbb{K}$. The matrical representation of $\mathfrak{A}$ is the triplet $(\lambda, \mathbb{P}, \gamma)$, where

- $\lambda \in \mathcal{M}_{1,N}(\mathbb{K})$ is a row vector, $\lambda_i = I(q_i)$ if $q_i \in I$,
- $\gamma \in \mathcal{M}_{N,1}(\mathbb{K})$ is a column vector, $\gamma_i = F(q_i)$ if $q_i \in F$,
- $\mathbb{P} : \mathcal{A} \longrightarrow \mathcal{M}_{N,N}(\mathbb{K})$ is a morphism of monoid of $\mathcal{A}^*$ representing $\mathfrak{A}$ in the monoid of the square matrices $\mathcal{M}_{N,N}(\mathbb{K})$ with the coefficients in $\mathbb{K}$. For $x \in \mathcal{A}$, the coefficient of transitions matrix $\mathbb{P}(x)$ of the symbol x is given by

$$\forall p, q \in \mathcal{Q}, \quad \mathbb{P}_{pq}(x) = \begin{cases} P(p, x) & \textit{if } \; T(p, x) = q, \\ 0 & \textit{otherwise.} \end{cases}$$

Proposition 14.11. *For any $u, v \in \mathcal{A}^*$, one has $\mathbb{P}(uv) = \mathbb{P}(u)\mathbb{P}(v)$. Expandedly, for any $s = x_{i_1} \dots x_{x_n} \in \mathcal{A}^*$, one has $\mathbb{P}(w) = \mathbb{P}(x_{i_1}) \dots \mathbb{P}(x_{i_n})$.*

A major application of probabilistic automaton is to recognize a given string. A string w is recognized by $\mathfrak{A} = (\lambda, \mathbb{P}, \gamma)$ if and only if it labels a path going from the initial state to a final state.

Theorem 14.12. *Let $\mathfrak{A} = (\lambda, \mathbb{P}, \gamma)$ be a PA. The appearance probability a string s is computed by*

$$P(s|\mathfrak{A}) = P_{\mathfrak{A}}(s) = \lambda \mathbb{P}_s \gamma.$$

In the case $P_{\mathfrak{A}}(s) \neq 0$, the string s is recognized by automaton $\mathfrak{A}$.

14.3.3 *Variable Length Processes on Strings*

Consider a full stationary process $\{X_t\}_{t\in\mathbb{Z}}$ of finite order h taking values in $\mathcal{A}$. Denote by capital letter X random variables and by small letters x fixed deterministic values of X. Denote also by $x_i^j = x_j, x_{j-1}, \ldots, x_i \in \mathcal{A}^*, i \leq j, i, j \in \mathbb{Z}$ a substring written in reverse order. Thus, for all $x_{-\infty}^0$, the process $\{X_t\}_{t\in\mathbb{Z}}$ of order h can be expressed by

$$P(X_1 = x_1 | X_{-\infty}^0 = x_{-\infty}^0) := P(X_1 = x_1 | X_{-h+1}^0 = x_{-h+1}^0). \quad (14.17)$$

We use the idea of a variable length memory which can also be seen as block of states in the history x_{-h+1}^0. Only some values from the infinite history $x_{-\infty}^0$ of the variable X_1 are considered. These can be thought of as a context for X_1. To achieve a flexible model, we let the length of a context depend on the recent values $x_{-\infty}^0$. Based on such idea, we can formulate such context by the variable length stationary processes adapting the definition presented in [4].

Definition 14.13. Let $\{X_t\}_{t\in\mathbb{Z}}, X_t \in \mathcal{A}$ be a stationary stochastic process. Denote by $c : \mathcal{A}^\infty \to \mathcal{A}^\infty$ a projection function such that $\forall x_{-\infty}^0 \in \mathcal{A}^\infty, c(x_{-\infty}^0) := x_{-\ell+1}^0$, where ℓ is defined by

$$\begin{aligned}\ell = \ell(x_{-\infty}^0) = \min\{\, h \mid \forall x_1 \in \mathcal{A},\ P(X_1 = x_1 | X_{-\infty}^0 = x_{-\infty}^0) = \\ = P(X_1 = x_1 | X_{-h+1}^0 = x_{-h+1}^0)\},\end{aligned}$$

and $\ell = 0$ corresponds to independence. Then $c(.)$ is called a context function and for any $t \in \mathbb{Z}$, $c(x_{-\infty}^{t-1})$ is called the context for the process X_t at time t. Now let $0 \leq h \leq \infty$ be the smallest integer such that

$$\forall x_{-\infty}^0 \in \mathcal{A}^\infty, \quad |c(x_{-\infty}^0)| = \ell(x_{-\infty}^0) \leq h.$$

Then the context $c(.)$ is called the context function of order h, and if $h < \infty, \{X_t\}_{t\in\mathbb{Z}}$ is called a stationary *variable length process* of order h.

Clearly, a stationary variable length process of order h is a Markov chain of order h, and now having a memory of variable length ℓ. If the context function $c(.)$ of order h is the projection $x^0_{-\infty} \to x^0_{-h+1}$ for all $x^0_{-\infty}$, the process is a full Markov chain of order h. Here, the *context* refers to the portion of the past that influences the next outcome of transitions. The definition of ℓ implicitly reflects the fact that the context length of variable X_t is $\ell = \ell(x^{t-1}_{-\infty}) = |c(x^{t-1}_{-\infty})|$ depending on the history $X^{t-1}_{-\infty} = x^{t-1}_{-\infty}$. By the projection structure of the context function $c(.)$, the context length $\ell(.) = |c(.)|$ determines $c(.)$ and vice versa. In other words, the memory of stationary stochastic process is allowed to be of variable length, a function of the values from the past. That means the time homogeneous transition probabilities

$$P(X_t = x_t|X_{t-1} = x_{t-1}, X_{t-2} = x_{t-2}, \dots)$$

are functions that depend on a variable number l of values

$$P(X_t = x_t|X_{t-1} = x_{t-1}, X_{t-2} = x_{t-2}, \dots, X_{t-l} = x_{t-l}),$$

where $\ell = \ell(x_{t-1}, x_{t-2}, \dots)$ is itself a function of the past. If $\ell = \ell(x_{t-1}, x_{t-2}, \dots) = h$ we obtain the habitual stationary process of order h (full homogeneous Markov chain of order h). If the variable $\ell(.)$ with $\sup\{\ell(x_{t-1}, x_{t-2}, \dots); x_{t-1}, x_{t-2}, \dots\} = h$, we have stationary process with an additional structure of a *variable length memory* ℓ. We usually identify process $\{X_t\}_{t\in\mathbb{Z}}$ with its probability distribution of transition $P(X_1 = x_1|X^0_{-\infty} = x^0_{-\infty}) = P(x_1|c(x^0_{-\infty}))$ over $\mathcal{A}^{\mathbb{Z}}$, well-known as conditional probability of x_1 knowing the context $c(x^0_{-\infty})$, *i.e.*,

$$P(X_t = x_t|X_{t-1} = x_{t-1}, X_{t-2} = x_{t-2}, \dots) = P(x_t \mid c(x_{t-1}, x_{t-1}, \dots)).$$

The states determining the transition probability $P(x_1|c(x^0_{-\infty}))$ are given by the value of the context function $c(.)$. The most convenient is to represent these states by a minimal data structure. In order to represent the minimal state space of the variable length process, we use the structure of variable length finite automata.

14.4 Variable Length Finite Automata (VLFA)

14.4.1 *Structure of VLFA*

This section studies the structure of VLFA including probabilistic and counting automata. Structurally, a VLFA is an automaton without any loops and has a particular structure of the stochastic automata defined as Definition 14.9. Now we describe the structure of VLFA representing the joint probabilistic distribution of $\mathcal{L}$ and the class of families $\mathcal{C} = \{+, -\}$.

Definition 14.14. A VLFA is a 8-tuples $\mathfrak{A} = (\mathcal{A}, \mathcal{C}, \mathcal{Q}, P, T, B, I, F)$, where

- $\mathcal{A} = \{x_1, x_2, \ldots, x_k\}$ is an alphabet of size k; $\mathcal{C} = \{+, -\}$ is the set of family indicator (trait); $\mathcal{Q}$ is the set of states,
- $I : \mathcal{Q} \to [0, 1]$ is the probability of initial states,
- $F : \mathcal{Q} \to [0, 1]$ is the probability of final states,
- $T : \mathcal{Q} \times \mathcal{A} \to \mathcal{Q}$ is the transition function, $\forall x \in \mathcal{A}, \forall q \in \mathcal{Q}, T(q, x) = r \in \mathcal{Q}$,
- $P : \mathcal{Q} \times \mathcal{A} \to [0, 1]$ is a set of transition probabilities labelled by a symbol of $\mathcal{A}$,
- $B : \mathcal{Q} \times \mathcal{C} \to [0, 1]$ is a set of emission probabilities of class $\mathcal{C}$ over the transitions.

The applications I, F, T, P, B must satisfy

$$\begin{aligned}
&\sum_{q \in \mathcal{Q}} I(q) = 1, \quad \sum_{q \in Q} F(q) = 1, \\
&\forall q \in Q, T(q, x) \neq q, \forall q \in Q, \quad \sum_{x \in \mathcal{A}} P(q, x) = 1, \\
&\forall q \in Q, \sum_{c \in \mathcal{C}, x \in \mathcal{A}} B(q, c|x) = \begin{cases} 1 & \text{if } P(q, x) \neq 0, \\ 0 & \text{otherwise} . \end{cases}
\end{aligned}$$

For $s = x_{i_1} x_{i_2} \ldots x_{i_n} \in \mathcal{A}^*$ and $c \in \mathcal{C}$, the joint probability $P_{\mathfrak{A}}(s, c)$ is computed by:

$$P_{\mathfrak{A}}(s, c) = I(q_{i_0}) \prod_{h=1}^{n-1} P(q_{i_h}, x_{i_{h+1}}) B(q_{i_h}, c|x_{i_{h+1}}) F(q_{i_n}).$$

From this definition, the VLFA is an *additive model* that models both possitive language $\mathcal{L}^+$ (class $c = +$) and the negative language $\mathcal{L}^-$ (class $c = -$) over the transitions. Since VLFA is a PDFA, thus the time complexity of the exact computation of the probability $P_{\mathfrak{A}}(s, c)$ is in $\mathcal{O}(n)$. For $x \in \mathcal{A}$, the transition probability $P(q, x)$ labelled by the symbol x is only defined if there is the transition $T(q, x) = r$ from state q to state r. For $c \in \mathcal{C}$, the emission probability $B(q, c|x)$ of class c is defined when the transition $P(q, x)$ is defined. Thus, any path from the initial state $I(q_0)$ to a final state in $q_f \in F$ represents an individual string and it corresponding class. In reading VLFA, the symbols are generated on the transitions and the strings are generated along the path of transitions from the initial state to a final state. For $x \in \mathcal{A}$ and $q, r \in Q$, the transition $T(q, x) = r$ with high transition probability $P(q, x)$ indicates that states q and r conserve the same string and, alternatively, the low transition probability represents the recombination event between strings.

A state q is called "splitting state" when it has at least two outgoing transitions from this state. The transitions associated to a splitting state are called "splitting transitions" ("splitting edges"). A splitting state represents the distinguished string clusters. A combination of consecutive non-splitting states compose exactly the same string cluster. When two or more transitions income into a same state q, state q is called "incoming state" or *memory state*, the stochastic process VLFA is loss of memory (Markov's property) at state q. That means the history of state q represents a union of the histories represented by incoming transitions ("incoming edges"). This Markov's property models the historical recombination event between the strings. The string clusters change at point of splitting state and incoming state. Precisely, one supposes that an incoming state q represents a collection $\{x_t^1, x_t^2\}$ of two string clusters x_t^1 and x_t^2 at instance t. Let X_{t+1} be a random variable representing the sequences of symbol from $t+1$ to n. Thus,

$$\begin{aligned} P_{\mathfrak{A}}(X_{t+1} = x_{t+1} | X_t = x_t^1) &= P_{\mathfrak{A}}(X_{t+1} = x_{t+1} | X_t = x_t^2) \\ &= P_{\mathfrak{A}}(X_{t+1} = x_{t+1} | X_t = x_t^1, X_t = x_t^2). \end{aligned}$$

Meaning that the conditional probability of all sequences in the cluster x_{t+1} does not depend on the symbols at the previous positions given by two string clusters x_t^1 and x_t^2. Consequently, the label of a path of non-splitting states starting from a splitting state to an incoming state defines a common string pattern for which the elements of this pattern are in linkage dependence (closely related). Formally, if the model lost of memory at instance i, we have

$$\begin{aligned} &P_{\mathfrak{A}}(X_t = x_t | X_{t-1} = x_{t-1}, \ldots, X_i = x_i, X_{i+1} = x_{i+1}, \ldots, X_{i+h} = x_{i+h}) = \\ &= P_{\mathfrak{A}}(X_t = x_t | X_{t-1} = x_{t-1}, \ldots, X_{i+1} = x_{i+1}). \qquad (14.18) \end{aligned}$$

Example 14.15. Figure 14.5 represents a VLFA model that follows a variable length process. Let X_t be a random variable representing a symbol at position t. By definition, state 9 is the incoming state. Thus, the VLFA lost completely the memory at this state. State 9 is the memory state

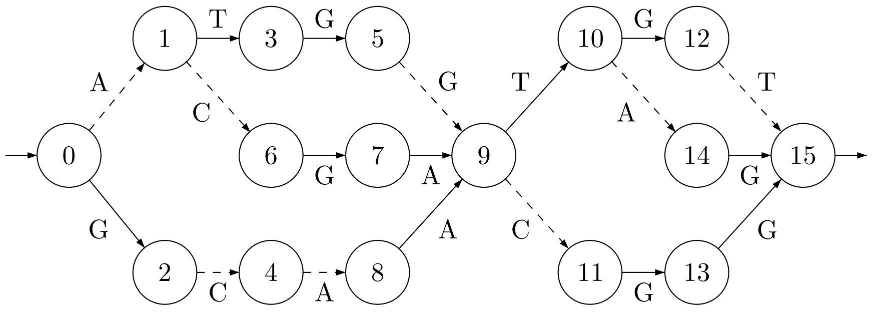


Fig. 14.5 An example of data struture of VLFA following a variable length process.

of the model and divides model into two independent subgraphs that correspond to two independent blocks. The length memory of first block is 4 while the second one is 3. Consequently, the strings are divided into two independent blocks of string segments, $\{ATGG, ACGA, GCAA\}$ and $\{TGT, TAG, CGG\}$ respectively. The substrings in each block define then the associated contiguous patterns. The variables in each pattern are closely linked to each other. Since the model lost of memory in instance $i = 4$ at state 9, according to (14.18) we have

$$\begin{aligned} P(X_7 = G|X_6 = G, X_5 = T, X_4 = G, \quad & \ldots, X_0 = A) = \\ & = P(X_7 = G|X_6 = G, X_5 = T) \end{aligned}$$

and

$$P(X_7 = G|X_6 = G, X_5 = T) \neq P(X_7 = G|X_6 = A, X_5 = T).$$

Thanks to Definitions 14.10, the algebra structure of weighted automata can be adapted for VLFA with their matrical representation. Let $\lambda \equiv I, \gamma \equiv F$ and $\mathbb{P} : \mathcal{A} \longrightarrow \mathcal{M}_{N,N}(\mathbb{K})$ be a morphism of monoid of $\mathcal{A}^*$ such that for each $a \in \mathcal{A}$, the transition matrix $\mathbb{P}(x)$ of the symbol x is defined by, for all $p, q \in Q, \mathbb{P}_{pq}(x) = P(p, x)$ if $T(p, x) = p \neq 0$. Consequently, the automaton $\mathfrak{A} = (\mathcal{A}, \mathcal{C}, \mathcal{Q}, P, B, I, F)$ is the represented by $(\lambda, \mathbb{P}, \gamma)$, and the appearance probability of a string s is computed by employing Theorem 14.12, $P_{\mathfrak{A}}(s) = \lambda \mathbb{P}_s \gamma$.

In practice, we use *Directed Acyclic Graph* (DAG) $\mathcal{G} = (V, E)$ to implement a VLFA, where V is a finite set of *vertices* representing the set of states Q, and E is a set of *edges* labelled by the symbol and its weight describing the triplet function $\{T, P, B\}$ of VLFA. A DAG represents a VLFA in the following manner. Each state $p \in \mathcal{V}$ represents the level of locus position and corresponding letter. At level 0, there is only one state, which does not itself contain information. For $i = 1, \ldots, n$, a state at level i represents a history or a collection of a subset of strings. For $p, q \in V$ and $x \in \mathcal{A}$, each edge $(p, x, q) \in E$ labelled by x originating from state p at level i and terminating at direct successor q at level $i + 1$ indicates the event of symbol x at position i following the history represented by state p. In order to facilitate the computation of probability and the test of statistic, we associate, for each edge $(p, x, q) \in \mathcal{E}$, the number of strings, number of strings in family $\mathcal{L}^+$ and number of strings in family $\mathcal{L}^-$: N_x, N_x^+ and N_x^-, respectively. The DAG plotted in Figure 14.6 represents the VLFA of the haplotype data given in Example 14.2 via the algorithm AcyclicAutomata described in the next section.

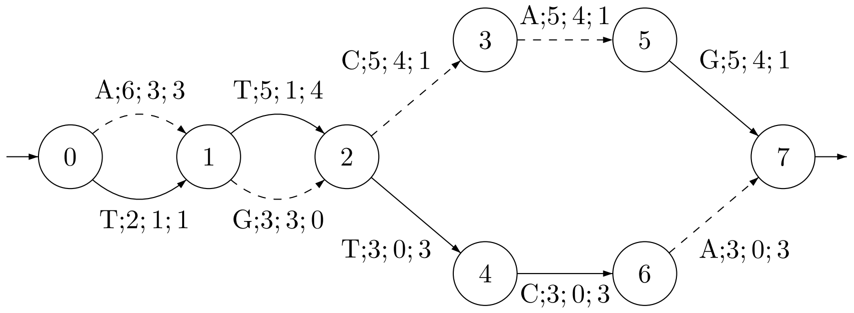


Fig. 14.6 VLFA and conditional independence.

14.4.2 *Fitting VLFA*

14.4.2.1 *Minimization of Finite Automata*

A given language $\mathcal{L} \subset \mathcal{A}^*$ can be recognized by different automata ($\forall s \in \mathcal{L}, P_{\mathfrak{A}}(s) \neq 0$). However, there is a unique automaton with a minimal number of states, called the *minimal automaton* of $\mathcal{L}$. We are always interested in learning algorithms to identify the minimal automaton. Here, we consider the following problem: for a given initial DFA (deterministic finite automaton), find an equivalent DFA with a minimum number of states.

The minimization algorithm identifying a minimum automaton is based on the notation of *distinguishability.* In general, given an automata $\mathfrak{A}$, a string $s \in \mathcal{A}^*$ distinguishes between two states $p, q \in \mathcal{Q}$ if either $T(p, s) \in F$ and $T(q, s) \notin F$, or $T(p, s) \notin F$ and $T(q, s) \in F$. Two states $p, q \in \mathcal{Q}$ are called distinguishable iff there is a string that distinguishes between them. States that are indistinguishable will also be called *equivalent.* This equivalence is based on the *Nerode equivalence* and defined by

$$\forall p, q \in \mathcal{Q}, \quad p \equiv q \text{ if and only if } \mathcal{L}_p = \mathcal{L}_q, \tag{14.19}$$

where $\mathcal{L}_p = \{s \in \mathcal{L} \mid T(p, s) \in F\}$ is the set of strings recognized by the automaton at state p. Based on this we can define the distinguishability of probabilistic automata as follows.

Definition 14.16. *Let $\mathfrak{A}$ be a probabilistic automaton and let $0 \leq \alpha \leq 1$ be a parameter. A pair of states p and q in $\mathcal{Q}$ is α-distinguishable if there exists a string u such that $|P_p(u) - P_q(u)| \geq \alpha$. The automaton $\mathfrak{A}$ is α-distinguishable if any pair of distinct states in $\mathfrak{A}$ is α-distinguishable.*

The distinguishability of an automaton leads to the following minimization method: Start with an initial automaton $\mathfrak{A}$. If $\mathfrak{A}$ has pair of indistinguishable (equivalent) states p, q, merge them into one state (remove q and reroute all transitions into q to go into p instead). Repeat this process until no more pair of indistinguishable states can be found. Any automaton whose all states are pairwise distinguishable must be minimum.

Lemma 14.17. *A DFA is minimum if and only if all pairs of states are distinguishable.*

Proof. Given automaton $\mathfrak{A}$, if $\mathfrak{A}$ has two indistinguishable states, one of them can be eliminated, and the transitions into this state can be changed to go to the other. Now, assume that in $\mathfrak{A}$ all states are pairwise distinguishable. Let $\mathfrak{A}$ have K states. Consider any other $\mathfrak{A}^*$ with $L < K$ states. We need to prove that $\mathcal{L}_{\mathfrak{A}^*} \neq \mathcal{L}_{\mathfrak{A}}$. In fact, for each state q of $\mathfrak{A}$, choose arbitrarily one string s_q such that $T(q_0, s_q) = q$ (with assumption that all states are reachable). Since $L < K$, there are two states $p \neq q$ of $\mathfrak{A}$ such that in $\mathfrak{A}^*$ we have $T^*(q_0, s_p) = T^*(q_0, s_q)$. Since p, q are distinguishable in $\mathfrak{A}$, there is a string $u \in \mathcal{A}^*$ such that $T(q, u) \in F$ but $T(p, u) \notin F$. This means that $\mathfrak{A}$ accepts $s_p u$ but not $s_q u$. But in $\mathfrak{A}^*$, we have $T^*(q_0, s_p u) = T^*(q_0, s_q u)$, so $\mathfrak{A}^*$ either accepts both $s_p u$ and $s_q u$ or rejects both. Therefore $\mathcal{L}_{\mathfrak{A}^*} \neq \mathcal{L}_{\mathfrak{A}}$. □

The early idea of minimization method discussed above is repeatedly merging pairwise indistinguishable states. However, we can do it with multi-indistinguishable states. Precisely, suppose that two states p, q are indistinguishable, and q, r are also indistinguishable. We want to combine p with q and q with r. Then, it is better if p and r are also indistinguishable. This property is formalized by the lemma below.

Lemma 14.18. *State indistinguishability is an equivalence relation.*

Proof. The following three conditions are trivial and directly checked from the definition of the indistinguishability relation (14.19): i) Symmetry: $p \equiv p$; ii) Reflexivity: $p \equiv q$ implies $q \equiv p$; iii) Transitivity: if $p \equiv q$ and $q \equiv r$ implies $\mathcal{L}_p = \mathcal{L}_q = \mathcal{L}_r$. Thus $p \equiv r$. □

Lemma 14.19. *For $u \in \mathcal{A}^*$, let $T(p, u) = p'$ and $T(q, u) = q'$. If p' and q' are distinguishable then p and q are.*

Proof. This is quite simple from the definition of the distinguishability: if p' and q' are distinguished by some string v, then p, q are distinguished by string uv. □

The minimization algorithm presented below is merging a set of equivalent (indistinguishable) states into a single state such that the new automaton keeps the same language. Given an initial automaton $\mathfrak{A}$, we will find all equivalence classes of the indistinguishability relation and join all states in each class into one state of the new automaton $\mathfrak{A}^*$. In general, the computation of minimal automaton consists of the three following steps:

(1) Choosing the appreciate criterion for the test of equivalence,
(2) Specifying the fitting model to optimize the minimization algorithm,
(3) Applying a specific minimization algorithm.

14.4.2.2 *Probabilistic Equivalence Criterion*

The basic of fitting acyclic automaton algorithm described below is how to find the equivalence criterion and to test the equivalence of two transition issued states. In term of probability, two states of VLFA model are defined to be equivalent if their outgoing transition probabilities are equal for every labelled symbol $x \in \mathcal{A}$ and their destination states (their descendants) of the two transitions for each symbol are also equivalent according to a recursive application of the same criterion. Formally,

$$\forall p, q \in \mathcal{Q}, \forall x \in \mathcal{A}, \quad p \equiv q \Longleftrightarrow \begin{cases} P(p, x) = P(q, x), \\ T(p, x) \equiv T(q, x). \end{cases} \tag{14.20}$$

Therefore, two states are compared by checking the equivalence of their transition probabilities, followed by a recursive checking process of any destination states reachable via transition symbols they have in common. That provides a criterion in order to reject equivalence of states. However, in practice, experimental data are subjected to statistical fluctuations and equivalence must be accepted within a confidence range. In such case, the states will be called compatible. To measure degree of compatibility, we use the Hoeffding's bound over binomial distribution defined as follows.

Definition 14.20. Let $\dfrac{f}{m}$ be the observed frequency of a Bernoulli variable of probability π. The confidence range for a Bernoulli variable with probability π and $\dfrac{f}{m}$ given by following bound:

$$P\Big(|\frac{f}{m} - \pi| < \sqrt{\frac{1}{2m}\log\frac{2}{\alpha}}\Big) > 1 - \alpha,$$

where α is a statistical significance level chosen as the cutoff probability.

Lemma 14.21. *Let* $\dfrac{f_1}{m_1}$ *and* $\dfrac{f_2}{m_2}$ *be observed frequencies of a Bernoulli variable with probability* p, *thus*

$$P\Big(|\frac{f_1}{m_1} - \frac{f_2}{m_2}| < \sqrt{\frac{1}{2}\log\frac{2}{\alpha}}(\frac{1}{\sqrt{m_1}} + \frac{1}{\sqrt{m_2}})\Big) > (1-\alpha)^2.$$

Proof. Let $\varepsilon_\alpha(m) = \sqrt{\dfrac{1}{2m}\log\dfrac{2}{\alpha}}$. Since two observed frequencies of the Bernoulli variable with probability π are independent, one has

$$\begin{aligned} P\Big(|\frac{f_1}{m_1} - \frac{f_2}{m_2} & | > \varepsilon_\alpha(m_1) + \varepsilon_\alpha(m_2)\Big) < \\ & < P\Big(|\frac{f_1}{m_1} - \pi| + |\frac{f_2}{m_2} - \pi| < \varepsilon_\alpha(m_1) + \varepsilon_\alpha(m_2)\Big) \\ & < P\Big(|\frac{f_1}{m_1} - \pi| < \varepsilon_\alpha(m_1) \cap |\frac{f_2}{m_2} - \pi| < \varepsilon_\alpha(m_2)\Big) \\ & < P\Big(|\frac{f_1}{m_1} - \pi| < \varepsilon_\alpha(m_1)\Big).P\Big(|\frac{f_2}{m_2} - \pi| < \varepsilon_\alpha(m_2)\Big) \\ & < (1-\alpha)^2. \end{aligned}$$

□

According to Lemmas 14.19 and 14.21, two states p and q of a VLFA are called α-compatible if $\forall x \in \mathcal{A}, \forall u \in \mathcal{A}^*$ and $0 < \alpha < 1$ one has

$$\begin{cases} (a)\ \Delta_{p,q}(x) = |\frac{N_p(x)}{N_p} - \frac{N_q(x)}{N_q}| < \sqrt{\frac{1}{2}\log\frac{2}{\alpha}}\Big(\frac{1}{\sqrt{N_p}} + \frac{1}{\sqrt{N_q}}\Big), \\ (b)\ \Delta_{p,q}(xu) < \sqrt{\frac{1}{2}\log\frac{2}{\alpha}}\Big(\frac{1}{\sqrt{N_p}} + \frac{1}{\sqrt{N_q}}\Big), \end{cases} \tag{14.21}$$

where, N_p is the entering frequency of state p and $N_p(x)$ is the transition frequency of symbol x outgoing from p. The constraint (b) means that $T(p, xu)$ and $T(q, xu)$ are also α-compatible satisfying Lemma 14.19. In practice, the procedure of equivalent test for any pair of states is based on the following standard statistical test:

(1) Formulate a null hypothesis (H_0) and alternative hypothesis (H_a) for the states of interest:
   - $H_0 : \pi_1 = \pi_2$ (two probabilities are the same)
   - $H_a : \pi_1 \neq \pi_2$ (two probabilities are different)

(2) Let N_p and N_q be the number of strings arriving at two states p and q. Let $N_p(x)$ and $N_q(x)$ be the number of strings outgoing from these states. Let α be a significance level chosen as the cutoff probability below which the hypothesis H_0 will be rejected.

(3) Two probabilities π_1 and π_2 represent the unknown true probabilities and will be estimated by $N_p(x)/N_p$ and $N_q(x)/N_q$, respectively. Assume that these can be modelled by binomial distribution $\mathcal{B}(x, m, p)$. The test statistic S is the absolute difference of observed proportions

$$S = |\frac{N_p(x)}{N_p} - \frac{N_q(x)}{N_q}|.$$

(4) Calculate the probability of a given value of S assuming H_0 is true. For two binomial distributions with identical probability, by Lemma 14.21,

$$S < \sqrt{\frac{1}{2}\log\frac{2}{\alpha}}\Big(\frac{1}{\sqrt{N_p}} + \frac{1}{\sqrt{N_q}}\Big) = K$$

with probability greater than $(1-\alpha)^2$. Since, we can not compute the exact value of p-value, but we can conclude that it is less than α if S is experimentally to be greater than K.

(5) If $S > K$, the statistic S has a p-value is smaller than α, we therefore reject the null hypothesis and decide two probabilities are different. Otherwise, the null hypothesis holds and we say that two probabilities π_1 and π_2 are equivalent, then two corresponding estimators $N_p(x)/N_p$ and $N_q(x)/N_q$ are equivalent.

Remark 14.22. The similarity of two frequencies can be measured by maximal standard deviation (MSD). In fact, when comparing $\frac{f_1}{m_1}$ and $\frac{f_2}{m_2}$, the variance of the difference $\Delta(\frac{f_1}{m_1}, \frac{f_2}{m_2}) = |\frac{f_1}{m_1} - \frac{f_2}{m_2}|$ is the sum of the variance of $\frac{f_1}{m_1}$ and of $\frac{f_2}{m_2}$. Since both $\frac{f_1}{m_1}$ and $\frac{f_2}{m_2}$ variables follow Bernoulli's law with probability π_1 and π_2 respectively, this variance is maximized when $\pi_1 = \pi_2 = 0.5$. Therefore,

$$\begin{aligned}\mathrm{Var}(\Delta(\frac{f_1}{m_1}, \frac{f_2}{m_2})) &= \mathrm{Var}(\frac{f_1}{m_1}) + \mathrm{Var}(\frac{f_2}{m_2}) \\ &= \frac{\pi_1(1-\pi_1)}{m_1} + \frac{\pi_2(1-\pi_2)}{m_2} \\ &\leq \frac{0.5(1-0.5)}{m_1} + \frac{0.5(1-0.5)}{m_2} \\ &= \frac{1}{4(m_1+m_2)}.\end{aligned}$$

The MSD of $\Delta(\frac{f_1}{m_1}, \frac{f_2}{m_2})$ is $\eta = 0.5(m_1^{-1} + m_2^{-1})^{\frac{1}{2}}$. Thus $\frac{f_1}{m_1}$ is similar with $\frac{f_2}{m_2}$ if $\Delta(\frac{f_1}{m_1}, \frac{f_2}{m_2})$ is less than cutoff η. Based on this, two states p and q are MSD-compatible if

$$\begin{cases}(a)\ \Delta_{p,q}(x) = |\frac{N_p(x)}{N_p} - \frac{N_q(x)}{N_q}| \leq 0.5(N_p^{-1} + N_q^{-1})^{\frac{1}{2}}, \\ (b)\ \Delta_{p,q}(xu) \leq 0.5(N_p^{-1} + N_q^{-1})^{\frac{1}{2}}, \quad \forall u \in \mathcal{A}^*.\end{cases} \tag{14.22}$$

An other similarity measure based on fixed cutoff value is used in [18]. Two states p and q are μ-compatible if

$$\Delta_{p,q}(x) = |\frac{N_p(x)}{N_p} - \frac{N_q(x)}{N_q}| \leq \frac{\mu}{2}, \quad \mu \leq 0. \tag{14.23}$$

14.4.2.3 *Fitting Model*

In general, learning a probabilistic automaton aims at inducing an automaton generating a distribution $\widehat{P}$ from a sample drawn according to some unknown target distribution P. The distribution $\widehat{P}$ forms the hypothesis that approximates the target probabilistic automaton. The purpose of a learning model is to formalize the notion of learning when a specific quality measure defines the distance between P and $\widehat{P}$ [18].

Definition 14.23. Let $\mathfrak{A}$ be the target PDFA and let $\widehat{\mathfrak{A}}$ be a hypothesis PDFA produced by a learning algorithm. Let $P_{\mathfrak{A}}$ and $P_{\widehat{\mathfrak{A}}}$ be the two probabilistic distributions they generate respectively. We say that $\widehat{\mathfrak{A}}$ is an ε-*good* hypothesis with respect to $\mathfrak{A}$, for $\varepsilon \geq 0$, if $D(P_{\mathfrak{A}}, P_{\widehat{\mathfrak{A}}}) \leq \varepsilon$.

In practice, we usually use the Kullback-Leibler divergence as the distance measure between the distributions $P_{\mathfrak{A}}$ and $P_{\widehat{\mathfrak{A}}}$:

$$D_{KL}(P_{\mathfrak{A}}, P_{\widehat{\mathfrak{A}}}) := \sum_{s \in \mathcal{L}} P_{\mathfrak{A}}(s) \log \frac{P_{\mathfrak{A}}(s)}{P_{\widehat{\mathfrak{A}}}(s)}. \tag{14.24}$$

The divergence can be interpreted as the number of additional bits needed to encode a message when an optimal code is chosen according to distribution $P_{\widehat{\mathfrak{A}}}$ while the message was produced according to distribution $P_{\mathfrak{A}}$.

Remark 14.24. The Kullback-Leibler divergence bounds the L_1 distance:

$$2\ln 2\sqrt{D_{KL}(P_{\mathfrak{A}}, P_{\widehat{\mathfrak{A}}})} \geq L_1(P_{\mathfrak{A}}, P_{\widehat{\mathfrak{A}}}) := \sum_{s\in\mathcal{L}} |P_{\mathfrak{A}}(s) - P_{\widehat{\mathfrak{A}}}(s)| \quad (14.25)$$

and the Hellinger distance as well

$$D_{KL}(P_{\mathfrak{A}}, P_{\widehat{\mathfrak{A}}}) \geq D_H(P_{\mathfrak{A}}, P_{\widehat{\mathfrak{A}}}) := \sum_{s\in\mathcal{L}} |\sqrt{P_{\mathfrak{A}}(s)} - \sqrt{P_{\widehat{\mathfrak{A}}}(s)}|^2. \quad (14.26)$$

Based on ε-*good* hypothesis property and divergence information, we use a learning algorithm that minimizes the divergence from the training sample distribution and the size of PDFA [9]. On one hand, this algorithm bases on maximum likelihood model built from the learning samples. On the other hand, favouring small automata, or equivalently automata derived from PDFAs with a large number of merging operations, corresponds to an increased prior probability associated to a reduced automaton size. Assume $\mathfrak{A}_i$ is an ε-*good* hypothesis PDFA produced by a learning algorithm at instant i and $\mathfrak{A}_{i+1}$ is a tentative new ε-*good* hypothesis that can be derived from $\mathfrak{A}_i$. In other words, $\mathfrak{A}_{i+1}$ can be obtained from $\mathfrak{A}_i$ by merging some candidate pair of states. The $\mathfrak{A}_{i+1}$ is the new temporary solution if the divergence increment relative to the size reduction:

$$\frac{D_{KL}(P_{\mathfrak{A}}, P_{\mathfrak{A}_i}) - D_{KL}(P_{\mathfrak{A}}, P_{\mathfrak{A}_{i+1}})}{|\mathfrak{A}_i| - |\mathfrak{A}_{i+1}|} < \theta, \quad \forall\theta > 0. \quad (14.27)$$

where $|\mathfrak{A}|$ denotes the size of an automaton $\mathfrak{A}$. Given a *confidence* parameter $0 \leq \alpha \leq 1$, and an *accuracy* (*precision*) parameter $0 \leq \varepsilon \leq 1$, the learning algorithm outputs, with probability at least $1 - \alpha$, an ε-*good* hypothesis PDFA $\widehat{\mathfrak{A}}$ with respect to target PDFA $\mathfrak{A}$.

14.4.2.4 *Minimization Algorithm*

Here we employ the methods proposed in [5, 18] to fit a VLFA. Given a training data set of strings and trait statuses, the algorithm starts by constructing a weighted prefix tree [24, 25]. Starting from the root, for each level (depth) of the tree, the algorithm tests if pairs of states, which corresponds to a large enough number of prefixes of strings, can be merged into a state. Two states belonging to the same level are eventually merged if the transition probabilities corresponding to all descendant states are statistically similar. In the state merging algorithm, the primitive operation of merging two states is to replace them into a single state. Merging of states corresponds to recognizing memory in the model and conditional dependence (Markov properties) between variables.

The main pseudo-code of learning algorithm is described in ACYCLICAUTOMATA. This algorithm produces deterministic acyclic weighted automata. In the course algorithm, a series of directed acyclic leveled graphs, $\mathfrak{A}_0(\mathcal{L}), \mathfrak{A}_1(\mathcal{L}), \mathfrak{A}_2(\mathcal{L}), \ldots, \mathfrak{A}_n(\mathcal{L})$ are built such that

$$|\mathfrak{A}_n(\mathcal{L})| \leq |\mathfrak{A}_{n-1}(\mathcal{L})| \leq \cdots \leq |\mathfrak{A}_1(\mathcal{L})| \leq |\mathfrak{A}_0(\mathcal{L})|$$

where the initial graph $\mathfrak{A}_0(\mathcal{L})$ is the weighted prefix tree $\mathfrak{T}(\mathcal{L})$ [24]. The final graph $\mathfrak{A}_n(\mathcal{L})$ is the target weighted automaton and has smallest number of states. Given the multiset of strings $\mathcal{L}$, algorithm ACYCLICAUTOMATA starts by taking successively the string and their class status and putting them into a weighted prefix tree, $\mathfrak{A}_0$. Each edge of $\mathfrak{A}_0$ represents then the number of strings, the number of strings in family $\mathcal{L}^+$ and the number of strings in family $\mathcal{L}^-$, $N_p(x), N_p^+(x)$ and $N_p^-(x)$ [26]. The data structure used along the run of the algorithm is only the current graph $\mathfrak{A}_i$ with the string counts on the edge of transition. For $i = 0, \ldots, n$, we associate with $\mathfrak{A}_i$ a depth (level) $d(i)$ ($d(0) = 1$ and $d(i) < d(i+1)$). The $\mathfrak{A}_i$ will be transformed to $\mathfrak{A}_{i+1}$ at the depth $d(i)$ by the call of the merging operation. Beginning from the root of weighted prefix tree and working down the levels, for each level $d(i)$, the algorithm searches for pair of states p and q which can be merged. Suppose that all pair of states at level $d(i-1)$ are merged and the obtained automaton is $\mathfrak{A}_i$. Algorithm checks the similarity of p and q at the level $d(i)$ by calling the subroutine COMPATIBLE$(p, q, \mathfrak{A}_i)$. If COMPATIBLE$(p, q, \mathfrak{A}_i)$ returns TRUE then the algorithm merges p and q in graph $\mathfrak{A}_i$ using the subroutine MERGE$(p, q, \mathfrak{A}_i)$. When the final depth $d(n)$ is reached, the target graph $\mathfrak{A}_n$ is obtained from $\mathfrak{A}_{n-1}$ by merging all final states into one final states.

The function COMPATIBLE$(p, q, \mathfrak{A}_i)$ is implemented by recursive function. It recursively calls DIFFERENT$(N_p, N_p(x), N_q, N_q(x))$ for all descendent pairs of children nodes of p and q and returns TRUE if and only if pair of states p and q is α-compatible (sufficiently similar) for all their downstream pair of children states, verifying formula (14.21). In the case if for some pair of state p and q are not α-compatible, the COMPATIBLE returns FALSE. Thus, two states p and q are not merged.

The subroutine MERGE$(p, q, \mathfrak{A}_i)$ is called if and only if the COMPATIBLE$(p, q, \mathfrak{A})$ returns TRUE. That merges two compatible states p

```
AcyclicAutomata($\mathcal{L}$)
 1: Input: $\mathcal{L}$, $0 \leq \alpha \leq 1$;
 2: Output: Acyclic Deterministic Finite Automaton;
 3: Initialize $i \leftarrow 0$; $\mathfrak{A}_0 \leftarrow \mathfrak{T}(\mathcal{L})$; $d(0) \leftarrow 1$;
 4: while $(d(i) < n)$ do
 5:    for each pair of states $(p, q)$ from level $d(i)$ in $\mathfrak{A}_i$ do
 6:       if $\big($Compatible$(p, q, \mathfrak{A}_i)\big)$ then
 7:          Call Merge$(p, q, \mathfrak{A}_i)$;
 8:          $\mathfrak{A}_{i+1} \leftarrow \mathfrak{A}_i$;
 9:          Renumber the states of $\mathfrak{A}_{i+1}$ in range $1, \ldots, |\mathfrak{A}_{i+1}|$;
10:          $d(i+1) \leftarrow d(i)$; $i$++;
11:       else
12:          $d(i) \leftarrow d(i) + 1$; $i$++;
13:       end if
14:    end for
15: end while
```

```
Compatible$(p, q, \mathfrak{A})$
 1: Input: $p, q \in \mathcal{Q}$;
 2: Output: Boolean;
 3: if $\big($Different$(N_p, F(p), N_q, F(q))\big)$ then
 4:    return false;
 5: end if
 6: for $\forall x \in \mathcal{A}$ do
 7:    if $\big($Different$(N_p, N_p(x), N_q, N_q(x))\big)$ then
 8:       return false;
 9:    end if
10:    if $\big($not Compatible$(T(p, x), T(q, x))\big)$ then
11:       return false;
12:    end if
13: end for
14: return true;
```

and q into new state p and recursively their children states in order to eliminate non-determinism in the underling model. It returns then a new smaller graph, $|\mathfrak{A}_{i+1}| \leq |\mathfrak{A}_i|$. Merge$(p, q, \mathfrak{A}_i)$ respectively updates all information, the number of strings, number of strings in family $\mathcal{L}^+$ and number of strings

```
DIFFERENT($N_p, N_p(x), N_q, N_q(x)$)
 1: **Input**:
 2:    $0 \leq \alpha \leq 1$; $N_p, N_q$: number of strings arriving at each node $p$ and $q$;
 3:    $N_p(x), N_q(x)$: number of strings outgoing at each node $p$ and $q$;
 4: **Output**: Boolean;
 5: **if** $\big(|\frac{N_p(x)}{N_p} - \frac{N_q(x)}{N_q}| > \sqrt{\frac{1}{2}\log\frac{2}{\alpha}}\big(\frac{1}{\sqrt{N_p}} + \frac{1}{\sqrt{N_q}}\big)\big)$ **then**
 6:    return false;
 7: **end if**
```

```
MERGE($p, q, \mathfrak{A}$)
 1: **Input**: $p, q, \mathfrak{A}$;
 2: **Output**: Smaller $\mathfrak{A}$;
 3: **for** all $(s, x) \in \mathrm{Pred}[q]$, $\forall s \in \mathcal{Q}, x \in \mathcal{A}$ **do**
 4:    Set $(s, x) \in \mathrm{Pred}[p]$;
 5:    $N_p(x) \leftarrow N_q(x) + N_p(x)$;
 6:    $N_p^+(x) \leftarrow N_q^+(x) + N_p^+(x)$;
 7:    $N_p^-(x) \leftarrow N_q^-(x) + N_p^-(x)$;
 8: **end for**
 9: **for** $\forall s, s' \in \mathcal{Q}$ such that $\forall x \in \mathcal{A}$, $(x, s) \in \mathrm{Succ}[p]$ and $(x, s') \in \mathrm{Succ}[q]$
    **do**
10:    Call MERGE($s, s', \mathfrak{A}$);
11: **end for**
12: $\mathfrak{A} \leftarrow \mathfrak{A} - \{q\}$;
```

in family $\mathcal{L}^-$, such that $N_p(x) = N_q(x) + N_p(x), N_p^+(x) = N_q^+(x) + N_p^+(x)$ and $N_p^-(x) = N_q^-(x) + N_p^-(x)$.

Remark 14.25. The cutoff coefficients in α-compatible and MSD-compatible for merging algorithm is a function of the string counts rather than a fixed value used in [18]. Since a state p having a small string count will have high variability in the observed conditional probability $N_p(x)/N_p$ and thus it will be to have a score less than a fixed cutoff ($\mu/2$ used in [18]) for a state q, even if p and q represent the same conditional probability distribution. Thus, that guarantees that the low frequency strings are continually merged into the graph. While the use of μ-compatible with fixed value cutoff μ has some limits. If μ cutoff is large then only high frequency strings are merged.

By this construction, the memory of VLFA models is allowed to be of variable length in which the length of memory of the process depends on the context. In the region the variables are closely dependent linkage, the VLFA models will have a longer memory, whereas, in the regions of low dependent linkage, the memory of the VLFA will be short. In particular, if the variables are independent, the target automaton has a structure of the homogeneous Markov chain. Thus, the class of proposed models is structurally richer than existing models. It includes and generates the class of stationary Markov chains. This class of automata is called variable-length finite automata.

Proposition 14.26. *The overall time complexity of the algorithm* AcycliCAutomata *is in* $\mathcal{O}(nm^2)$.

Proof. The complexity of the algorithm AcyclicAutomata depends on the number of comparisons conducted by subroutine Compatible for testing the similarity between states at each level of prefix trees. If we let t be the number of states at a level of tree then requiring $t(t-1)/2$ pairs of states be compared, *i.e.* the number of comparisons is in $\mathcal{O}(t^2)$. Since the number of states per level is bounded by the number of strings m, thus the maximal number of comparison at each level is in $\mathcal{O}(m^2)$; and furthermore, the recursive comparison continues to $\mathcal{O}(n)$ level. In worse-case the time complexity of the algorithm is $\mathcal{O}(nm^2)$. □

The computation of target VLFA depends on the parameter α in the expression α-compatible (14.21). The identification in the limit of a VLFA takes place if the function Compatible behaves in the limit of large m of strings [5]. For this purpose, we allow the parameter α depending on the size of the prefix tree (initial automaton $\mathfrak{A}_0$), $t = |\mathfrak{A}_0|$. Therefore,

Proposition 14.27. *The minimization algorithm* AcyclicAutomata *returns a target VLFA in the limit of large m if*

$$\lim_{m\to\infty} (1-\alpha(t))^t \to 1.$$

Proof. In fact, the subroutine Compatible involves at most $t = |\mathfrak{A}_0|$ iterations, then the subroutine Different is called $(1+\mathcal{A})t$ times. Thus, following the formula (14.21), Compatible returns the correct value with the probability greater than $(1-\alpha(t))^{2t}$. The condition $(1-\alpha(t))^t \to 1$ for large m signifies that $\alpha(t)$ decreases faster than $1/t$. That means one takes $\alpha(t) = \gamma/t$, with $0 < \gamma < 1$ is a small constant. The $\alpha(t)$ converges to 0 when m increases to infinite. □

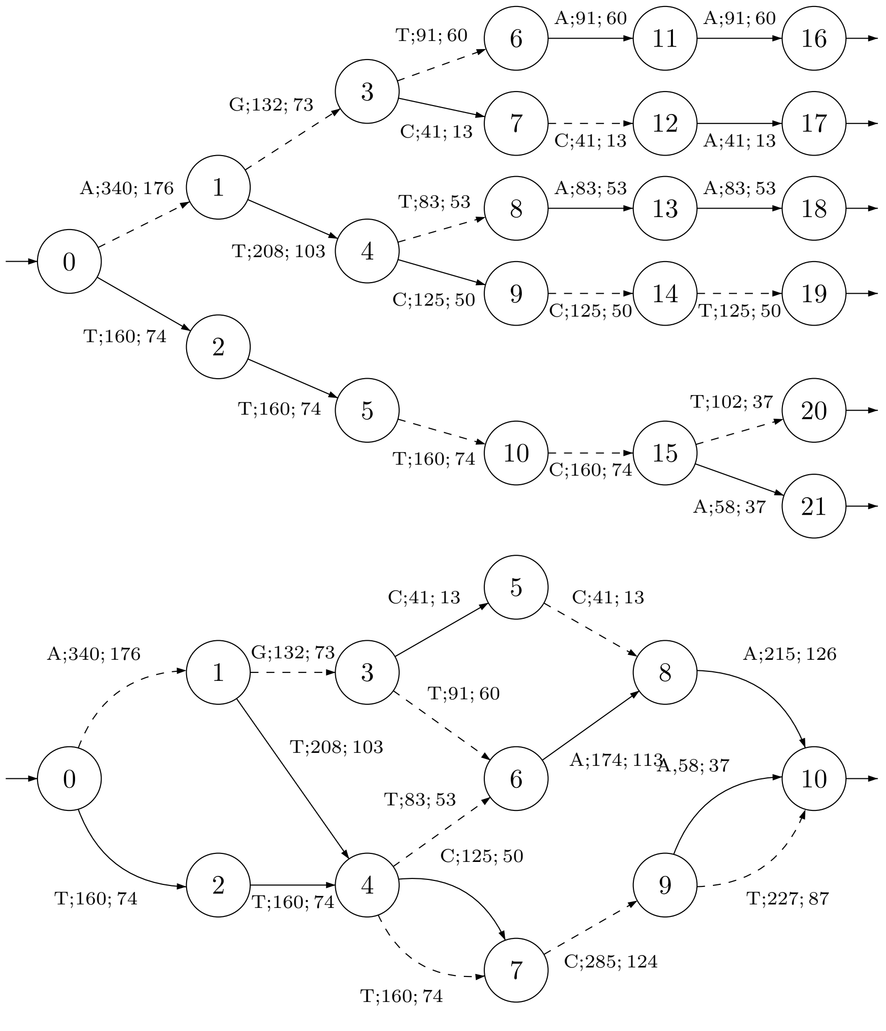


Fig. 14.7 An example of acyclic automaton learning.

Example 14.28. Consider the haplotype data represented in Table 14.3, which consists of 250 case and 250 control haplotypes, on five bi-allelic SNP marker loci and their trait. We assume that the first alleles at the loci are A, G, T, C and T, respectively; and the second alleles are T, T, C, A, and A. To optimize the implementation, the first alleles at a marker is coded by symbol 1 and the second one is coded by symbol 2. A haplotype is

Table 14.3 Haplotype data on five SNP marker loci and their trait.

Haplotype	Coded String	$\lvert\mathcal{L}^+\rvert$	$\lvert\mathcal{L}^-\rvert$	$\lvert\mathcal{L}\rvert$
AGCCA	11212	13	28	41
AGTAA	11122	60	31	91
ATTAA	12122	53	30	83
ATCCT	12211	50	75	125
TTTCT	22111	37	65	102
TTTCA	22112	37	21	58
		250	250	500

then represented by a binary string over alphabet $\mathcal{A} = \{1, 2\}$. The first graph of Figure 14.7 is the weighted prefix tree associated with the coded strings of haplotype data in table 14.3. Each path from the root (state 0) to a terminal state of the tree represents a distinct haplotype string. A dashed edge between states at level i and $i + 1$ represents symbol 1; a solid edge represents symbol 2. The triplet $\{x; N_x; N_x^+\}$ on the edge indicates the following information. The first element x is the real allele, the next number N_x is haplotype count and the last N_x^+ indicates the case haplotype count. Note that $N_x^- = N_x - N_x^+$. Thus, the edge $(1, 4)$ represents a cluster of 208 haplotypes which 103 are cases haplotypes that have allele 1 at first SNP and allele 2 at the second SNP. The second graph is the VLFA model obtained by the merging algorithm AcyclicAutomata from weighted prefix tree with $\alpha = 0.001$. The pairs of states 4 and 5 at level 3, 6 and 8 at level 4 of tree have been respectively merged into state 4 and state 6 in VLFA, etc. The decision to merge these pairs of states also results in the children states merging. Hence, the merge of 6 and 8 results the merge of the pair 11 and 13. The VLFA model lost the memory at the states 4, 6, 7 and 8. From these memories, we can determine the block of haplotype. For example, two blocks of haplotype associated to state 4 are $\mathcal{P}_1 = \{AT, TT\}$ and $\mathcal{P}_2 = \{CCA, CCT, TCA, ACT\}$.

14.4.3 *Applications of VLFA*

14.4.3.1 *Decision and Classification*

Thank to VLFA stochastic processes, we will solve several following problems for gene mapping analysis. Firstly, the VLFA gives a tool for representing enormous mass of string data. Then, it provides efficient model to result the prototype inference problem. We propose an algorithm which returns a string having the maximal joint probability for extracting *prototype string* or *most protective string* of the training data. Starting from the root of the VLFA and descending sequentially, at each state p, the choice of symbol a corresponding to the maximum product of $P(p, a, q)B(p, c = +, q|a)$ will be used to determine the paths to follow of states which gives the string having the maximal joint probability.

Theorem 14.29. *Let* $\mathfrak{A}$ *be the VLFA. A* positive prototype *or* frequent positive string *of the training data* $\mathcal{L}$ *is given by*

$$\widehat{s} = \bigcup_{\widehat{p},\widehat{q}\in\mathcal{Q},\widehat{a}\in\mathfrak{A}} (\widehat{p}, \widehat{a}, \widehat{q}) = \bigcup_{p,q\in\mathcal{Q},a\in\mathcal{A}} \arg\max_{p,q\in\mathcal{Q},a\in\mathcal{A}} P(p, a, q)B(p, c = +, q|a).$$

Proposition 14.30. *Searching for a prototype has the time complexity in* $\mathcal{O}(kn)$*.*

On the other hand, the recognition and the classification or prediction problem are fundamental issues in data mining. The prediction procedure consists of two steps. The first step is the recognition of a new individual string to verify its existence in the model. The second step is to predict, if it exists in the model, which class it will be classified. In recognition, a string s is recognized by a VLFA $\mathfrak{A} = (\mathcal{G}, \mathcal{C}, \mathcal{Q}, P, B, I, F)$ if and only if it labels a path going from the initial state to a final state. Therefore,

Theorem 14.31. *A given string* $s = x_{i_1}x_{i_2}\ldots x_{i_n}$ *is recognized by* $\mathfrak{A}$ *if and only if*

$$P_{\mathfrak{A}}(s) = I(q_0)\prod_{h=1}^{n-1} P(q_h, x_{i_h})F(q_n) = \lambda\mathbb{P}_s\gamma \neq 0.$$

Proposition 14.32. *Seeking a path in VLFA* $\mathfrak{A}$ *corresponding to the string* $s = x_{i_1}x_{i_2}\ldots x_{i_n}$ *is in time* $\mathcal{O}(n)$*.*

In classification, the goal of a learning algorithm is to construct a classifier given a training data set. The VLFA provides a classifier permitting to predict if an individual string will to be in a set of risk (positive) strings. Therefore,

Theorem 14.33. *(Decisional Theorem)Let* $\mathfrak{A}$ *be the VLFA. An individual string* s *is classified as the class* $c = +$ *(positive string family) if and only if*

$$\frac{P_{\mathfrak{A}}(s, c = +)}{P_{\mathfrak{A}}(s, c = -)} > 1.$$

Proof. A classifier is a function that assigns a class label to a string. According to Bayes's rule, the probability of a string s being class $c \in C$ is:

$$P(c|s) = \frac{P_{\mathfrak{A}}(s|c)P(c)}{P_{\mathfrak{A}}(s)}.$$

Thus, the string s is classified into the class $c = +$ if and only if

$$f_B(s) = \frac{P_{\mathfrak{A}}(c=+|s)}{P_{\mathfrak{A}}(c=-|s)} = \frac{P_{\mathfrak{A}}(s|c=+)P(c=+)}{P_{\mathfrak{A}}(s|c=-)P(c=-)} > 1, \tag{14.28}$$

where $f_B(s)$ is called a Bayesian classifier. □

Theorem 14.34. *(Decisional Theorem)Suppose that $\widehat{s}^{(+)}$ and $\widehat{s}^{(-)}$ are respectively two optimal prototypes of positive language $\mathcal{L}^+$ (class $c = +$) and the negative language $\mathcal{L}^-$ (class $c = -$). Thank to Definition 14.6 and 14.7, a string s is classified into the class $c = +$ if and only if*

$$d_H(s, \widehat{s}^{(+)}) < d_H(s, \widehat{s}^{(-)}) \text{ or } d_S(s, \widehat{s}^{(+)}) > d_S(s, \widehat{s}^{(-)}).$$

Example 14.35. Given the probabilistic automaton in Figure 14.7. By Theorem 14.29, the path of states $1 \to 3 \to 6 \to 8 \to 10$ constituting the string **1122** represents the most risk string with the maximal probability. And thank to Theorem 14.33, this string is to be in the set of risk strings because $P_{\mathfrak{A}}(1122, c = +) > P_{\mathfrak{A}}(1122, c = -)$.

14.4.3.2 *Cluster Association Test*

Using VLFA model, we can test all edges of tree for the association with trait status. However, not every edge of the VLFA needs to be tested. In fact, an exhaustive search over the full tree is not computationally efficient and, more importantly, involves excessively large numbers of comparisons, which would make the procedure less powerful. Moreover, a state that is not splitting represents exactly the same or similar string cluster as one or more other states and tree has many non "splitting states". Since string clusters change at points of splitting in the VLFA, we test only the "splitting states". In addition, we search only candidate string clusters associated with positive strings $\mathcal{L}^+$, so only clusters that have the positive count greater than negative count (*i.e.* the edges that verify $m_{AP} > m_{CP}$) will be tested.

The pseudo code of the algorithm of cluster association is given in ClusterAssociation. Denote Succ$[p] = \{(x, q)\}_{x \in \mathcal{A}}$ the set of k labelled direct successors corresponding k symbols of $\mathcal{A}$ at state p. The procedure for finding candidate string clusters associated with disease consists of three steps:

```
ClusterAssociation($\mathfrak{A}(\mathcal{L}), \beta$)
 1: **Input**: a $\mathfrak{A}(\mathcal{L})$, $\beta > 0$;
 2: **for** each state $p$ of VLFA $\mathfrak{A}$ **do**
 3:    **if** (Card(Succ[$p$]) $\geq 2$) **then**
 4:       **for** each successor $(x, q)$ of $p$ **do**
 5:          **if** ($m_{AP} > m_{CP}$) **then**
 6:             compute $\pm\chi^2(x, q)$ score by (14.7);
 7:             **if** ($\pm\chi^2(x, q) \geq \beta$) **then**
 8:                return cluster corresponding to $(x, q)$;
 9:             **end if**
10:          **end if**
11:       **end for**
12:    **end if**
13: **end for**
14: **Output**: Strong positive clusters;
```

(i) First, a set of candidate states satisfying Card(Succ[p]) ≥ 2 (*i.e.* splitting states) is identified through the search algorithm.

(ii) Then, for each selected candidate p, if it exists a successor (x, q) of p that verifies the constraint $m_{AP} > m_{CP}$, we compare the string frequencies between positive class $\mathcal{L}^+$ and negative class $\mathcal{L}^-$ using the degree of association $\pm\chi^2(x, q)$ given by formula (14.7) or the Pearson's chi-square test.

(iii) Finally, clusters with $\pm\chi^2(x, q)$ score (or P-value satisfies a threshold significance level) is greater than the threshold β will be selected as candidate clusters.

Proposition 14.36. *The time complexity of* ClusterAssociation *is bounded by* $\mathcal{O}(mn)$.

Example 14.37. With VLFA model illustrated in Figure 14.7, the haplotype clusters change at the splitting states 1, 3 and 4. Thus, the splitting edges $(1, 3)$, $(3, 6)$ and $(4, 6)$ of these states will be tested. For example, the edge $(3, 6)$ corresponds to the haplotype cluster of the form ****1****, where * is an arbitrary symbol (allele), with the 60 case haplotypic strings and 31 control haplotype strings, compared with 190 case and 219 control haplotypes not on this edge, which gives the Chi-score $\pm\chi^2 = 10.5$ and P-value $= 0.0012$ with Chi-square's test. This cluster may be associated with the family of positive strings $\mathcal{L}^+$.

14.4.3.3 *Pattern Discovery*

The essential application of VLFA is the search for maximal patterns in a set of strings corresponding to problem (14.8). Concerning the linkage constraint of these problems, the maximal patterns will be identified within the independent blocks of substrings via the fitted memory of the model. Suppose that in the ideal case the graph $\mathcal{G}$ is decomposed into a set of independent subgraphs via the set of memory states of VLFA model. Let $\mathcal{G} = \{\mathcal{G}_1, \dots, \mathcal{G}_K\}$ refers a set of K independent subgraphs decomposed from VLFA $\mathfrak{A}$ defining a set of corresponding (independent) blocks of substrings, where $\mathcal{G}_i = (V_i, E_i)$ and $\forall \mathcal{G}_i, \mathcal{G}_j \in \mathcal{G}, E_i \cap E_j = \emptyset, V_i \cap V_j = \{q\}$ with q is the memory state of VLFA $\mathfrak{A}$. Given a block $\mathcal{G}_i \in \mathcal{G}$, the size of patterns defined by formula 14.4 is characterized as follows.

Definition 14.38. Given a block $\mathcal{G}_i \in \mathcal{G}$. Let u be the substring representing the pattern $X_{I,J}$ in G_i. According to the definition 14.3, the size of $X_{I,J}$ is defined as:

$$f(X_{I,J}) := |I| + |J| = f(u) := |u| + N_u.$$

This is used to quantify the maximal patterns in each subgraph of VLFA. However, since the substrings in each block $\mathcal{G}_i$ have the same length, the objective function f is maximized when substring u has a maximal number of occurrences,

$$\widehat{f}(u) = |u| + \max_{u \in \mathcal{G}_i} N_u. \tag{14.29}$$

In other words, for each $\mathcal{G}_i \in \mathcal{G}$, the optimal solution of (14.8), for which the objective function is maximized, is

$$u = \operatorname*{argmax}_{u \in \mathcal{G}_i} \widehat{f}(u). \tag{14.30}$$

In practice, we do not use the graph decomposition because the decomposition of target graph of VLFA model is sometimes difficult. We therefore propose two following searching algorithms that employ the path having maximal frequency in VLFA and the memory states of the model. These algorithms allow us to identify efficiently the optimal solution for (14.8).

Finding Frequency Linkage Pattern. One of advantages of VLFA model is fast search using the maximal path in the model. We propose an algorithm in linear time for the problem of frequent pattern. Denote $\text{Pred}[q] = \{(p, x)\}_{x \in \mathcal{A}}$ the implementation of the set of k direct predecessors of state p. The algorithm begins from the initial state, following the top to down of VLFA, and successively employs the probable path (path having maximum frequency). For each internal state p satisfying the constraint $\max_{(x,p)\in \text{Succ}[p]} N_p(x)$, the algorithm checks property decay of memory of the model at the successor state of p, $q = T(p, x)$:

(i) if the state q is not the memory of the model, $\text{Card}(\text{Pred}[q]) = 1$ (there is only one incoming edge into q), the state q represents then the same cluster of strings as p from the root, the algorithm iteratively continues by executing the next state,

(ii) if the state q decays the memory of model, $\text{Card}(\text{Pred}[q]) \geq 2$ (q has least two direct predecessors incoming into itself), VLFA model is lost the memory at q. Then, algorithm returns the prefix u associated with q with the size $\widehat{f}(u) = |u| + N_u$ that is the maximal linkage pattern with high linkage disequilibrium verifying linkage constraint of problem (14.8), where the variables in this pattern are closely related. Then, input the pattern $M = \big(u, \widehat{f}(u)\big)$ into the set of patterns $\mathcal{P}$.

Algorithm iteratively repeats step (i) and (ii) above by executing the next internal state $p = ux$ and p is now considered as a new root. The algorithm stops when p is the final state. The pseudo code of this algorithm is described in FrequentLinkagePattern. Since the algorithm executes from the initial states at the depth 0 and stops at the final state at the depth n, therefore

Proposition 14.39. *The time complexity of the algorithm* FrequentLinkagePattern *is in* $\mathcal{O}(n)$.

Example 14.40. With VLFA model illustrated in Figure 14.7, the algorithm begins from the state $p = 0$ and compares the frequency of two direct successors $(A, q = 1)$ and $(T, r = 2)$. Since $N_1(A) = 360 > N_2(T) = 160$, the algorithm employs the path beginning by the edge $(0, 1)$. Because $\text{Card}(\text{Pred}[1]) = 1$, state 1 is not memory of the VLFA model, the algorithm continues by setting $p = q = 1$ and considering next direct successors of 1, $(G, q = 3)$ and $(T, r = 4)$. Since $N_3(G) = 132 > N_4(T) = 208$, the algorithm follows the edge $(1, 4)$. Since the state 4 has $\text{Card}(\text{Pred}[4]) = 2$ (state 4 has two direct predecessors), thus the model lost of memory at state 4. The path $0 \xrightarrow{A} 1 \xrightarrow{T} 4$ constitutes then the maximal pattern $P = AT$ of the size $f(AT) = 2 + 208$ in which $X_2 = T$ depends only on $X_1 = A$, $P(X_2 = T|X_1 = A, X_0, X_{-1}, \ldots) = P(X_2 = T|X_1 = A)$ and $P(X_2 = T|X_1 = A) \neq P(X_2 = T|X_1 = T)$. The algorithm continues by executing the state 4 as the initial state. Finally, the maximal linkage pat-

```
FrequentLinkagePattern($\mathfrak{A}(\mathcal{L})$)
 1: Input: a $\mathfrak{A}(\mathcal{L})$;
 2: Output: a set of frequent conservative patterns $\mathcal{P}$;
 3: $\mathcal{P} \leftarrow \emptyset$;                    /*pattern vector*/
 4: $p \leftarrow$ root;
 5: while ($p$ is not final state of VLFA $\mathfrak{A}$) do
 6:    if ($\exists q \in$ Succ[$p$] s.t. $N_u$ maximum) then
 7:       if (Card(Pred[$q$]) = 1) then
 8:          $p \leftarrow q$;
 9:       else
10:          return $u$;
11:          $\widehat{f}(u) \leftarrow |u| + N_u$;
12:          $\mathcal{P} \leftarrow \mathcal{P} \cup \big(u, \widehat{f}(u)\big)$;
13:          $p \leftarrow q$;
14:       end if
15:    end if
16: end while
```

terns AT, T and CT are identified on the maximal path of nodes $0, 1, 4, 7, 9$ and 10.

Finding Protective Linkage Pattern. Similarly with algorithm FrequentLinkagePattern, we propose an algorithm in linear time for the problem of protective linkage pattern (14.8). This time the criterion of execution is changed to decide the path of follows. The path with maximal positive string count and the degree of association $\pm\chi^2$ is used to determine the path to follow. The algorithm begins from the initial state, following the top to down of VLFA, and successively employs the probable positive path (path having maximum frequency of positive string count) with string patterns satisfying $\pm\chi^2 > \beta$, where $\beta > 0$ is the significance threshold. For each internal state p satisfying respectively two constraints $\max_{(x,p)\in \text{Succ}[p]} N_p(x)$ and $\pm\chi^2(x) > \beta$, the algorithm checks property decay of memory of the model at the successor state of p, $q = T(p, x)$:

(i) if the state q is not the memory of the model, Card(Pred[q]) = 1 (there is only one incoming edge into q), the state q represents then the same cluster of positive strings as p from the root, the algorithm iteratively continues by executing the next state,

(ii) if the state q decays the memory of model, $\mathrm{Card}(\mathrm{Pred}[q]) \geq 2$ (there is least two direct predecessors or incoming into the state q), VLFA model is lost the memory at q. Algorithm returns the pattern u associated to q with the size $\widehat{f}(u) = |u| + N_u$. The string u represents then the maximal protective pattern with high linkage disequilibrium verifying linkage constraint of problem (14.8), where the variables in this pattern are closely related. Then, input the pattern $M^+ = \big(u, \widehat{f}(u), \pm\chi^2(u)\big)$ into the set of patterns $\mathcal{P}^+$.

Algorithm iteratively repeats step (i) and (ii) above by executing the next internal state $p = ux$ and p is now considered as a new root. The algorithm stops when p is the final state. It stops when p reaches the final state. The pseudo code of this algorithm is described in PROTECTIVELINKAGEPATTERN.

```
PROTECTIVELINKAGEPATTERN($\mathfrak{A}(\mathcal{L}), \beta$)
1: **Input**: a $\mathfrak{A}(\mathcal{L})$;
2: **Output**: a set of protective linkage patterns $\mathcal{P}^+$;
3: $\mathcal{P}^+ \leftarrow \emptyset$;                    /*pattern vector*/
4: $p \leftarrow$ root;
5: while ($p$ is not final state of VLFA $\mathfrak{A}$) do
6:   if ($\exists q \in \mathrm{Succ}[p]$ s.t. $N_u^+$ maximum and $\pm\chi^2(u) > \beta$) then
7:     if ($\mathrm{Card}(\mathrm{Pred}[q]) = 1$) then
8:       $p \leftarrow q$;
9:     else
10:      return $u$;
11:      $\widehat{f}(u) \leftarrow |u| + N_u$;
12:      $\mathcal{P}^+ \leftarrow \mathcal{P}^+ \cup \big(u, \widehat{f}(u), \pm\chi^2(u)\big)$;
13:      $p \leftarrow q$;
14:    end if
15:  end if
16: end while
```

Example 14.41. Given VLFA model in Figure 14.7, the algorithm PROTECTIVELINKAGEPATTERN works as follows:

- Beginning from the state $p = 0$ and compares the frequency of two direct successors $(A, q = 1)$ and $(T, r = 2)$. Since $N_1^+(A) = 176 < N_1^-(A) = 184$ and $N_2^+(T) = 74 < N_2^-(T) = 86$ $(\pm\chi^2 < 0)$, the algorithm does not employ both edges $(0, 1)$ and $(0, 2)$.

- At state $p = 1$, the direct successor $(G, 3)$ of this state verifies $N_3^+(G) = 73 > N_3^-(G) = 59$ and $\pm\chi^2 = 1.74 > 0$ (P-value $= 0.18$), thus the algorithm follows the direction $1 \rightarrow 3$. Because Card(Pred[3]) = 1, state 3 is not memory of the VLFA model, the algorithm continues by setting $p = q = 3$ and considering next direct successors of 3, $(C, q = 5)$ and $(T, r = 6)$.
- On the edge $(3, C, 5)$, since the haplotype count gives $N_5^+(C) = 13 < N_5(C) = 28$ thus this edge will not be tested for association test. Since the edge $(3, T, 6)$ has $N_6^+(T) = 60 > N_6^-(T) = 31$, this edge will be tested for determining the protective pattern. The Chi-square's test gives $\pm\chi^2 = 10.5 > 0$ and P-value $= 0.0012$. Thus, the algorithm continues by following the edge $(3, 6)$ and the path $1 \xrightarrow{G} 3 \xrightarrow{T} 6$ constitutes the protective pattern GT with high statistical significance.
- Because the state 6 has Card(Pred[6]) = 2, thus the model lost of memory at state 6. We obtain then the maximal protective pattern $P = GT$ of the size $f(AT) = 2 + 60$ in which $X_3 = T$ depends on $X_2 = G$ and $X_1 = A$, $P(X_3 = T|X_2 = G, X_1 = A, X_0, X_{-1}, \dots) = P(X_3 = T|X_2 = G, X_1 = A)$ and $P(X_3 = T|X_2 = G, X_1 = A) \neq P(X_3 = T|X_2 = T, X_1 = T)$ and $P(X_3 = T|X_2 = G, X_1 = A) \neq P(X_3 = T|X_2 = T, X_1 = A)$. The algorithm continues by executing the state 6 as the initial state and gives the protective pattern AA.

Remark 14.42. The algorithms FrequentLinkagePattern and ProtectiveLinkagePattern can be performed another way as follows. First, we search the maximal path by the use of Theorem 14.29. Hence, we mark all memory states on this path. The pattern located between two consecutive memory states is the linkage pattern.

14.5 Experimental Results on SNP Data

14.5.1 *VLFA Model for Haplotype Data*

We evaluate the VLFA model through applications to a widely studied real dataset originally presented by Kerem *et al.* in the study of the fine-mapping of the cystic fibrosis gene [12]. The cystic fibrosis disease caused by CFTR (Cystic Fibrosis Transmembrane conductance Regulator) gene mutations is well understood and a single functional gene CFTR has been located on chromosome $7q31$. The cystic fibrosis data contain 94 case haplotypes and 92 control haplotypes from 23 biallelic RFLP markers (each

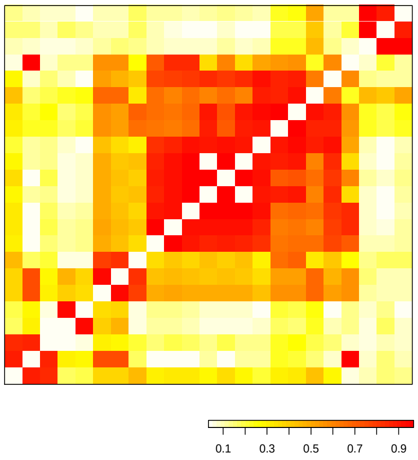


Fig. 14.8 Pair-wise LD of cystic fibrosis markers based on the D' measure.

marker has only two alleles, major and minor alleles) in a 1.8-Mb region including the CFTR gene. The disease susceptibility locus was located between marker 17 and 18 (≈ 0.88 cM away from the first marker), and a 3-bp deletion ΔF_{508} in the CFTR gene represents 66% (70 chromosomal haplotypes) of chromosomal mutations in the same gene. The sample incorporates genetic heterogeneity at the CFTR locus since only 62 of the case chromosomes carry ΔF_{508}. Consequently, the resulting haplotype data have become a useful test dataset for fine-mapping methods. Figure 14.8 shows the pairwise LD between 23 SNP markers computed by using D' value of the formula (14.1). We easily see that the 11 markers $9-20$ form a block with the high LD in which the block of 7 markers $9-15$ and the block of 5 markers $16-20$ are completely LD with $D' \geq 0.9$.

As we know that the loci markers are selected along from top to down of a chromosome and the recombination between the sites along a chromosome, thus all transitions of the stochastic automaton have the top-down transition form. In addition, a locus marker does not have the LD and the recombination by itself, thus all states of the automaton have no loop. Due to the LD between SNP markers mapping along a chromosome, the model has to be inhomogeneous memory because such model has the transition probabilities and memory length vary from one position to another on the chromosome. By these reasons, the stochastic acyclic weighted automaton can be well adapted. In fact, the structure of stochastic automata will adapt to the recombination and the LD between markers with increasing marker distance. In particular, the VLFA processes have the structure of variable length Markov chains in which the length of memory of the VLFA processes depends on the LD between nearby markers. The model has the longer memory if the markers are in the regions of high LD, whereas, the memory is short in the regions of low LD (see Example 14.28).

Figures 14.9 and 14.10 represent the graphs of VLFA model built from Cystic Fibrosis data with different compatible measures (the graphs are plotted by the graph visualization software Graphviz, *http://www.graphviz.org/*). For readability, only edges with the haplotype frequency above 0.01 are shown. The graphs **A** and **B** are the VLFAs learned by the μ-compatible ($\mu/2 = 0.75$ and $\mu/2 = 0.9$, respectively) and the graph **C** is the VLFA built by MSD-compatible criteria. While the VLFAs **D**, **E**, **F** and **G** are respectively constructed by α-compatible criterion with $\alpha = 0.5, \alpha = 0.1, \alpha = 0.01$ and $\alpha = 0.001$. With $\alpha = 0.001$, the VLFA is decomposed into 4 independent subgraphs that define 4 haplotypic blocks. The thickness path represents the haplotypic pattern strong associated with the disease ($\pm\chi^2 > 3.84$). Biologically, the VLFA model reflects then the historical recombination and string block LD structure. We observe that the α-compatible measure gives the VLFA that fits well with the data. In fact, with $\alpha = 0.001$, the VLFA (graph **G**) shows clearly 4 blocks of haplotypes with high LD that corresponds to the blocks identified by pairwise LD of 23 SNPs given in Figure 14.8. In this graph we see that the model lost completely the memory at some states such as state 3, 8 and 19. These states divide the data into 4 independent blocks of haplotypes I, II, III, IV, in which the block IV is very strong LD and the haplotypes are distinguished into two clusters. The first cluster is associated with the case haplotypes and the second one is associated with control haplotypes.

14.5.2 *Haplotype-Disease Association*

14.5.2.1 *Haplotype Cluster Association*

As the prediction, the association tests based on haplotypic clusters tend to be more powerful than classical single marker analyses. The disadvantage of haplotype association test is the relatively large number of observed haplotypes, which increases the degrees of freedom for the test statistic. That reduces the power of haplotype association analyses. For example, for m haplotypes, a goodness-of-fit [27] test that compares haplotype fre-

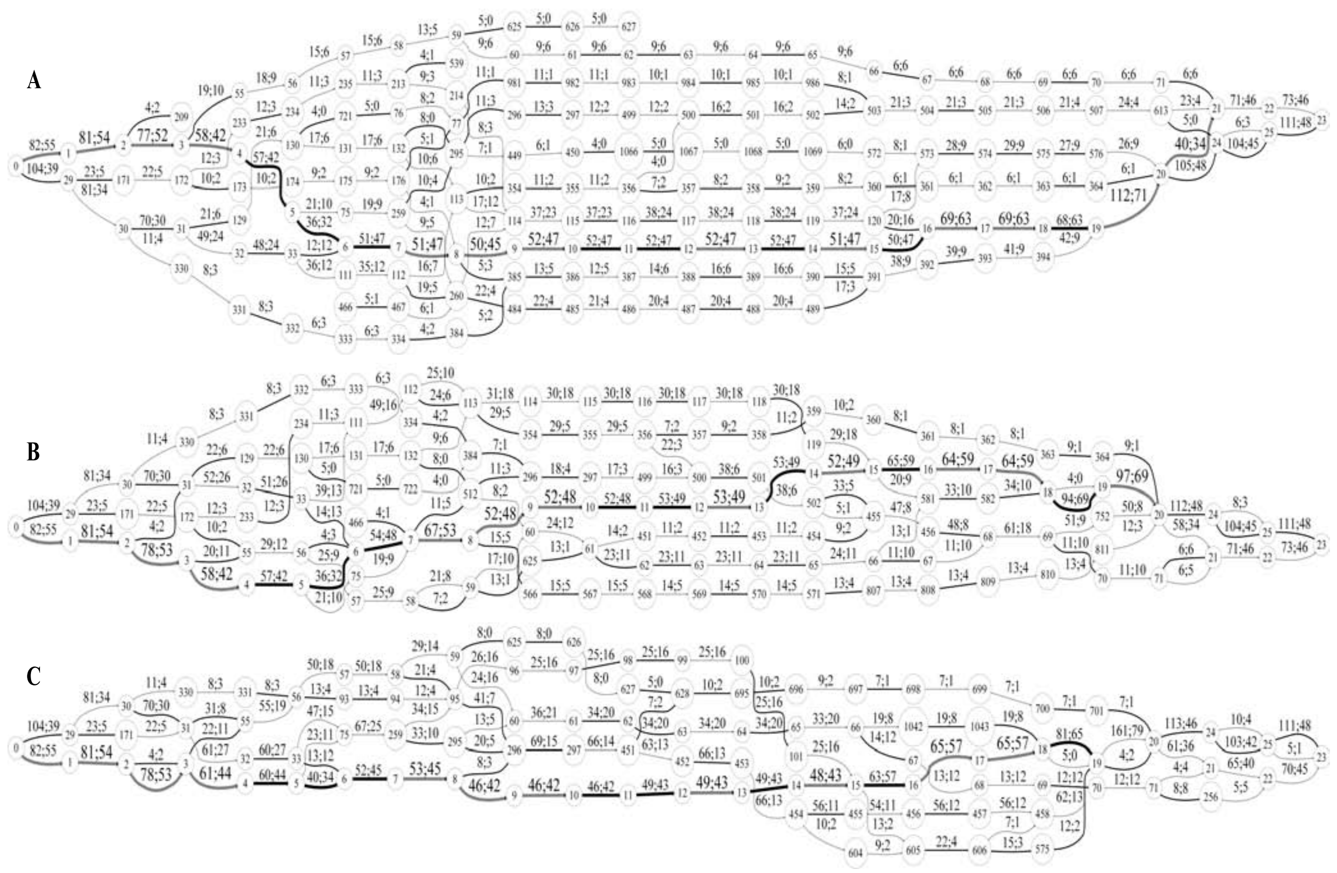


Fig. 14.9 VLFA of case and control haplotypes on the map of 23 loci markers with μ-compatible (**A** ($\mu/2 = 0.75$) and **B** ($\mu/2 = 0.9$)) and MSD-compatible (**C**).

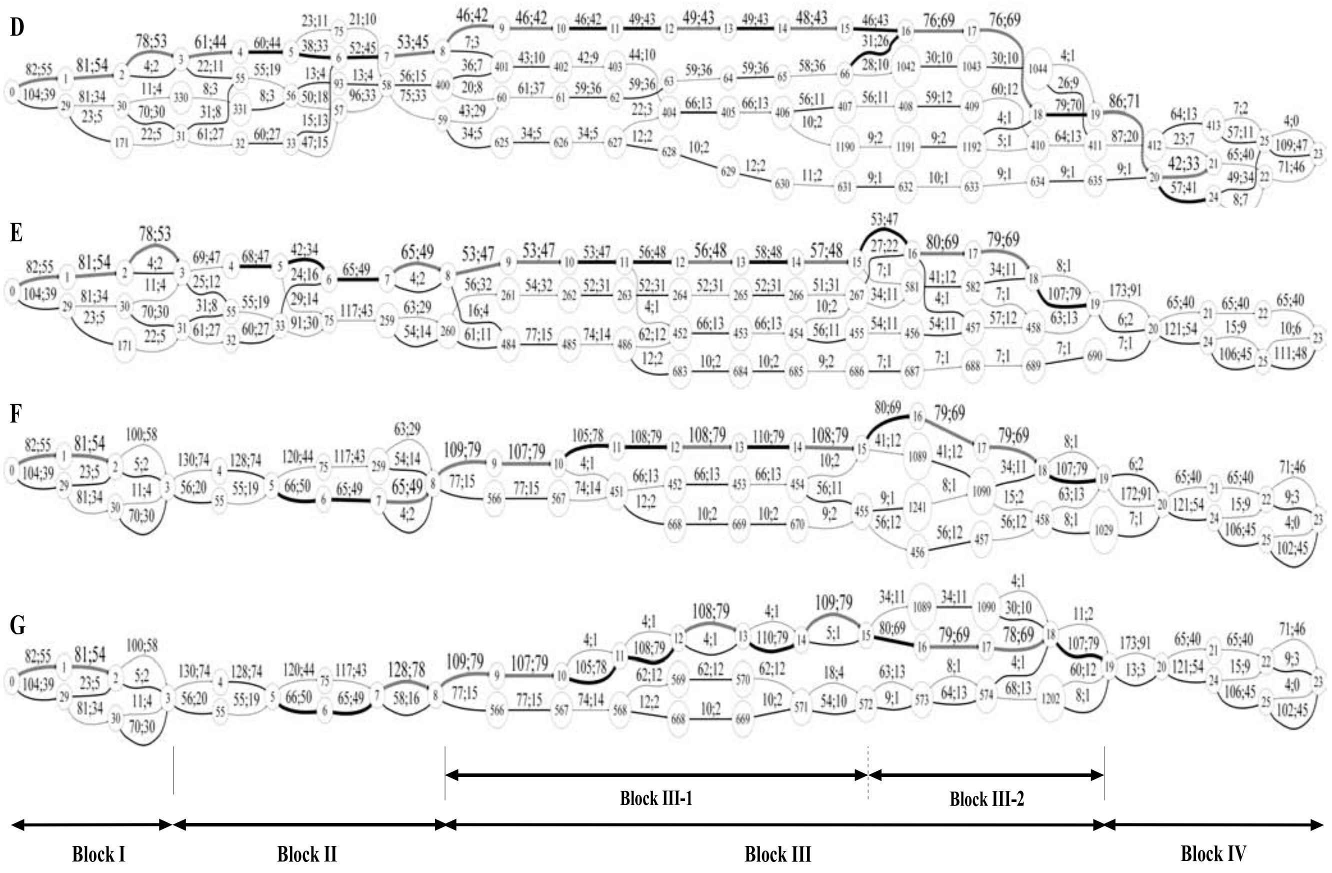


Fig. 14.10 VLFA of case and control haplotypes with α-compatible ($\alpha = 0.5, \alpha = 0.1, \alpha = 0.01$ and $\alpha = 0.001$, respectively).

quencies in cases with those in controls asymptotically follows a chi-square distribution with $m-1$ degrees of freedom under the null hypothesis of no association. We believe that haplotypes grouped into the same clusters by VLFA model–each cluster consisting of haplotypes with similar haplotype structure, and hopefully similar risks–reduces the degrees of freedom and then increases the power of haplotype association tests. In this approach, haplotype similarity in cases is compared with that in controls around each marker. Based on this, using the algorithm ClusterAssociation, we performed the association test for all splitting edges in order to recognize cluster strongly associated with the phenotype trait. In this experimentation, the test of statistic is based on Chi-square $\pm\chi$ test with the degrees of freedom of 1. So if we choose the significant probability P-value< 0.05, the haplotype clusters and haplotype patterns having the $\pm\chi^2$-score greater than $\beta = 3.84$ will be considered as the significant patterns. The result represented in this section is affected by this threshold value.

Figure 14.11 presents the distribution of the Chi-square $\pm\chi$ and corresponding $-\log_{10} P$ values of all "splitting edges" across 23 depths of VLFA using Pearson's chi-square test for 2x2 contingency table. We observed that the cystic fibrosis data have a very strong signal-large variation allelic

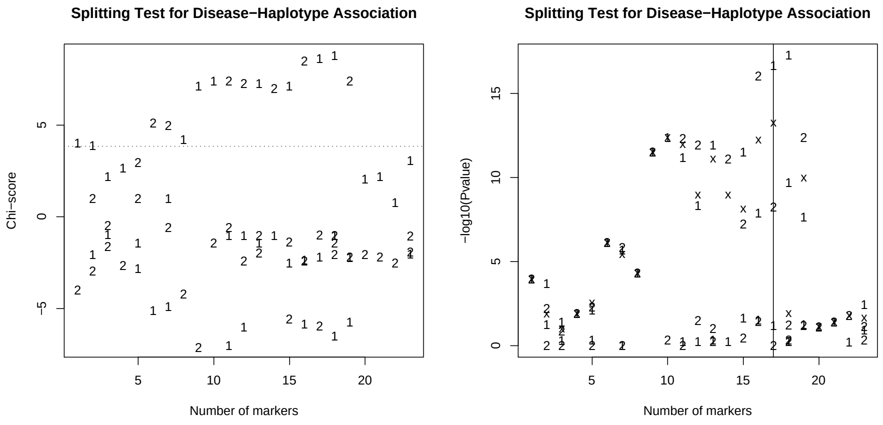


Fig. 14.11 The distribution of haplotypic cluster association test. The symbol (1) shows the association test of allele **1**, the symbol (2) represents the test of allele **2** at each splitting edge and the (x) one indicates the single marker test. The solid vertical line indicates the true location of ΔF_{508} at marker 17, the most common disease mutation identified in the CFTR gene. We clearly see that the splitting edge test is more powerful than single marker test.

frequency between the cases and controls, so test P-values tend to be extremely small. With a statistical significance level of 0.001 we obtain the subgraph beginning at state 8 and ending at the state 19 that represents the haplotypic cluster, which is strongly associated with the disease (the path marked in red in graph **G** of Figure 14.10). This cluster contains the marker 17, the true location of deletion ΔF_{508} in the CFTR gene, where the disease susceptibility locus was located. This result is coherent with those reported in [12].

For comparison with results from our approach, we accurately compare very small P-values between haplotype cluster test and single marker test (allelic tests with one test per marker). We observed that the result of association test on the haplotype cluster given by VLFA is more powerful the single marker association test. The smallest splitting edge test P-value was $5.9e^{-18}$, whereas the smallest single marker test P-value was $6.2 \times e^{-14}$.

14.5.2.2 *Haplotype Pattern Association*

According to algorithm ProtectivePattern, using the most association path (path having maximum $\pm\chi^2$ value) of cystic fibrosis VLFA (Figure 14.10), the blocks of risk haplotype patterns were recognized permitting to locate disease susceptibility loci. The graphs of Figure 14.12 respectively show the case frequency of maximal haplotype and the $\pm\chi^2$-score corresponding to the VLFA model with different compatible measure. The result shows that the variation allelic frequency between the case and the control is very large, so the $\pm\chi^2$-scores of the association test are large and then the LD is also large. The case frequency of maximal haplotype identified on the model associated to α-compatible is greater than those given by MSD-compatible and μ-compatible. We observe that, with the significant level $\pm\chi^2 > 3.84$, the haplotype **111122211122121211121111** is strong associated of the disease (the red paths on the graphs). The maximal haplotype patterns with the high degree of LD will be found on this maximal protective haplotype according to the memory of the VLFA model.

In fact, the VLFA model with $\alpha = 0.001$ (graph **G** of Figure 14.10) provides 4 great blocks of SNP markers in high LD at the memory states 3, 8 and 19. For each block, we only determine the haplotypic fragments that are significantly overrepresented in case haplotype samples. The table 14.4 presents the most protective pattern identified from these blocks that have significantly more occurrences in the family of case haplotype. In particular, the 4 markers $\{16, 17, 18, 19\}$ exhibits the common haplotypic

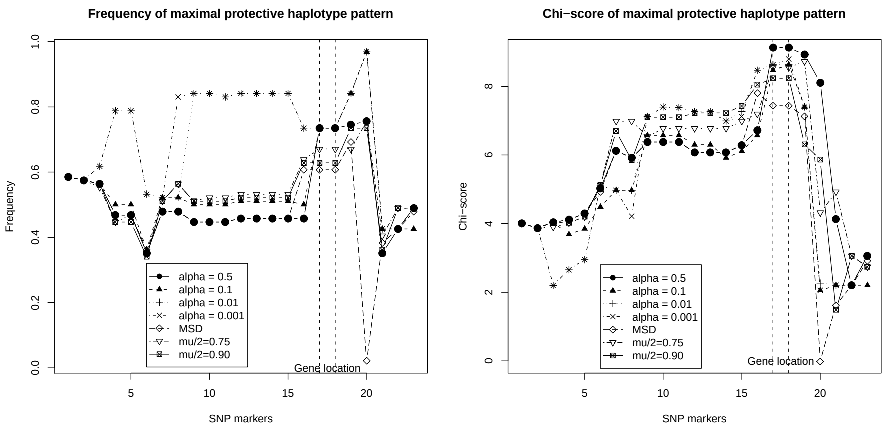


Fig. 14.12 Location of disease susceptibility loci. The figure shows the frequency and $\pm\chi^2$-score of maximal protective haplotype associated with the disease. Gene location was localized at the region between marker 17 and 18 (the region between vertical dashed lines). The frequency and $\pm\chi^2$-score carried out by α-compatible in the region of gene location are greater than those given by the μ-compatible and MSD-compatible.

pattern **2212**. This pattern is shared by 69 (73%) case haplotypes and passes through the most significant associated edge of the VLFA that has the very large $\pm\chi^2$-score (≥ 8.26), which is very close to the total number of disease chromosomes that have the disease susceptibility mutation. That could be viewed as the *risk haplotype pattern* of the cystic fibrosis. This haplotype pattern is driving the results in both the RFLP and the coding region. And then, the deletion ΔF_{508} is located in the region nearby RFLP marker 17 and 18 with much high significant level, which is compatible with the results given by [12]. These results demonstrate that our method is competitive with standard single marker and haplotypic test on these data. This method does not require a haplotype window size and it involves only a small number of test.

Table 14.4 Block of markers with high LD and strong associated with the disease.

Block	Markers	Hap. pattern	Hap. Freq.	Case freq.	Control freq.	$\pm\chi^2$
I	1-2	**11**	0.44	0.67	0.33	3.87
II	6-8	**221**	0.35	0.76	0.24	5.10
III	9-15	**1122121**	0.58	0.73	0.27	7.25
IV	16-19	**2212**	0.42	0.87	0.13	8.79

14.5.3 *Comparative Study*

The approaches concerning the multilocus analysis related to our method are introduced in [8, 10, 23]. These methods are based on data mining technique including the clustering technique and discovery of frequent haplotype pattern that are gaining more interest as potential tools in identification of complex disease loci. In [23], the authors proposed a non-parametric method for haplotype mapping called haplotype pattern mining (HPM). In this approach, the haplotype patterns in case and control are examined and the pattern frequencies are used for predicting the disease gene locations. This method shown a powerful localization, even when the number of phenocopies and markers are large. Several methods used sliding-window technique [8, 10], a commonly approach used for analyzing a large number of markers. Given a fixed window size (WS), one slides this window along the region of interest and computes the test statistic for each window. In CLADHC program [8], the sliding-window approach via cladistic analysis developed in a logistic-regression framework, for which, the cladistic and haplotype similarity methods test for clustering of similar haplotypes within cases and controls. Using direct data mining, the Hapminer algorithm, a most recent algorithm developed in [10], is robust and effective even for data containing a large number of markers and high rate of phenocopies (small relative risks). This algorithm utilized the density-clustering algorithm. It allows capturing the sharing of haplotype due to historical recombination events. The effectiveness of Hapminer depends on the similarity of haplotype fragments.

However, the HPM approach has also a limitation, it does not consider LD between consecutive markers by allowing 'don't care' symbol in the patterns; and then many haplotypes have been counted multiple times. In genetic epidemiology, HPM may not be appreciated studying complex disease because the reason is that disease susceptibility gene embedded haplotypes tend to be close to each other due to LD, while other haplotypes may be viewed as the random noise. The CLADHC program and Hapminer algorithm are powerful and returned more accurate results comparing with the classical single test. These algorithms, however, are not structurally rich and robust. Biologically, these approaches do not adapt to the degree of LD, which can vary throughout a region. If the WS are too small, the information of LD is lost, whereas, if the WS are too large, excessive noise is introduced and it decreases then the degree of LD. The authors of the paper [10] demonstrated that the proposed algorithm Hapminer is more robust and efficient than CLADHC and HPM. Therefore, in this work we only compare the our own method with two robust related works focused on data mining technique including the clustering algorithm and discovery of frequent haplotype pattern: the Hapminer algorithm[10] and the recursive partitioning tree (RPT) [29]. These proposed methods are gaining more interest as potential tools for identifying disease susceptibility gene.

Hapminer program depends on the similarity measure of pairwise haplotype fragments defined with respect to a particular marker locus. Given a fixed WS, for each marker, one slides this window along the region of interest and computes the pair-wise distances of haplotype segments for each window. Hence, one finds the similar clusters and reports the ones with the highest statistic of the test ($\pm\chi$-score). This method allows capturing the sharing of haplotype due to historical recombination events. The effectiveness of Hapminer program depends on the similarity measure of pair-wise haplotype fragments defined with respect to a particular marker locus. Given a fixed WS indexed by $-\ell, \dots, -1, 0, 1 \dots, r$ ($WS = \ell + r + 1$) and the physical distance x_t from any locus to locus 0, $-\ell \leq t \leq r$, the similarity between pair of haplotype fragments s_i, s_j with respect to the locus 0 is defined by

$$S(s_i, s_j) = \sum_{t=\ell}^{r} w_1(x_t)\mathbf{1}_{\{s_i(t), s_j(t)\}} + \sum_{t=1}^{r'} w_2(x_t) + \sum_{t=-1}^{\ell'} w_2(x_t), \quad (14.31)$$

where $\mathbf{1}_{\{s_i(t), s_j(t)\}}$ is the identity function (returns 1 if $s_i(t) = s_j(t)$ and 0 otherwise), $s(t)$ is the allele at locus t, and r' and $-\ell'$ are two boundary loci such that the two haplotypes s_i, s_j are identical between these two loci and different at both locus $r' + 1$ and locus $-\ell - 1$. The weights w_1 and w_2 are two decreasing functions so that the measure on each locus is weighted according to the distance from locus 0. This measure generalizes several haplotype similarity measures in the literature. The first term in (14.31) is a weighted measure of the number of alleles in common between haplotypes s_i and s_j in the region, which can be thought of as Hamming similarity. The remaining terms form a weighted measure of the longest continuous interval of matching alleles around locus 0, which has some resemblance to the notion of a longest common substring

The second approach focused on the recursive partitioning tree (RPT) based on the classification and regression of decision trees for predicting the disease susceptibility marker [29]. The algorithm based on RPT is well-known for its robustness and learning efficiency with its learning time complexity of $\mathcal{O}(mn \log n)$ [3, 21]. The algorithm takes the haplotype data and builds a classification binary RPT, in the presence of interactions employing a given set of independent marker alleles (co-variates or attributes), to partition the haplotype data into relatively more similar subsets according to a splitting rule. This splitting rule is defined based on an attribute (marker) and its binary partition that divides the corresponding co-variate space into two non-overlapping sub-regions. To choose a best attribute for growing the RPT, the method used the Gini index function whose measures the degree of interaction between different classes and choose attribute that provides the minimal degree of interaction between classes of haplotypes in the splitting process. In the recursive partitioning procedure, the haplotype data is first split into two children nodes by choosing the best SNP marker for the splitting. The splitting process can be repeated recursively on the sub-trees until terminated by a stopping criterion. Finally, the algorithm conducts the association test based on groups of haplotypes on the RPT model instead of individual haplotypes.

The Hapminer algorithm, however, has a limitation. The degree of LD can vary throughout a region. If the WS are too small, the information of LD is lost, whereas, if the WS are too large, excessive noise is introduced and it decreases then the degree of LD. Indeed, For the comparison, we run the Hapminer program with the different haplotype segment length parameters WS=3, WS=5 and WS=7. Because the major advantage of our approach is adaptive to LD and haplotype similarity, for the coherence in comparison, the weighted measure parameters in the Hapminer program is set to be LD and Hamming similarity measures. More precisely, for each WS, the Hapminer program runs successively with the following parameter: $w_1 = LD, w_2 = 0$; $w_1 = 0, w_2 = LD$; $w_1 = w_2 = LD$ (LD measure) and finally $w_1 = 1, w_2 = 0$ (Hamming similarity measure). We observed that the results have a large variance throughout haplotypic segment length and weighted parameters. For any WS parameter, the results corresponding to $w_1 = LD, w_2 = 0$ and $w_1 = 0, w_2 = LD$ are bad at any meaning. While the results with $w_1 = LD, w_2 = LD$ and $w_1 = 1, w_2 = 0$ is much better. If we set $w_1 = 1, w_2 = 0$ the results corresponding to the haplotype segment parameters WS=3 and WS=5 are better than the one WS= 7, while the results associated to WS= 5 are best in both cases $w_1 = w_2 = LD$ and $w_1 = 1, w_2 = 0$. In addition, the different WS and weighted parameter give the different identified markers (see Table 14.5). It is difficult to capture the disease susceptibility genes. That can be explained as follows. The haplotype fragments are more similar in small WS than in large WS due to the noise. This experiment shows that the effectiveness of the Hapminer algo-

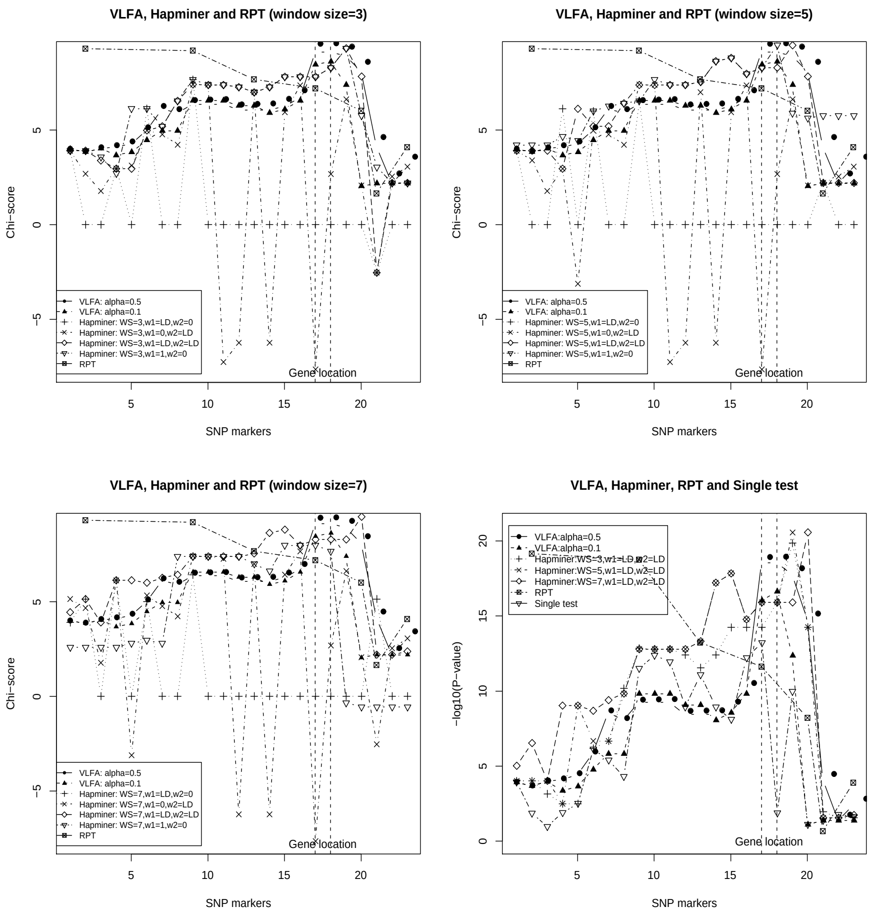


Fig. 14.13 Comparison VLFA model versus Hapminer. Our predicted disease location is the region between two markers 17 and 18 with higher significant level than those by Hapminer and single marker test.

rithm depends on the similarity measure and the power of association test varies throughout the haplotype segment parameter WS and the weighted parameters w_1 and w_2. Figure 14.13 and Table 14.5 illustrate the comparison of VLFA model with Hapminer algorithm, RPT and simple χ-test of single marker based on the power of association test measured $\pm\chi^2$-score and the statistical significance P-value.

Table 14.5 Comparative study on cystic fibrosis data.

Method	$\pm\chi$-score	$-log_{10}(P\text{-value})$	Identified SNPs
VLFA ($\alpha = 0.5$)	9.13	18.6	$16, 17, 18, 19$
VLFA ($\alpha = 0.1$)	8.46	16.65	$16, 17, 18, 19$
RPT	9.27	19.12	2
Hapminer WS$= 3, w_1 = w_2 = LD$	9.29	19.83	19
Hapminer WS$= 3, w_1 = 1, w_2 = 0$	9.29	19.83	19
Hapminer WS$= 5, w_1 = w_2 = LD$	9.46	20.54	19
Hapminer WS$= 5, w_1 = 1, w_2 = 0$	9.46	20.54	18
Hapminer WS$= 7, w_1 = w_2 = LD$	9.46	20.54	20
Hapminer WS$= 7, w_1 = 1, w_2 = 0$	7.96	14.76	$15, 16, 17$
Test of single marker		13.21	17

From the RPT, we perform the association test based on grouped haplotypes and identify an optimal sub-tree on which the association test provides the strongest disease-associated allelic markers. For each sub-tree, we compare the haplotype frequencies between case and control by the use of $\pm\chi$-score statistic. An optimal sub-tree will be selected if its raw P-value is small enough (0.01). The RPT method predicts likely locating the group of markers $\{2, 3, 7, 10, 17, 23\}$ that is most strong disease-associated markers with the P-value varying from $1.3e^{-4}$ ($-\log_{10}(P\text{-value})=3.88$) to $7.5e^{-20}$ ($-\log_{10}(P\text{-value})=19.12$) and the $\pm\chi^2$-score varying from 4.06 to 9.27. The marker 2 is predicted with strongest $\pm\chi^2$-score (9.27) and smallest P-value ($7.5e^{-20}$). The known disease susceptibility marker 17 has the P-value of $6.1e^{-14}$ ($-\log_{10}(P\text{-value})=13.21$) and the $\pm\chi^2$-score of 7.17. However, some important markers $\{16, 18\}$, where the deletion ΔF_{508} was located [12], are not identified by RPT, but they are identified by VLFA. The explanation is that if the number of attributed marker is large, the recursive partitioning algorithm makes the loss of important information in the splitting process. In fact, when we build the RPT for each small region in high LD, for example block $1-8$, $8-15$, $16-20$, the RPT of each block gives the same results (results not show) with the VLFA. Thus, the advantage of RPT is that it can be powerfully applied for pair-wise markers analysis with a small number of markers.

We observed that the result from VLFA is better than Hapminer, RPT and simple χ-test. In particular, in the region that had the disease susceptibility mutation, the alleles are strong LD, the $\pm\chi^2$-score (resp. P-value) provided by VLFA model is greater (resp. smaller than) than the one given by Hapminer for any WS parameter and any weighted parameter w_1 and

w_2, by RPT and by the single marker test. In both cases $\alpha = 0.5$ and $\alpha = 0.1$, our predicted disease location is the region of marker 16, 17, 18 and 19 ($\pm\chi^2$-score= 9.13 and significant P-value=$-\log_{10}(P\text{-value})$= 18.6) indicated by the dashed vertical lines that are very close to the true disease susceptibility mutation ΔF_{508}. While, the predicted disease location by Hapminer is varying depending on WS and weighted parameter and far away comparing with the true location. In the case WS= 3, the disease location is at marker 19 ($\pm\chi^2 = 9.29$, $-\log_{10}(P\text{-value})$= 19.83) for both weighted parameter $w_1 = w_2 = LD$ and $w_1 = 1, w_2 = 0$; if WS= 5, the disease location is at marker 18 and 19 ($\pm\chi^2 = 9.46$, $-\log_{10}(P\text{-value})$= 20.54) for $w_1 = w_2 = LD$ and $w_1 = 1, w_2 = 0$; and if WS= 7, the disease location is at marker 20 ($\pm\chi^2 = 9.46$, $-\log_{10}(P\text{-value})$= 20.54) for and $w_1 = w_2 = LD$ and at marker $15, 16$ and 17 ($\pm\chi^2 = 7.96$, $-\log_{10}(P\text{-value})$= 14.76) for $w_1 = 1, w_2 = 0$. The RPT predicts the disease susceptibility mutation at marker 2 with large $\pm\chi^2$-score (9.27) and small P-value ($-\log_{10}(P\text{-value})$= 19.12). This marker is so far away with the true disease susceptibility mutation. The true marker 17 is predicted with the smaller significant statistic ($\pm\chi^2$=7.17, $-\log_{10}(P\text{-value})$=13.21).

14.6 Conclusion

We have introduced combinatorial optimization method based on discrete structure of VLFA for motif finding with mathematical programming approach and other applications. This method is based on symbolic method and non-numerical algorithm that are powerfully applied for large database. The structure of VLFA provided the tools to store, visualize, classify the data and identify specific disease-associated genes from the biological data with a simple method and efficient algorithm. While the mathematical programming has not been applied to the pattern discovery problem, our models demonstrated that it provides a powerful alternative to successfully tool for diverse applications. The experimental results on biological data shown that multigenetic marker analysis has higher potential for powerful detection of trait-marker associations than do single marker analysis. The VLFA models could be adapted to diverse pattern finding in different types of biological and biomedical data. In particular, these models can be efficiently applied for data mining (clustering and the bi-clustering) and knowledge discovery from such data.

The methods presented in this chapter open several promising directions for the future work. First, the learning linkage pattern via VLFA will be applied for mining co-regulated genes from time series expression data (gene expression is temporal process). This is a most important step for identifying gene regulatory networks. Regulatory network will be constructed from patterns with assumption that the genes in each pattern have the same regulation. In fact, the biological processes start and finish in a contiguous period of time points. That exhibits a strong auto-correlation between successive time points, and then it leads to increase (or decrease) the activity of set of genes that may be appeared in linkage pattern with variable length in time. Second, the combinatorial optimization method based on VLFA models will offer a new tool for the diagnosis. Starting from a large collection of individual patients, we will insulate the characteristic molecular portraits of a given pathology. The characteristic haplotype could be seen as a detectable signature. These problems will successively be studied in the next future work.

References

1. Balding, D. (2006). A tutorial on statistical methods for population association studies, *Nature Genetic, Reviews* **7**, pp. 781–791.
2. Brazma, A. (1998). Approaches to the automatic discovery of patterns in biosequences, *Journal of Computational Biology* **5**, 2, pp. 277–304.
3. Breiman, L. (1984). Classification and regression trees, *Wadsworth* .
4. Buhlmann, P. (1999). Variable length markov chains, *The Ann. of Stat.* **27**, 2, pp. 480–513.
5. Carrasco, R. and Oncina, J. (1994). Learning stochastic regular grammars by means of a state merging method, in *Grammatical Inference and Applications, LNAI-862* (Springer-Verlag), pp. 139–152.
6. Daly, M. (2001). High-resolution haplotype structure in the human genome, *Nature Genetics* **29**, pp. 229–232.
7. Das, M. and Dai, H. (2007). A survey of dna motif finding algorithms, *BMC Bioinformatics* **8**, 7.
8. Durrant, C. J. (2004). Linkage disequilibrium mapping via cladistic analysis of single nucleotide polymorphism haplotype, *Am. J. Hum. Genet* **75**, 1, pp. 35–43.
9. F. Thollard, P. D. and la Higuera, C. D. (2000). Probabilistic DFA inference using Kullback-Leibler divergence and minimality, in *proceedings of ICML'00*, pp. 975–982.
10. Jing, L. and Tao, J. (2005). Haplotype-based linkage disequilibrium mapping via direct data mining, *Bioinformatics* **21**, 24, pp. 4384–4393.
11. Jonassen, I. (1995). Finding flexible patterns in unaligned protein sequences, *Protein Science* **4**, 8, pp. 1587–1595.

12. Kerem, B. (1989). Identification of the cystic fibrosis gene: genetic analysis, *Science* **245**, 4922, pp. 1073–1080.
13. Kohler, K. and Bickeboller (2005). Case-control association tests correcting for population stratification, *Annals of Human Genetics* **69**, pp. 98–115.
14. Kristin, G. (2002). Partterns of linkage disequilibrium in the human genome, *Nature Genetics* **3**, pp. 299–310.
15. Lon, R. and John, I. (2001). Association study designs for complex diseases, *Nature Genetics* **2**, pp. 91–99.
16. Lothaire, M. (2005). *Applied Combinatorics on Words* (Cambridge University Press).
17. Peeters, R. (2000). The maximum edge biclique problem is np-complete, *Discrete Applied Math* **131**, 3, pp. 651–654.
18. Ron, D. (1998). On the learnability and usage of acyclic probabilistic finite automata, *Comp Syst Sci* **56**, pp. 133–152.
19. Sakarovitch, J. (2003). *Éléments de théorie des automates* (Vuibert Informatique, France).
20. Salomaa (1969). *Theory of automata* (Pergamon Press).
21. Therneau, T. (2000). An introduction to recursive partitioning using the rpart routines, Mayo clinic section of biostatistics, Mayo Foundation.
22. Thomas, A. (2004). Graphical modeling of the joint distribution of alleles at associated loci, *Am. J. Hum. Genet.* **74**, pp. 1088–1101.
23. Toivonen, H. (2000). Data mining applied to linkage disequilibrium, *Am. J. Hum. Genet* **67**, pp. 133–145.
24. Tran, T. (2007). Management and analysis of dna microarray data by using weighted trees, *Journal of Global Optimization* **39**, 4, pp. 623–645.
25. Tran, T. (2008). Biclustering des données de biopuces par les arbres pondérés de plus long préfixe, *Tech. et Sc. Info. (TSI)* **27**, 1-2, pp. 83–108.
26. Tran, T. (2010). Risk haplotype pattern discovery for gene mapping by recursive partitioning method based on weighted classification trees, *International Journal of Computational Intelligence in Bioinformatics and Systems Biology (IJCIBSB)* **1**, 3, pp. 213–245.
27. Tzeng, J. (2003). On the identification of disease mutations by the analysis of haplotype similarity and goodness of fit, *Am. J. Hum. Genet.* **72**, pp. 891–902.
28. Verzilli, C. J. (2006). Bayesian graphical models for genomewide association studies, *Am. J. Hum. Genet* **79**, 1, pp. 100–112.
29. Yu, K. (2005). Using tree-based recursive partitioning methods to group haplotypes for increased ower in association studies, *Ann. Hum. Genet.* **69**, pp. 577–589.
30. Zaslavsky, E. and Singh, M. (2006). A combinatorial optimization approach for diverse motif finding applications, *Algorithms for Molecular Biology* **13**, 1.

This page is intentionally left blank

Author Index

Alhazov, Artiom 459

Bartha, Miklós 585
Berlea, Alexandru 205
Blakeley, Brandon 59
Blanchet-Sadri, Francine 59
Bryant, Barrett R. 421

Caron, Pascal 103

Daciuk, Jan 133

Flouret, Marianne 103

Gunter, Josh 59

Holzer, Markus 1
Hrnčič, Dejan 421

Janicki, Ryszard 313
Javed, Faizan 421

Kleijn, Jetty 313
Koutny, Maciej 313
Krész, Miklós 585
Krassovitskiy, Alexander 459
Kutrib, Martin 1

Manea, Florin 525
Martín-Vide, Carlos 525
Mereghetti, Carlo 563
Mernik, Marjan 421
Mitchener, W. Garrett 633
Mitrana, Victor 525

Palano, Beatrice 563
Piskorski, Jakub 133

Ristov, Strahil 133
Rogozhin, Yurii 459

Simmons, Sean 59

Tîrnăucă, Cristina 387
Trang, Tran 663

Verlan, Sergey 459

Weissenstein, Eric 59

Subject Index

ε, *see* empty forest
λ, *see* empty string
ε-equivalent, 115, 116
λ-free normalform, 37

$A_{\vec{G}}$
  accepting regular forest languages, 228
  for k-ary queries, 249
  for simple binary queries, 240
  for XML streams, 278
acceptance probability, 569
accepted language, 569
accepting networks of evolutionary processors, 527
accepting networks of evolutionary processors with filtered connections, 532
accepting networks of picture processors, 549
accepting networks of splicing processors, 545
accepting networks of splicing processors with filtered connections, 547
activator arc, 373
activator occurrence net, 373
acyclic automaton, 167
adjacency list, 187
adjoint matrix, 566
all-matches policy, 231
alphabet, 62, 217, 322
alternating finite automata, 10
alternating loop, 590
alternating unit
  in graph, 590
ambiguity bounded pushdown automata, 47
ambiguous, 231
amo-net, 380
ANOVA, 635
ao-net, 373
apparent time, 636
association analysis, 669
association constraint, 675
attribute grammars, 253, 291, 302
  AG-like rules, 240
  and tree transformations, 291
  boolean valued, 254
  relation-valued, 254
attribute name, 214
attribute value, 215
attributes, 214
automata
  acyclic, 140
  alternating FA (AFA), 10
  ambiguity bounded PDA, 47
  Boolean FA (BA), 11
  branching bounded PDA, 47
  cellular automata (CA), 50
  deterministic, 137, **138**
  finite (FA), 6
  finite-turn PDA (FTPDA), 28
  Glushkov, 21

implementation, 194
incomplete, 140
iterative arrays (IA), 50
minimal, 137
minimization, 167
moderate PDA, 25, 27
multi-head FA, 48
multiple entry FA
deterministic (NDFA), 8
nondeterministic (NNFA), 12
position, 21
pushdown (PDA), 23, 25
recursive, 137, 194
restarting, 19, 49
sweeping FA, 19
Turing machine (TM), 42
two-way FA, 16
two-way multi-head FA, 49
unambiguous FA, 19
verified PDA, 47
visit FA, 19
automaton
commutative, 591
cyclic, 182
deterministic finite, 590
full, 591
incremental construction from sorted data, 168
incremental construction from unsorted data, 175
input alphabet, 590
minimal, **143**, 144
non-deterministic finite, 590
nondeterministic, **146**
permutation automaton, 591
recursive, **145**
semi-full, 591
soliton automaton, 596
states, 590
subautomaton, 591
transition function, 590
transition monoid, 591
trivial, 591
axes, 269

$B_G^{\leftarrow}$
for k-ary queries, 249
for general binary queries, 247
for simple binary queries, 240
backward equivalent, 114
Bayes's formula, 644, 657–658
Bayesian inference, 644
BBQ, 259
Berry-Sethi, 208
Berry-Sethi construction, 209, 210, 228, 229
bidirectionally equivalent, 115
binary queries, 207, 232
simple binary queries, 239
binomial distribution, 645
Boolean automata, 11
Boolean closure of CFL, 48
bottom-up, 251, 253
bottom-up forest automata, 225
bottom-up tree automata, 225
bounded language, 28
bounded measure, 37
BPP, 564
BQP, 564
branching bounded pushdown automata, 47
bridge
cutting, 621
in connected graph, 613
odd, 619
open, 619
semi-odd, 619
browser, 215

caching, 249
canonical equivalence
canonical class, 597
principal, 598
canonical partition, 597
on soliton graphs, 597
canonical form, 71
canonical form of words, 154
canonical stratified poset, 327
canonical total poset, 326
caterpillar expressions, 255
caterpillars, 255

CD grammar systems, 50
cellular automata, 50
characteristic vector, 10
chestnut, 600
  baby chestnut, 605
    regular, 605
children, 212
Chomsky normalform, 29, 37
Chrobak normalform, 8, 22
Church-Rosser languages, 49
cloning, 176, 182
closure expression, 105
coherent constraint, 675
coherent contiguous pattern, 672
combinatorial optimization, 674
communication complexity, 22
compatible partial words, 62
complement of a word, 63
complete answer aggregate, 259
complete finite automata, 7, 12
complexity classes
  logspace, 17
  polynomial time, 46
complexity measure, 4
  a-width, 20, 23
  awidth, 22
  height, 20, 23
  length, 4, 20
  o-length, 20
  prodc, 34
  prod, 36
  rpn, 20
  size, 20, 30
  state, 4
  sum, 20
  symb, 34
  var, 34–36
  width, 20
  bounded, 37
  connectedness, 35
  cycle rank, 23
  dense, 37
  derivation step, 35
  derivation tree, 35
  grammatical level, 35
  relation, 21
  s-measure, 4
  sn-measure, 40
  star height, 20, 22, 23
comtrace, 338
comtrace alphabet, 338
comtrace congruence, 338
condition, 365
conditional probability, 657
conditionally independent, 678
confidence interval, 644
confluence, 118
confluence state, 175, 182
congruence, 323
conjugate, 566
conjunctive queries over trees, 258
connectedness, 35
cons, 224, 232
constant rate effect, 637
constraint systems, 229
ConsvInf, 393
ConsvTxt, 393, 400
content model, 216, 218
content models, 221, 223
content-model, 216
context free grammars, 254
context information, 226
context-free grammar, 4, 29–31, 35–38
  λ-free, 37
  LL(k), 48
  LR(k), 33
  bounded, 32, 33
  Chomsky normalform (CNF), 31, 32, 37, 38
  Greibach normalform (GNF), 37, 38
  position restricted normalform, 37
  restricted, 37, 38
  right-linear, 30, 34, 36
  right-linear normalform, 34
  self-embedding, 36
  unary, 32, 35
contiguous pattern, 672
continuous distribution, 657
conversion

(deterministic) PDA to (deterministic) PDA, 24
(real-time) PDA to (real-time) PDA, 24
2NFA/2DFA to DFA, 17
2NFA/2DFA to NFA, 16
AFA to NFA, 13
AFA to NNFA, 12
Boolean automata to AFA, 11
Boolean automata/AFA to DFA, 11
bounded (normalform) FTPDA to (normalform) FTPDA, 29
bounded CFG to FTPDA, 33
bounded CNF grammar to FTPDA, 32
bounded PDA to FTPDA, 29
CFG to restricted CFG, 37
deterministic PDA to DFA, 26
DFA/2NFA to DPDA, 28
DFA/NFA to CNF grammar, 30
FTPDA to FTPDA, 29
limited NFA to DFA, 9
NDFA to DFA, 9
NFA to DFA, 7
NFA to NDFA, 9
NFA/DFA to regular expression, 22
normalform PDA to (unary) NFA/DFA, 27
normalform PDA to CNF grammar, 31
real-time DPDA to CFG, 31
regular expression to NFA/DFA, 21
restricted CFG to CNF/GNF grammar, 38
reversed AFA to DFA, 12
unary (normalform) DPDA to DFA, 28
unary 2DFA to NFA/DFA, 18
unary 2NFA to 2DFA, 18
unary 2NFA to NFA/DFA, 18
unary 2NFA/2DFA to AFA, 19
unary AFA to 2NFA/2DFA, 19
unary AFA to DFA, 15
unary AFA to NFA, 15
unary CNF grammar to DFA, 32
unary CNF grammar to NFA, 32
unary DFA to AFA, 15
unary DFA to CNF grammar, 30
unary DPDA to DFA/2NFA, 28
unary NFA to 2DFA, 17
unary NFA to AFA, 16
unary NFA to DFA, 8
unary normalform DPDA to CNF grammar, 31
Core XPath, 252, 258, 274
corpora, 274
correction query, 394
  edit distance, 402
  length bounded, 401
  prefix, 388
crossing
  alternating, 590
crossing sequence, 16
cut-off edit distance, 161
cycle
  alternating, 590
    accessible, 590
  in graph, 588
  usable, 602
cycle rank, 23
cyclic automaton, 182

data values, 256
datatypes, 222, 224
deep matching, 260
degree of association, 673
delta measure, 657
dense measure, 37
depth-first search, *see* DFS
derivation, 219, 249
  forest derivation, 232
  tree derivation, 232
derivations, 300
descriptional complexity, 1
descriptional system, 3
descriptors, 3
deterministic
  finite string automata, 223
  LPA, *see* DLPA
deterministic finite automaton, 6, *see* DFA

determinization, 2, 7, 12, 19, 32
DFA, 209
DFS, 213, 225, 227, 280
diachronic, 636
dictionary, 133, **134**
  compression, 197
  dynamic, 135
  implementations, 136
  static, 135
difference cover, 574
differential equation, 635
direct sum and product, 567
directed acyclic graph, 685
disambiguating policy, 248, 251
  left-longest, 248
  non-deterministic choice, 248
  right-longest, 248
discrete distribution, 657
discrete Fourier transform, 572
distinguishability, 687
DLPA, 227
do-support, 633–656
  definition, 636
document order, 213
document type declaration, 216
document type definition, *see* DTD
DOM, 275
domain, 62
Domain Analysis, 424
Domain Specific Language, 421
double quotes, 215
down-relation, 227
DSL design, 426, 428
DSL development phases, 422
DTD, 216, 218, 223
  external, 216
  internal, 216
dynamic data value comparison, 271
dynamic programming, 161

earliest detection location, 283
early detection location, 283
EDC, 402
EDCQ, 389
edge
  M-negative, 590
  M-positive, 590
  allowed, 589
  constant, 589
  cut edge, 613
  external, 588
  forbidden, 589
  impervious, 596
  internal, 588
  mandatory, 589
  of graph, 588
  viable, 596
edit distance, 159
EditCorQ, 402
element, 214, *see* XL element270
element content, 214
element name, 214
elementary component
  C-ear, 598
  external, 597
  family, 599
  internal, 597
  mandatory, 597
  of soliton graph, 597
  one-way, 598
  two-way, 598
  two-way accessible, 598
elementary net system, 362
  contact-free, 362
  with inhibitor arcs, 372
Empty(), 209
empty content, 214
empty element, 214
empty forest, 212
empty string, 208, 213
EN-system, 362
encoding, 212, 224
end tag, 214
end-tag, 275, 280, 281
ENI-system, 372
ENIM-system, 379
Epsilon Normal Form, 109, 110
equational monoid, 324
equivalence, 3
equivalence query, 392
equivalence relation, 320
EquQ, 392

event, 365
event-based processing, 280
event-driven run, 281
events, 275
Evolutionary Algorithm, 422, 430
evolutionary picture processors, 551
evolutionary processor, 530
exchange format, 206
existential queries, 252, 253
expected value, 658
extended regular tree grammars, *see* forest grammars
external predicates, 221, 224
extreme-value distribution, 649
extremum set, 576

factor, 63
family
  of elementary components, 599
  external, 599
  internal, 599
  root of, 599
father, 212
final states, 209, 227
FinInf, 393
finite automata
  alternating (AFA), 10–13, 15, 16, 19
  Boolean (BA), 11
  Chrobak normalform, 8, 22
  complete, 7, 12
  deterministic (DFA), 6, 7, 9, 11, 12, 15, 17, 18, 21, 22, 26–28, 30, 32
  Glushkov, 21
  limited NFA, 9
  multi-head, 48
  multiple entry
    deterministic (NDFA), 8, 9
    nondeterministic (NNFA), 9, 12
  nondeterministic (NFA), 6, 7, 9, 13, 15–18, 21, 22, 27, 30, 32
  position, 21
  sweeping, 19
  two-way, 16–19, 28
  two-way multi-head, 49
  unambiguous, 19
  unary, 7, 8, 15–19, 30
  visit, 19
finite automaton, 209
finite string automata, 223, 268
finite-state automata, 137
finite-state machines, 137
finite-state transducers, 137
finite-turn pushdown automata, 28
FinTxt, 393
firing sequence, 362
$First()$, 210
first moment, 658
first order logic, *see* FO
fixation, 634
flat documents, 259
flow relation, 361
FluXQuery, 293
FO, 252, 256, 257, 276
$Follow()$, 210
FOREG, 257
  guarded, 257
forest
  right-completion, *see* right-completion
forest automata
  bottom-up, 225
  pushdown forest automata, 225
forest derivation, 219, 232
forest derivations
  with external predicates, 222
forest grammars, 217, 218, 221, 224, 230, 232, 291
  and XML schema languages, 223
  definition, 217
  productions, 218
  queries, 251
  with external predicates, 221
forest states, 226
forests, 212
formal language, 3
formal system, 2
forward equivalent, 114
full word, 60, 62–64, 70, 71
functional programming

pattern matching, 248
Fxgrep, 207, 251, 258, 259, 277
and XPath, 272
conjunctions
paths, 261
qualifiers, 263, 264
disjunctions
paths, 261
negations
paths, 261
paths, 260
pattern language, 260
patterns, 260
primer, 260
qualifiers, 262–264
attributes, 263
context, 264
structure, 262
regular expressions
context qulaifiers, 264
names, 260
paths, 261
qualifiers, 262
start and end markers, 263
text nodes, 260
wildcards, 263

g-comtrace alphabet, 354
g-comtrace congruence, 355
gamma distribution, 649
gazetteer, 161
generalized star graph, 613
generalized tree, 605
generating function, 571
generating networks of evolutionary processors, 527
geometric distribution, 640
Glushkov $\mathbb{K}$-graph, 108
Glushkov automata, 21
Glushkov construction, 107
extended, 106
Glushkov graph, 112
Glushkov $\mathbb{K}$-graph, 120, 124–127
Glushkov WFA, 107, 108
Glushkov/Yamada-McNaughton construction, 186
grafting, 244
grammar
LL(k), 48
LR(k), 33
context-free, 4, 29
monotone, 38
phrase structure, 38
right-linear context-free, 30, 34
self-embedding context-free, 36
grammar queries, 206, 230, 267
Grammatical Inference, 421, 429
graph, 588
2-edge component, 620
n-star, 610
bipartite, 588
bipartition, 588
bridge, 613
chestnut, 600
closed, 588
factor-critical, 621
generalized star graph, 613
open, 588
redex, 603
reducible, 112–114
soliton graph, 593
strongly stable, 126
strongly transverse, 126
subgraph, 589
graph without orbit, 112
graph-controlled insertion-deletion systems, 464
insertion-only, 509
with priorities, 466
greedy maximally concurrent form, 342
Greibach normalform, 37
grep, 259
gso-structure, 333
guessing automaton, 155
GUI, 259

hammock, 111, 112
hashing function, 144, 150
minimal perfect, 150
hedge automata, 219, *see* forest automata

Hellinger distance, 692
Hermitian matrix, 567
Hilbert space, 567
Hoeffding's bound, 689
hole, 62
homomorphic image of
  automaton, 591
homomorphically complete
  class of automata, 592
homomorphism of
  automaton, 591
horizontal path formulas, 257
hypertree, 445
hypothesis test, 653
hypothesis tests, 635

incremental construction, 167
incremental construction from sorted
  data, 168
incremental construction from
  unsorted data, 175
incremental learning, 430
indicator function, 658
inhibitor arc, 372
initial state, 209
initial states, 227
inner product, 567
insertion systems, 497
  graph-controlled, 509
insertion-deletion systems, 463
  context-free, 464, 472
  graph-controlled, 464
  insertion-only, 497
  one-sided, 464, 481
intermediate path formula, 257
internal, 216
invalid computations, 43
  INVALC_A, 43
  INVALC_C, 43
  INVALC_R, 43
invariant closure, 330
isolated cut point, 569
isolated-word correction, 158
isomorphically complete
  class of automata, 592
isomorphism
  of automaton, 591
  strong, 591
iterated a-substitution, 36
iterated substitution expression, 36
iterative arrays, 50

$\mathbb{K}$-expression
  proper, 105
  valid, 105
$\mathbb{K}$-graph, 106, 108, 119, 120
  $\mathbb{K}$-balanced, 126
  $\mathbb{K}$-reducible, 119, 120, 127
  $\mathbb{K}$-stable, 124, 126
  $\mathbb{K}$-transverse, 125, 126
  strongly $\mathbb{K}$-balanced, 126
k-reversible language, 391
$\mathbb{K}$-rules, 114
lBCorQ, 401
k-Rev, 391
Kullback-Leibler divergence, 691

L-attributed, 291
label, 212
labeling, 232, 287
labelled firing sequence, 366
labelled set, 319
labelled step sequence, 366
labelling, 319
Landau's function, 7
languages
  Boolean closure of CFL, 48
  bounded, 28
  Church-Rosser, 49
  finite, 7, 8
  simple, 24
  ultralinear, 28
  unary, 7, 8
lazy evaluation, 249
LBCorQ, 401
LBCQ, 401
leaf, 212, 213
learning grammar versions, 431
least-squares curve fit, 649
left sibling, 212
left-to-right pushdown forest
  automaton, *see* LPA

length, 105
  litteral length, 105, 106
lexicographic order, 213, 283, 287, 303
lexicon, **134**
likelihood, 644
LimInf, 392
limited nondeterminism, 9
LimTxt, 392
linkage constraint, 675
linkage disequilibrium, 667
linkage pattern, 673
local minimization, 169
locations, 214
logical formalism, 256
logistic differential equation, 637
logistic population model, 637, 638, 645
logistic sigmoid, 635, 637, 645
  definition, 637
longest common factor, 676
loop
  secondary, 604
lower bound, 5
lower bound techniques, 22
LPA, 226–228, 278, 280
  deterministic, *see* DLPA
  for XML streams, 278
LPath, 275
L^*, 388

μ-formulas, 229
MAGIc, 430, 432
magic numbers, 2
marking, 361
  enabled, 361
Markov chain, 635–653
  absorption, 646
  definition, 638
  stationary distribution, 646
Markov chain Monte Carlo, 646
match-relevant non-terminals, 280
matches
  alternative definition, 300
  of k-ary queries, 232
  of unary queries, 231
matches of k-ary queries, 232
matching
  covered vertex, 589
  in graph, 588
  perfect, 589
  perfect T-matching, 625
  perferct internal, 589
mathematical model, 633
matrix
  input matrix, 125
  orbit matrix, 124, 126
  output matrix, 125
maximal patterns, 673
maximum likelihood method, 633, 635, 644–650
Mazurkiewicz trace, 324
mean, 658
measure-once, 564, 568
membership query, 392
Memetic Algorithm, 422, 430
MemQ, 392
Middle English, 636, 639
minimal automaton, 686
minimal period, 62
minimization
  local, 169
minimization algorithm, 692
minimization of automata, 167
mixed content, 215
moderate pushdown automata, 25, 27
Modern English, 639
monadic second order logic, *see* MSO
monoid, 323
  equational, 324
  null monoid, 608
  partially commutative, 324
  trivial null monoid, 608
monoid of step sequences, 323
monotone grammar, 38
Monte Carlo method, 645–650
  definition, 645
Moran model, 637, 638
morphological analysis, 153
morphological synthesis, 153
MQ, 402
MSO, 233, 251–258, 276, 277
  guarded, 257

Presburger MSO, 258
multi-head finite automata, 48
multiple entry
deterministic finite automata, 8
nondeterministic finite automata, 12
multiple entry states, 8, 12
mutex arc, 379

nearest common ancestor, 242
Nerode equivalence, 687
net, 361
networks of evolutionary processors, 525
networks of picture processors, 527
networks of splicing processors, 527
NFA, 209–212, 228, 249, 279, 280, 296, 302
nil, 224, 232
node, 213
nodes, 213
non-deterministic finite automaton, *see* NFA
non-recursive trade-off, 5, 40
ambiguity bounded PDA, 47
branching bounded PDA, 47
cellular automata, 50
CFG, 46, 47
Church-Rosser languages, 49
deterministic space bounded TMs, 46
DFA/NFA, 41, 42, 46, 48
DPDA, 45, 47, 49
finite-turn PDA, 48
iterative arrays, 50
linear CFG, 41, 42
multi-head FA, 49
one-turn PDA, 42, 48, 49
PDA, 48, 49
proof scheme, 40, 44
restart automata, 49
two-way multi-head NFA/DFA, 49
unambiguous CFG, 45, 47
unary two-way multi-head NFA/DFA, 49
verified ambiguous CFG, 47
verified PDA, 47
verified unambiguous CFG, 47
non-terminal symbols, 217
non-terminals, 219
nondeterministic finite automata, 6
lower bound techniques, 22
norm, 567
normal distribution, 644
normal matrix, 567
normalform
λ-free, 37
Chomsky, 29, 37
Chrobak, 8, 22
Greibach, 37
position restricted, 37
restricted, 37
right-linear, 34
numbered automaton, 155
numerical document queries, 258

observable, 569
occurrence net, 365
one-match policy, 249
one-pass query evaluation, 276
open conjugated system, 625
operation problem, 2
orbit, 111, 112, 119, 126, 128
maximal, 111–113, 125, 126
stable, 112
strongly stable, 112
strongly transverse, 112
transverse, 112, 125
ordered, 212, 258
ordered system
of elementary soliton automata, 608
constant, 608
orthogonal subspace, 567
orthograhic errors, 158
orthonormal subspace, 567

$\mathbb{P}$, 391
PACS learning, 430
Parallel communicating grammar systems, 526
parsers, 216

partial word, 62
partial word(s), 59–99
partially commutative monoid, 324
partially ordered set, 320
partition, 70
path, 213
  alternating, 590
    external, 590
    positive, 590
  in graph, 588
pattern language, 391
pattern translation, 267
pattern-language, 273
PCorQ, 394
$PCorQ^{\mathbb{A}}$, 395, 400
PCQ, 388
PDF, 206
period, 62, 63
periodic event, 570
periodic language, 580
permutation group, 601
  block, 601
  primitive, 601
  trivial, 601
Petri net, 361
phrase structure grammar, 38
picture languages, 549
place, 361
Pol0BCorQ, 411
Pol(l-1)BCorQ, 411
PollBCorQ, 404
PolEditCorQ, 404
PolMemQ, 404
PolPCorQ, 404
population genetics, 637
poset, 320
  intersection, 320
  linearisation, 320
  stratified, 320
  total, 320
position automata, 21
position restricted normalform, 37
positive semidefinite matrix, 568
posterior distribution, 644
postorder, 167
powerset construction, 7, 12, 32
Pref$(\cdot)$, 400
Presburger formulas, 258
Presburger tree automata, 258
primaries, *see* primary match
primary match, 239
primitive partial word, 62
prior distribution, 644
  uninformative, 644
probabilistic deterministic finite automaton, 679
probabilistic finite automaton, 679
probabilistic non-deterministic finite automata, 679
probability density function, 657
probability distribution, 644, 656
probability mass function, 657
process, 367
processing instruction, 270
processing instruction nodes, 215
processing instructions, 218
product of automata
  α_i-product, 592
  disjoint ϵ-product, 592
  feedback function, 591
  general ε-product, 592
  general product, 591
  ordered ϵ-product, 608
  quasi-direct product, 592
  strict, 592
productions, 218, 221
  meaning, 220, 222
proof scheme non-recursive trade-offs, 40, 44
protective linkage patterns, 672, 674
protective patterns, 674
pumping lemma, 44
pushdown automata, 4, 23–25
  ambiguity bounded, 47
  bounded, 29
  branching bounded, 47
  deterministic (DPDA), 23, 24, 26, 28
  finite-turn (FTPDA), 28, 29, 32, 33
  moderate, 25, 27
  nondeterministic (PDA), 23, 24
  normalform, 27–29, 31

real-time, 24, 31
unary, 27, 28, 31
verified, 47
pushdown forest automata, 225, 249
languages of, 227
right-to-left, *see* RPA

quantum automaton, 568
quantum computing, 563
quasi-ε-equivalent, 115
queries
k-ary queries, 206, 230, 253, 256–258
definition, 232
evaluation, 248
binary queries, 232, 247, 248
evaluation, 247
grammar queries, 230, 251, 276
linguistics, 274
logical formalisms, 256
numerical document queries, 258
right-ignoring, 276, 278, 282
unary queries, 230, 252, 255, 256
definition, 231
query automata, 252
quotient monoid, 323

random variable, 644, 656
random walk, 641
ranked, 212, 224, 253, 255
reachable marking, 362
recognizer, 140
recombination event, 668
redex
center of, 603
connected, 618
focal edges, 603
focal vertices, 603
implied redex, 618
in a graph, 603
redex grove, 618
maximum, 618
shrinking, 603
reference constraints, 224
register automata, 256
register machine, 462
regular expression, 20–22, 186
lower bound technique, 22
regular expressions, 207, 227, 229, 257, 259
languages of, 208
regular forest languages, 217, 221, 225
and XML languages, 217, 230
regular hedge expressions, 254
regular path expressions, 261
regular ranked tree languages, 233
regular string language, 208
regular tree grammars, 224, 232, 233
regular tree languages, 225
regular vertical path, 268
relabeling, 219
relational invariant, 329
relational tuple, 318
RelaxNG, 222, 224
restarting automata, 19, 49
restricted normalform, 37
right sibling, 212
right-completion, 283
right-ignoring, 278, 280
match, 302
right-linear grammar, 34
right-linear normalform grammar, 34
right-to left pushdown forest automaton, *see* RPA
root element, 214
row, 406
RPA, 227
rules, 112
run, 253, 300
suffix run, 302
runs, 219, 252

s-measure, 4
Sakoda and Sipser problem, 17
SAX, 275
schema, 216
schema languages, 217
SDI, 292
secondaries, *see* secondary match
secondary match, 239
selecting automata, 258
selecting states, 252

selecting tree automata, 252
selective dissemination of information, *see* SDI
self-transition
  in soliton automata, 616
semantical framework, 362
sequence, 322
  empty, 322
Sequential XPath, 293
Sequitur Algorithm, 431
serialisability, 337
set of holes, 62
siblings, 212
side-relation, 227
$\Sigma^{\leq k}$, 389
sigmoid, *see* logistic sigmoid
simple binary query, 239
simple language, 24
simultaneity, 337
single quotes, 215
size
  of descriptors, 4
  of grammar, 30
size, 390
sn-measure, 40
so-structure, 332
social network, 655
social networks, 635
software engineering, 421
soliton automaton, 596
  constant, 606
  deterministic, 596
  self-transition, 616
  strongly deterministic, 596
soliton graph, 593
  1-extendable, 597
  canonical class, 597
  canonical equivalence, 597
  canonical partition, 597
  deterministic, 596
  elementary, 597
  elementary component, 597
  pruning, 621
  state, 593
  strongly deterministic, 596
  tree-decomposition, 602
  trivial elementary, 597
  viable, 597
soliton trail, 606
  c-trail, 606
  l-trail, 606
  alternating double soliton c-trail, 606
  cycle of, 606
  handle of, 606
soliton walk, 596
spelling correction, 158
spelling errors, 158
SPEX, 293
star height, 20, 22, 23
Star Normal Form (SNF), 104, 108, 109
start expression, 218
start tag, 214, 275, 277, 280, 281
state
  unreachable, 183
state elimination technique, 22
state merging algorithm, 693
state superposition, 569
states, 209, 219
static data value comparison, 271
stay transitions, 252
step, 322, 361
  alphabet, 323
  sequence, 322, 362
stochastic event, 569
stochastic matrix, 638
stratified order structure, 332
stratified poset extension, 332, 333
StreamGlobe, 293
strings, 208
structural constraints, 268
  conjunctions, 268
STX, 293
subautomaton, 591
subgraph, 589
  induced, 589
  induced by, 589
  nice, 589
succinctness, 2, 4
suffix run, 302
support, 105

sweeping automata, 19
sym, 210
symbols, 208
synchronic, 635
Szpilrajn's Theorem, 321

tag, 214
Tail, 389
target processor, 215
terminal symbols, 217
text, 215
text content, 215
text nodes, 221, 222, 270
tgrep, 274
tgrep2, 274
Thompson construction, 186
timed accepting networks of
  evolutionary processors, 533
tissue-like membrane systems, 526
top-down, 252, 253
trace alphabet, 335, 370
trace monoid, 335
trail
  in graph, 588
transducer, 138, **147**, 153
  p-subsequential, 148
  subsequential, 148
TransformX, 291
transition, 361
transition matrix, 569, 638
transition monoid
  of automaton, 591
transition relation, 209, 210
transpose matrix, 566
tree derivation, 218, 232
tree derivations
  with external predicates, 222
tree grammars, 253
tree queries, 258
tree states, 226
tree-walking automata, 255
  marble/pebble automata, 256
trie, 137, **141–142**, 166
triple construction, 31
TurboXPath, 293
Turing machine, 42
two-sided infinite word, 63
two-way finite automata, 16
  multi-head, 49
two-way tree automata, 252
typograhical errors, 158

ultralinear language, 28
unambiguous, 231, 253, 291
  content models, 223
unambiguous automata, 19
unary language, 570
unary quantum automaton, 569
unavoidable set, 60
Unicode, 221, 222
uniform distribution, 644
uninformative prior distribution, 644
uniqueness constraints, 224
unitary matrix, 567
universal queries, 252, 253
unordered, 258
unranked, 212
unranked trees, 252
unreachable state, 183
up-relation, 227
upper bound, 5

VALC, 398
valid, 216
valid computations, 42
  VALC_A, 43
  VALC_C, 43
  VALC_R, 43
validation, 216
  XML streams, 281, 291
VARBRUL, 635, 654
variability, 441
variable length finite automata, 682
variable length memory, 682
variable length processes, 681
vector
  head-orbit vector, 124
  input vector, 125, 126
  output vector, 125, 126
  tail-orbit vector, 124
verb-raising, 636–656
  definition, 636

verified pushdown automata, 47
vertex
  external, 588
  internal, 588
    accessible, 590
  of graph, 588
vertical path formulas, 257
very large documents, 276
very large XML documents, 226
visit automata, 19
visXcerpt, 259

walk
  empty, 588
  external alternating, 595
    switching on, 595
  in graph, 588
  soliton walk, 596
  total external alternating walk, 625
Weighted Finite Automaton (WFA), 103, 104, 106–108, 110, 114, 124
white spaces, 214
wildcards
  in grammars, 222
word, 3, 62
  empty, 3
  length, 3
  reversal, 3

X^2, 258
XDuce, 248
XFilter, 292
Xing, 259
XML
  streams, 259
XML element, 270
XML language, 216
XML Schema, 224
XML schema, 206, 222
XML schema languages, 216
XML Stream Attribute Grammars, 291
XML streams, 207, 275, 276, 280
XML-GL, 259
XPath, 206, 231, 251, 258, 259, 269, 277
  abbreviated syntax, 270
  and Fxgrep, 272
  axes
    forward, 269
    reverse, 269
  Core, 274
  dynamic data value comparison, 271
  node test, 270
  predicate, 270, 271
  qualifiers, 271
  static data value comparison, 271
XPush, 292
XQuery, 206
XSLT, 206, 215
XSM, 293
XStreamQuery, 293
XTrie, 292

YFilter, 292

Mathematics, Computing, Language, and Life:
Frontiers in Mathematical Linguistics and Language Theory
Vol. 2

Scientific Applications of Language Methods

Presenting interdisciplinary research at the forefront of present advances in information technologies and their foundations, *Scientific Applications of Language Methods* is a multi-author volume containing pieces of work (either original research or surveys) exemplifying the application of formal language tools in several fields, including logic and discrete mathematics, natural language processing, artificial intelligence, natural computing and bioinformatics.

P707 hc
ISBN-13 978-1-84816-544-1
ISBN-10 1-84816-544-7
9 781848 165441
ISSN 2042-1044

Imperial College Press
www.icpress.co.uk