Lecture Notes in Computer Science 2962
Edited by G. Goos, J. Hartmanis, and J. van Leeuwen

Springer
Berlin
Heidelberg
New York
Hong Kong
London
Milan
Paris
Tokyo

Stefano Bistarelli

Semirings for Soft Constraint Solving and Programming

Springer

Author

Stefano Bistarelli
Università degli Studi "G. D'Annunzio" di Chieti-Pescara
Dipartimento di Scienze
Viale Pindaro, 42, 65127 Pescara, Italy
E-mail:bista@sci.unich.it
and
Istituto di Informatica e Telematica, C.N.R.
Via G. Moruzzi, 1, 56124 Pisa, Italy
E-mail: stefano.bistarelli@iit.cnr.it

Cataloging-in-Publication Data applied for

A catalog record for this book is available from the Library of Congress.

Bibliographic information published by Die Deutsche Bibliothek
Die Deutsche Bibliothek lists this publication in the Deutsche Nationalbibliografie; detailed bibliographic data is available in the Internet at <http://dnb.ddb.de>.

CR Subject Classification (1998): D.3.3, D.1.6, D.3.1, D.3, F.3.1, D.2.4, I.2.2-3

ISSN 0302-9743
ISBN 3-540-21181-0 Springer-Verlag Berlin Heidelberg New York

Typesetting: Camera-ready by author, data conversion by DA-TeX Gerd Blumenstein
Printed on acid-free paper SPIN: 10988022 06/3142 5 4 3 2 1 0

Stefano Bistarelli

Semirings for Soft Constraint Solving and Programming

July 14, 2004

Springer-Verlag
Berlin Heidelberg New York
London Paris Tokyo
Hong Kong Barcelona
Budapest

Foreword

Constraint satisfaction and constraint programming have shown themselves to be very simple but powerful ideas. A constraint is just a restriction on the allowed combinations of values for a set of variables. If we can state our problem in terms of a set of constraints, and have a way to satisfy such constraints, then we have a solution. The idea is general because it can be applied to several classes of constraints, and to several solving algorithms. Moreover, it is powerful because of its unifying nature, its generality, its declarative aspects and its application possibilities. In fact, many research and application areas have taken advantage of constraints to generalize and improve their results and application scenarios.

In the last 10 years, however, this simple notion of constraint has shown some deficiencies concerning both theory and practice, typically in the way over-constrained problems and preferences are treated. When a problem has no solution, classical constraint satisfaction does not help. Also, classical constraints are not able to model conveniently problems which have preferences, for example over the selection of the most relevant constraints, or about the choice of the best among several solutions which satisfy all the constraints.

Not being able to handle non-crisp constraints is not just a theoretical problem, but it is also particularly negative for applications. In fact, over-constrained and preference-based problems are present in many application areas. Without formal techniques to handle them, it is much more difficult to define a procedure which can easily be repeated to single out an acceptable solution, and sometimes it is not even possible.

For this reason, many researchers in constraint programming have proposed and studied several extensions of the classical concepts in order to address these needs. This has led to the notion of *soft constraints*. After several efforts to define specific classes of soft constraints, like fuzzy, partial and hierarchical, the need for a general treatment of soft constraints became evident, a treatment that could model many different classes altogether and to prove properties for all of them. Two of the main general frameworks for soft constraints were *semiring-based soft constraints* and *valued constraints*.

This book is a revised, extended version of the Ph.D. thesis of Stefano Bistarelli, whom we had the pleasure to supervise at the University of Pisa. It focuses mainly on the semiring-based soft constraint formalism, also comparing it with many of the specific classes and also with valued constraints. Semiring-based soft constraints are so called because they are based on an underlying semiring structure, which defines the set of preferences, the way they are ordered, and how they can be combined. This concept is very general and can be instantiated to obtain many of the classes of soft constraints that have already been proposed, including their solution algorithms, and also some new ones.

The book includes formal definitions and properties of semiring-based soft constraints, as well as their use within constraint logic programming and concurrent constraint programming. Moreover, it shows how to adapt to soft constraints some existing notions and techniques, such as abstraction and interchangeability, and it shows how soft constraints can be used in some application areas, such as security.

This book is a great starting point for anyone interested in understanding the basics of semiring-based soft constraints, including the notion of soft constraint propagation, and also in getting a hint abut the applicability potential of soft constraints. In fact, it is the first book that summarizes most of the work on semiring-based soft constraints. Although most of its content also appears in published papers, this is the only place where this material is gathered in a coherent way.

This book is the result of several threads of collaborative work, as can be seen from the many publications that are cited in the bibliography and whose content is reflected in the book. Therefore many authors have contributed to the material presented here. However, Stefano Bistarelli succeeded in providing a single line of discourse, as well as a unifying theme that can be found in all the chapters. This uniform approach makes the material of this book easily readable and useful for both novice and experienced researchers, who can follow the various chapters and find both informal descriptions and technical parts, as well as application scenarios.

November 2003

Ugo Montanari[1] and Francesca Rossi[2]

[1] Dipartimento di Informatica
Universita' di Pisa
Italy

[2] Dipartimento di Matematica Pura ed Applicata
Universita' di Padova
Italy

Preface

The *Soft Constraints* idea is able to capture many real-life situations that cannot be represented and solved with the classical crisp constraint framework. In this book we first describe a general framework for representing many soft constraint systems, and then we investigate the related theoretic and application-oriented issues.

Our framework is based on a semiring structure, where the carrier of the semiring specifies the values to be associated with each tuple of values of the variable domain, and the two semiring operations, $+$ and $\times$, model constraint projection and combination, respectively. The semiring carrier and operations can be instantiated in order to capture all the *non-crisp* constraints representing fuzziness, optimization, probability, hierarchies, and others. The solution of each instance of the soft Constraint Satisfaction Problem (CSP) is computed by using the appropriate $\times$ and $+$ semiring operation.

This uniform representation can be used to give sufficient conditions for the correctness and applicability of local consistency and dynamic programming algorithms. In particular:

- We show that using an idempotent $\times$ operator the classical local consistency (and also dynamic programming) techniques can be used to reduce the complexity of the problem without modifying its solution.
- We adapt to the soft framework partial local consistency and labeling techniques, which require fewer pruning steps of the domain. This means that, although they are able to remove fewer non-optimal solutions than classical algorithms can, partial local consistency algorithms can be beneficial because they are faster and easier implemented.
- We extend general local consistency algorithms that use several pruning rules until the fix-point is reached.

Solving a soft CSP is generally harder than solving the corresponding crisp CSP. For this reason we introduce an abstraction/concretization mapping over soft CSPs in order to solve a problem in an easier environment and then use the abstract results to speed up the search of solutions in the concrete one. Several mappings between existing soft frameworks are given. These mappings will especially be useful for applying soft local consistency techniques in a safe, easy, and faster way. Also useful, when looking for optimal solutions, are the notions of *substitutability* and *interchangeability*. In crisp CSPs they have been used as a basis for search heuristics, solution adaptation, and abstraction techniques.

The next part of the book involves some programming features: as classical constraint solving can be embedded into Constraint Logic Programming (CLP) systems, so too can our more general notion of constraint solving be handled within a logic language, thus giving rise to new instances of the CLP scheme. This not only gives new meanings to constraint solving in CLP, but it also allows one to treat in a uniform way optimization problem solving within CLP, without the need to resort to ad hoc methods. In fact, we show that it is possible to generalize the semantics of CLP programs to consider the chosen semiring and thus solve problems according to the semiring operations. This is done by associating each ground atom with an element of the semiring and by using the two semiring operations to combine goals. This allows us to perform in the same language both constraint solving and optimization. We then provide this class of languages with three equivalent semantics, model-theoretic, fix-point, and proof-theoretic, in the style of CLP programs. The language is then used to show how the soft CLP semantics can solve shortest-path problems. In a way similar to the soft CLP language we also extend the semantics of the Concurrent Constraints (cc) language. The extended cc language uses soft constraints to prune and direct the search for a solution.

The last part of the book aims to describe how soft constraints can be used to solve some security-related problems. In the framework, the crucial goals of *confidentiality* and *authentication* can be achieved with different levels of security. In fact, different messages can enjoy different levels of confidentiality, or a principal can achieve different levels of authentication with different principals.

Acknowledgement. This monograph is a revised and extended version of my doctoral dissertation which was submitted to the University of Pisa Computer Science Department and accepted in March 2001.

The results presented here have been influenced by many people and I would like to take this opportunity to thank them all.

I wish to thank first of all the supervisor of my Ph.D. thesis Ugo Montanari; the main part of this book came from the research performed under his significant guidance.

I want also to thank Francesca Rossi, my unofficial supervisor; she shared with me the passion for constraints. Many of the ideas collected in this book came from ideas we developed together.

A special thanks is also due to all the friends who shared with me some of the results I collected in this book: Giampaolo Bella, Boi Faltings, Rosella Gennari, Hélène Fargier, Yan Georget, Nicoleta Neagu, Elvinia Riccobene, Thomas Schiex, and Gérard Verfaillie.

Many thanks are due to the external reviewers of my Ph.D. thesis, Philippe Codognet and Pascal Van Hentenryck; they carefully read a preliminary version of the thesis and provided many useful comments and suggestions.

My warmest thanks go to my friend Olga Petosa; she was so kind to read all the conversational parts of my work, correct some typing errors, and make it just a little nicer!

Finally, I am also grateful both to the anonymous referees of this book and to Jan van Leeuwen; they gave me a number of hints on a draft version of the book which helped to improve the final presentation.

Some thanks also to Anna Kramer and Alfred Hofmann who helped me with the last details needed for the publication of this book.

December 2003 Stefano Bistarelli

Contents

List of Figures

List of Tables

1. Introduction

"Constraint programming represents one of the closest approaches computer science has yet made to the Holy Grail of programming: the user states the problem, the computer solves it."
Eugene C. Freuder, Constraints, April 1997

Overview

Constraint programming is an emergent software technology for declarative description and effective solving of large, particularly combinatorial, problems especially in areas of planning and scheduling. It has recently emerged as a research area that combines researchers from a number of fields, including Artificial Intelligence, Programming Languages, Symbolic Computing and Computational Logic. Constraint networks and constraint satisfaction problems have been studied in Artificial Intelligence starting from the seventies. Systematic use of constraints in programming has started in the eighties. In constraint programming the programming process consists of the generation of requirements (constraints) and solution of these requirements, by specialized constraint solvers.

Constraint programming has been successfully applied in numerous domains. Recent applications include computer graphics (to express geometric coherence in the case of scene analysis), natural language processing (construction of efficient parsers), database systems (to ensure and/or restore consistency of the data), operations research problems (like optimization problems), molecular biology (DNA sequencing), business applications (option trading), electrical engineering (to locate faults) and circuit design (to compute layouts).

This book is centered around the notion of constraint solving [189] and programming [141]. The interest in constraint satisfaction problems can be easily justified, since they represent a very powerful, general, and declarative knowledge representation formalism. In fact, many real-life situations can be faithfully described by a set of objects, together with some constraints among them. Examples of such situations can be found in several areas, like VLSI, graphics, typesetting, scheduling, planning, as well as CAD, decision-support systems and robotics.

1.1 From the Beginning ...

The constraint idea comes from the early 1960's when Sutherland introduced Sketchpad [187], the first interactive graphical interface that solved geometric constraints. After that came Fikes' REF-ARF [97] and at the beginning of 1970's Montanari described fundamental properties of the constraints when applied to picture processing [149]. Another study in finite domain constraint satisfaction was done at the end of 1970's in Laurière's ALICE [133], a system developed to solve prediction/detection problems in geology. After these, several constraint languages have been proposed in the literature: the language of Steele ([184]), CONSTRAINTS [186] of Sussman & Steele, Thinglab ([60]) and Bertrand ([134]).

From Sketchpad until now, a lot of research has been done and improvements made, and the classical constraint satisfaction problems (CSPs) [139, 150] have been shown to be a very expressive and natural formalism to specify many kinds of real-life problems.

1.2 Applications

Today, the use of the constraint programming idea to solve many real-life problem is reality. Many important companies develop tools based on the constraint technology to solve *assignment*, *network management*, *scheduling*, *transport* and many other problems:

1.2.1 Assignment Problems

Assignment problems were one of the first type of industrial applications that were solved with the CLP technology. These problems usually have to handle two types of resources, and constraints among them, and try to assign one resource of the first kind to one of the second kind such that all constraints are satisfied.

An example is the stand allocation for airports, where aircrafts (the first kind of resources) must be parked on the available stands (the second kind of resources) during their stay at the airport. The first industrial CLP application was developed for the HIT container harbor in Hong Kong [161], using the language CHIP: the objective was to allocate berths to container ships in the harbor, in such a way that resources and stacking space for the containers is available. Other Hong Kong applications are at the airport, where a CHIP-based system is used to solve the counter allocation problem [69], and another constraint-based system, which uses the ILOG libraries, is used for the stand allocation problem since mid-1998 [70]. Another system, called APACHE [88], was a demonstrator for stand allocation at Roissy airport in Paris: the objective was to replan the allocation when a change of arrival/departure times occurred.

1.2.2 Personnel Assignment

Personnel assignment problems are a special case of assignment problems where one resource type consists of humans. This peculiarity makes them specific enough to be considered separately. In fact, changing work rules and regulations impose difficult constraints which evolve over time. Also, user preferences often lead to over-constrained problems, which have no solution satisfying all constraints. Another important aspect is the requirement to balance the work among different persons, which leads to hard optimization problems.

The Gymnaste system [65] produces rosters for nurses in hospitals, and is used in many hospitals in France. Around 250 technicians and journalists of the French TV and radio station RFO are scheduled with the OPTI-SERVICE system [73], which generates weekly workplans from the individual activities which must be covered by different persons with different qualifications. The personnel for the TGV high-speed train bar and restaurant services is scheduled with the EPPER application [92]: all services for a month are covered by a team of 250 people. Recently a distributed CLP system has been used to tackle a workforce management problem within British Telecom [131]. Also Telecom Italia is using a constraint-based system to schedule all its technical tasks (about 100.000 tasks involving 20.000 technicians) over its territory [113]; the system which controls the whole workforce management has a scheduling module, called ARCO, which dispatches activities to technicians, and which makes extensive use of constraint propagation techniques.

1.2.3 Network Management

Another application domain for finite domain CLP is network management, where many different problems can be addressed and solved using CLP.

The LOCARIM system was developed by COSYTEC for France Telecom: starting from an architectural plan of a building, it proposes a cabling of the telecommunication network of the building. The PLANETS system, developed by the University of Catalonia in Barcelona for the Spanish electricity company, is a tool for electrical power network reconfiguration which allows to schedule maintenance operations by isolating network segments without disrupting customer services. The company Icon in Italy produces a load-balancing application which is controlling network flow for the inter-banking system in Italy. The Esprit project CLOCWiSe (IN 10316I) is using CHIP for the management and operational control of water systems. The planning of wireless digital networks for mobile communication has been tackled by the system POPULAR [110], written in ECLiPSe and then in CHR: the main advantages with respect to other approaches to this same problem (using traditional imperative programming) are a good balance among flexibility, efficiency, and rapid prototyping.

1.2.4 Scheduling Problems

Perhaps the most successful application domain for finite domain CLP are scheduling problems. Given a set of resources with given capacities, a set of activities with given durations and resource requirements, and a set of temporal constraints between activities, a “pure” scheduling problem consists of deciding when to execute each activity, so that both temporal constraints and resource constraints are satisfied.

A typical example of a constraint-based scheduling application is ATLAS [181], which schedules the production of herbicides at the Monsanto plant in Antwerp. The PLANE system [26] is used by Dassault Aviation to plan the production of the military Mirage 2000 jet and the Falcon business jet. The objective is to minimize changes in the production rate, which has a high set-up cost, while finishing the aircraft just in time for delivery. The MOSES application was developed by COSYTEC for an animal feed producer in the UK: it schedules the production of compound food for different animal species, eliminating contamination risk and satisfying costumer demand with minimal cost. The FORWARDC system is a decision support system, based on CHIP, which is used in three oil refineries in Europe to tackle all the scheduling problems occurring in the process of crude oil arrival, processing, finished product blending and final delivery [115]. Recently, Xerox has adopted a constraint-based system for scheduling various tasks in reprographic machines (like pothocopiers, printers, fax machines, etc.); the role of the constraint-based scheduler is to determine the sequence of print making and to coordinate the time-sensitive activities of the various hardware modules that make up the machine configuration [109]. Recent results on the tractability of classes of constraint problems have shown that such scheduling problems are indeed tractable, and thus amenable for an efficient solution [164].

1.2.5 Transport Problems

A variety of transport problems have been tackled using constraints. These problems are often very complex due to their size, the number and variety of constraints, and the presence of complex constraints on possible routes. Moreover, often these problems have a personnel allocation problem as a sub-aspect, usually with complex safety and union regulations.

The COBRA system [180] generates diagrams representing workplans for train drivers of North Western Trains in the UK. For each week, around 25.000 activities must be scheduled in nearly 3.000 diagrams, taking a complex route network into account. The DAYSY Esprit project (8402) and the SAS-Pilot program [14] considers the operational re-assignment of airline crews to flights. This same problem is tackled also by another system [100], which uses a combination of CLP and OR techniques. A recently developed system uses the ILOG constraint libraries to produce and optimize train operating plans for a freight railway company, by creating transport movements for rolling stock between sources, hubs and sinks while satisfying transport requirements [143].

1.3 Crisp Constraints

In this section we review some definitions and notions which will be fundamental for our work. The section is divided in three parts: In the first one we define the Constraint Satisfaction Problems (CSPs) and we introduce the classical solving techniques; In the second one, we introduce Constraint Logic Programming

(CLP) framework; lastly, we describe the concurrent constraint (cc) framework. There is also some other background material that will be used later in the book. That material, however, will be introduced only when needed.

1.3.1 CSPs

Constraint Satisfaction Problems (CSPs) are a very powerful and general knowledge representation formalism, since many real situations can be faithfully described by a set of objects, together with some constraints among them. They were a subject of research in Artificial Intelligence for many years starting from the 1970s. From that date several classes of constraints were studied (e.g., temporal constraint problems, constraint problems over intervals and over reals, linear constraints) but in this book we will concentrate on the specific class of *constraints over Finite Domain.*

We believe that finite domain constraint problems [85,105,138,150,152], i.e., constraint problems where the objects may assume a finite number of distinct configurations, are of special interest, since they are significantly simpler to analyze than general constraint problems, while still being able to express, either faithfully or through a certain degree of abstraction, the main features of several classes of real life problems.

Before providing the formal definition of the CSP and its solution methods, let's look at some examples [168], also to illustrate the variety of the potential application fields.

Example 1.3.1 (8-queens). Famous test-problem popular also in the CSP world is the 8-queens problem: place 8 queens on the chess board such that they do not attack each other (see Figure 1.1. In order to formulate this problem as a CSP, the location of the queens should be given by variables, and the "do not attack each other" requirement should be expressed in terms of a number of constraints. A simple way to do this is to assign a variable to each queen.

As the 8 queens must be placed in 8 different columns, we can identify each queen by its column, and represent its position by a variable which indicates the row of the queen in question. Let x_i stand for the row of the queen in the i-th column. The domain of each of the variables $x_1, \ldots, x_8$ is $\{1, 2, ...8\}$. For any two different variables the following two constraints must hold, expressing that the queens should be in different rows and on different diagonals:

$$x_i \neq x_j$$
$$|x_i - x_j| \neq |i - j|$$

In this formulation of the problem, we have to find a solution out of the total possible instantiations of the variables, which is 8^8. This formulation, though seems natural, does contain a trick: a part of the requirements of the problem is reflected in the representation, not in the constraints. We could have used the most straightforward representation, namely identifying the squares of the chess board by the $1, 2, \ldots, 64$ numbers, and having 8 variables for the 8 queens all

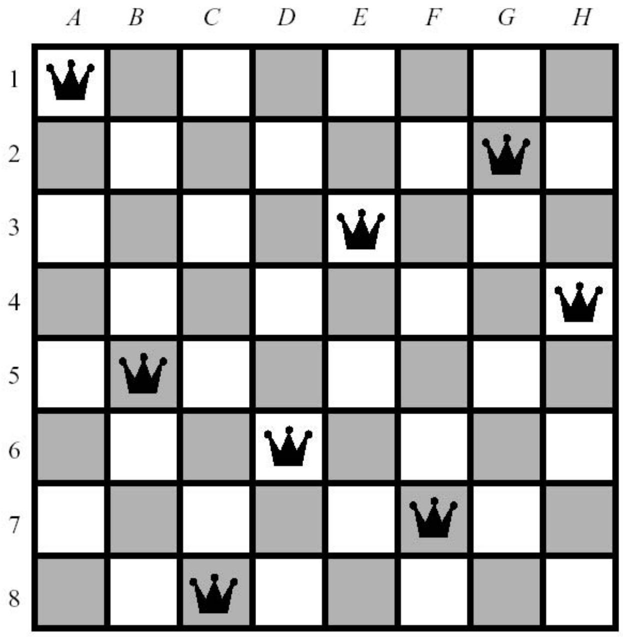


Fig. 1.1. A possible solution to the 8-queens problem

with the domain $\{1, 2, \ldots, 64\}$. In this case, the "different columns" requirement should be expressed too by constraints, and all the three types of constraints become more intrinsic to formulate. The total number of possible arrangements becomes as large as 64^{64} , containing a configuration of queens multiple times due to the identification of the 8 queens. So we have many reasons to prefer the first representation over the second one. It is true in general that a problem can be formulated as a CSP in a number of ways. The resulting CSPs may differ significantly considering the number and complexity of the constraints and the number of the possible instantiations of the variables, and thus may require very different amount of time and memory to be dealt with. Hence when modelling a problem as a CSP, one has to pay attention to different possibilities, and try to commit to the one which will be the easiest to cope with. The in-depth analysis of the different solution methods and of the characteristics of the CSPs may provide a basis to make a good choice. Several cases have been reported when a notoriously difficult problem could be solved finally as a result of change of the representation. Both representations of the 8-queens problem are pleasantly regular: the domain of all the variables is the same, all the constraints refer to 2 variables, and for each pair of variables the same type of constraints are prescribed. Hence the 8-queens problem is not appropriate as a test case for solution algorithms developed to solve general CSPs. In spite of this intuitive observation, earlier the 8-queens had been a favourite test problem: there had been a race to develop search algorithms which were able to solve the problem for bigger and bigger n. (It is possible to construct a solution analytically.) This practice was stopped by two discoveries. On the one hand, Sosic [183] came up with a polynomial-time search algorithm, which was heavily exploiting the above mentioned special characteristics of the problem. On the other hand, by analysing the search space of the n-queens problem, it was shown that the general belief that "the bigger the n the more difficult the problem is" does not hold - in fact, the truth is just the opposite [153].

Example 1.3.2 (graph colouring). Another, equally popular test problem is graph colouring: colour the vertices of a given graph using k colours in such a way that connected vertices get different colours. It is obvious how to turn this problem into a CSP: there are as many variables as vertices, and the domain for each variable is $\{1, 2, \ldots, k\}$, where k is the allowed number of colours to be used. If there is an edge between the vertices represented by the variables x_i and x_j, then there is a constraint referring to these two variables, namely: $x_i \neq x_j$. Though for the first sight graph colouring may seem to be just as a toy problem as the n-queens, there are basic differences between the two problems. First of all, graph colouring is known to be NP-complete, so one does not expect a polynomial-time search algorithm to be found. Secondly, it is easy to generate a great number of test graphs with certain parameters, which are more or less difficult to be coloured, so the family of graph colouring problems is appropriate to test algorithms thoroughly. Finally, many practical problems, like ones from the field of scheduling and planning, can be expressed as an appropriate graph colouring problem.

Let's now provide a formal definition of constraint satisfaction problem:

Definition 1.3.1 (constraint satisfaction problem). *A Constraint Satisfaction Problem is a tuple $\langle V, D, C, con, def, a\rangle$ where*

- *V is a finite set of variables, i.e., $V = \{v_1, \ldots, v_n\}$;*
- *D is a set of values, called the domain;*
- *C is a finite set of constraints, i.e., $C = \{c_1, \ldots, c_m\}$. C is ranked, i.e. $C = \bigcup_k C_k$, such that $c \in C_k$ if c involves k variables;*
- *con is called the connection function and it is such that*

$$con : \bigcup_k (C_k \to V^k),$$

  where $con(c) = \langle v_1, \ldots, v_k\rangle$ is the tuple of variables involved in $c \in C_k$;
- *def is called the definition function and it is such that*

$$def : \bigcup_k (C_k \to \wp(D^k)),$$

  where $\wp(D^k)$ is the powerset of D^k, that is, all the possible subsets of k-tuple in D^k;

– $a \subseteq V$, *and represent the* distinguished *variables of the problem.*

In words, function *con* describes which variables are involved in which constraint, while function def specifies which are the domain tuples permitted by the constraint. The set a is used to point out the variables of interest in the given CSP, i.e., the variables for which we want to know the possible assignments, compatibly with all the constraints.

Note that other classical definitions of constraint problems do not have the notion of distinguished variables, and thus it is as if all variables are of interest. We choose this definition, however, for two reasons: first, the classical definition can be trivially simulated by having a set of distinguished variables containing all the variables, and second, we think that this definition is much more realistic.

In fact, it is reasonable to think that the CSP representation of a problem contains many details (in terms of constraints and/or variables) which are needed for a correct specification of the problem but are not important as far as the solution of the problem is concerned. In other words, the non distinguished variables play the role of existential variables: we want to assure that they can be assigned to values consistently with the constraints, but we do not care to know the assignment.

The solution $Sol(P)$ of a CSP $P = \langle V, D, C, con, def, a\rangle$ is defined as the set of all instantiations of the variables in a which can be extended to instantiations of all the variables which are consistent with all the constraints in C.

Definition 1.3.2 (tuple projection and CSP solution). *Given a tuple of domain values $\langle v_1, \ldots, v_n\rangle$, consider a tuple of variables $\langle x_{i1}, \ldots, x_{im}\rangle$ such that $\forall j = 1, \ldots, m$, there exists a k_j with $k_j \in \{1, \ldots, n\}$ such that $x_{ij} = x_{k_j}$. Then the projection of $\langle v_1, \ldots, v_n\rangle$ over $\langle x_{i1}, \ldots, x_{im}\rangle$, written $\langle v_1, \ldots, v_n\rangle_{|\langle x_{i1}, \ldots, x_{im}\rangle}$, is the tuple of values $\langle v_{i1}, \ldots, v_{im}\rangle$. The solution $Sol(P)$ of a CSP $P = \langle V, D, C, con, def, a\rangle$ is defined as*

$$\{\langle v_1, \ldots, v_n\rangle_{|a} \textit{ such that } \begin{cases} v_i \in D \textit{ for all } i; \\ \forall c \in C, \langle v_1, \ldots, v_n\rangle_{|con(c)} \in def(c). \end{cases} \}$$

The solution to a CSP is hence an assignment of a value from its domain to every variable, in such a way that every constraint is satisfied. We may want to find just one solution, with no preference as to which one, or all solutions.

To give a graphical representation of a CSP problem, we use a labelled hypergraph which is usually called a *constraint graph* ([85]).

Definition 1.3.3 (labelled hypergraph). *Given a set of labels L, a hypergraph labelled over L is a tuple $\langle N, H, c, l\rangle$, where N is a set of nodes, H is a set of hyperarcs, $c : \bigcup_k (H_k \rightarrow N^k)$, and $l : \bigcup_k (H_k \rightarrow \wp(L^k))$. I.e., c gives the tuple of nodes connected by each hyperarc, and l gives the label of each hyperarc.*

Definition 1.3.4 (from CSPs to labelled hypergraphs). *Consider a CSP $P = \langle V, D, C, con, def, a\rangle$. Then the labelled hypergraph corresponding to P, written $G(P)$, is defined as the hypergraph $G(P) = \langle V, C, con, def\rangle$ labelled over D.*

In the hypergraph corresponding to a CSP, the nodes represent the variables of the problem, and the hyperarcs represent the constraints. In particular, each constraint c can be represented as a hyperarc connecting the nodes representing the variables in $con(c)$. Constraint definitions are instead represented as labels of hyperarcs. More precisely, the label of the hyperarc representing constraint c will be $def(c)$.

The representation of a CSP by a hypergraph turns out to be very useful when focusing on properties of the CSP which involve notions of sparseness and/or locality. In particular, locality can be defined in term of subgraphs, where a subgraph of a given graph consists of a subset S of the nodes together with some of the hyperarcs connecting subsets of S. For example, the theory of *local consistency* techniques for discrete CSPs (which we will review later), whose aim is to remove local inconsistencies, can be given in terms of operations on hypergraphs.

As for the notion of solution of a finite domain CSP, consider for example the CSP depicted in Figure 1.2, where each arc is labeled by the definition of the corresponding constraint, given in terms of tuples of values of the domain, and the distinguished variables are marked with a *. The solution of this problem is the set $\{\langle a, b\rangle\}$.

Solution Techniques. The methods to generate a solution for a CSP fall mainly into two classes [168]. In the first class are the variants of backtracking search. These algorithms construct a solution by extending a partial instantiation step by step, relying on different heuristics and using more or less intelligent backtracking strategies to recover from dead ends. The reduction of a problem is advantageous, resulting in a smaller solution space to be searched. The second class are the so-called constraint propagation algorithms do eliminate some non-solution elements from the search space. In general, these algorithms do not eliminate all the non-solution elements, hence, they do not produce a solution on their own. They are used either to pre-process the problem before another type of algorithm is applied, or interwoven with steps of another kind of - e.g. backtracking search - algorithm to boost its performance.

All the algorithms from the above three classes investigate the solution space systematically. Hence all those algorithms of the above classes which are meant to find a solution, in theory really do the job as long as there is a solution. These algorithms are:

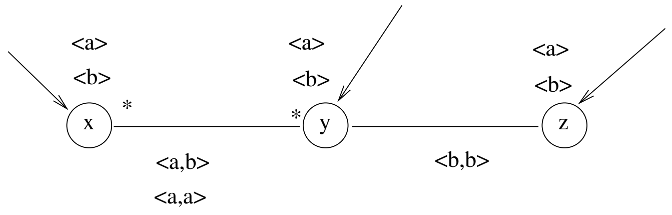


Fig. 1.2. A CSP which is not solved

– *sound*, that is if they terminate with a complete instantiation of the variables than it is a solution;
– *complete*, that is capable to investigate the entire search space and hence find all the solutions.

These requirements seem to be very essential, however, often one has to be satisfied with algorithms which do not fulfill one or both of them. The systematic search algorithms require exponential time for the most difficult CSPs. A CSP is difficult if (almost) the entire search space has to be investigated before finding a solution or concluding that the problem has none. If the search space is large, then it may take days or weeks to run a complete and sound algorithm. This can be forbidding in case of applications where a solution can be used only if provided within a short time.

In such cases a compromise is made, by using an algorithm which provides an answer fast, but the answer is not guaranteed to be a solution. However, it is "good enough" in the sense that not all the constraints are satisfied, but the number of non-satisfied constraints the and degree of violations can be accepted. Though such an algorithms cannot be used to generate all the (good) solutions for sure, usually it is possible to generate several quite different "almost solutions" (if they exist). The so-called *local stochastic search algorithms* have in common that they explore the solution space in a non-systematic way, stepping from one complete instantiation to another, based on random choices, and may navigate on the basis of heuristics, often adopted from systematic search methods. In the recent years such algorithms have been used with success to solve large practical problems, and they are suitable to handle CSPs extended with some objective function to be optimised.

Constraint Propagation. By eliminating redundant values from the problem definition, the size of the solution space decreases. Reduction of the problem can be done once, as pre-processing step for another algorithm, or step by step, interwoven with the exploration of the solution space by a search algorithm. In the latter case, subsets of the solution space are cut off, saving the search algorithm the effort of systematically investigating the eliminated elements, which otherwise would happen, even repeatedly. If as a result of reduction any domain becomes empty, then it is known immediately that the problem has no solution. One should be careful with not spending more effort on reduction than what will "pay off" in the boosted performance of the search algorithm to be used to find a solution of the reduced problem. The reduction algorithms eliminate values by propagating constraints. The amount of constraint propagation is characterised by the consistency level of the problem, hence these algorithms are also called *consistency-algorithms.* The iterative process of achieving a level of consistency is sometimes referred to as the relaxation process, which should not be mixed up with relaxation of constraints. Constraint propagation has a long tradition in CSP research. Below we introduce the most well-known and widely used algorithms.

Node- and arc-consistency. A CSP is *node-consistent*, if all the unary constraints hold for all the elements of the domains. The straightforward node-consistency algorithm (NC), which removes the redundant elements by checking the domains one after the other has O(dn) time complexity, where d is the maxim of the size of the domains and n is the number of the variables. A CSP is arc-consistent, if for any u value from the domain of any variable x, any binary constraints which refers to x can be satisfied.

k-consistency. Arc-consistency can be also understood as telling something about how far a partial solution can always be extended. Namely, any partial solution containing only one instantiated variable can be extended by instantiating any second variable to a properly chosen value. Applying the same principle for more variables, we arrive at the concept of k-consistency.

Definition 1.3.5 (k-consistency). *A CSP is k-consistent, if any consistent instantiation of any $k-1$ variables can be extended by instantiating any one of the remaining variables.*

It is important to understand clearly the significance of consistency. A CSP which is k-consistent, is not necessarily solvable, and in the other way around, the solvability of a problem does not imply any level of consistency, not alone 1-consistency. The consistency as a feature of a problem does guarantee that certain values and value h-tuples which are not in the projection of the solution set have been removed form the domains and constraints. The level of consistency, k indicates for what h-values has this been done.

In general, the main idea is to improve the brute force backtracking by cutting down the search space. This is done by first modifying the original constraint network to a more *explicit* one (by eliminating redundancy), and then to run the search.

In a finite domain CSP, redundancy can be found mainly in the definitions of the constraints. In other words, a tuple of domain values in a constraint definition may be considered to be redundant if its elimination does not change the solution of the problem.

A sufficient condition for tuple redundancy is that the tuple be inconsistent with (all the tuples in) some other constraint in the CSP. In fact, if this condition holds, then the tuple will never be used in any solution of the CSP (otherwise the other constraint would be violated), and thus the solution will not change if the tuple is deleted. Consider, for example, a CSP which, among other variables and constraints, contains the variables x, y, and z, and the two constraints as depicted in Figure 1.3.

Then, tuple $\langle a, a\rangle$ of the constraint between x and y is inconsistent with the constraint between y and z. In fact, if x and y are both instantiated to a, then there is no way to instantiate z so as to satisfy the constraint between y and z. Therefore, tuple $\langle a, a\rangle$ is redundant and may be eliminated from the CSP, yielding a new CSP which is obviously equivalent to the old one. More precisely, given a CSP P_1 , the aim is to obtain a new CSP P_2 such that P_1 and P_2 have the same solution and the same constraint graph, and P_2 has smaller constraint definitions.

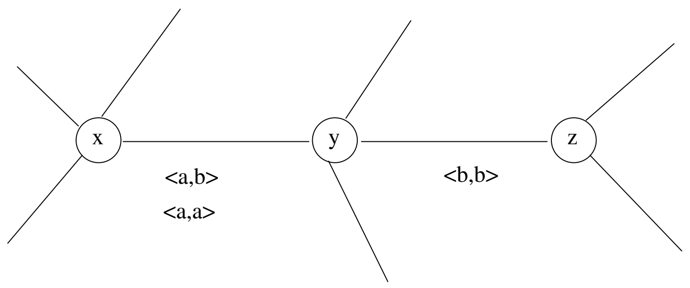


Fig. 1.3. Tuple redundancy

Given that in order to check the redundant status of a tuple we look at some constraints which are adjacent to the one containing that tuple (because constraints not sharing variables with the given one would not help), usually this kind of redundancy is also called *local inconsistency.* In fact, we do not look at the whole CSP, but only at some small portion of it (which contains the constraint with the tuple under consideration). Local inconsistency obviously implies global inconsistency, and this is why the condition of local inconsistency is sufficient for redundancy.

1.3.2 Constraint Logic Programming

As far as we know, even though some languages for handling special kinds of constraints were developed in the past (like ThingLab [60], Bertrand [134], and ALICE ([133]), no general linguistic support were given to constraint solving and constraint propagation until the development of the constraint logic programming (CLP) scheme. This scheme can be seen as a logic programming shell, supported by an underlying constraint system which may handle and solve any kind of constraints. For this reason the CLP scheme can be denoted by CLP(X), where X specifies the class of constraints we want to deal with.

In this section we describe the basic concepts and the operational semantics of the constraint logic programming framework. For a longer and more detailed treatment, which also includes other kinds of semantics, we refer to [125,126,127]. We assume the readers to be already familiar with the logic programming formalism, as defined in [135]. The main idea in CLP is to extend logic programming, in such a way that

- substitutions are replaced by constraints,
- unification is replaced by a constraint solution, and
- all the semantic properties of logic programming (mainly, the existence of declarative, denotational, and procedural semantics and the equivalence of these three objects) still hold.

Figure 1.4 describes informally the CLP framework. As we can see, the logic programming shell and the underlying constraint system communicate by means of consistency checks, which are requested by the shell and performed by the constraint system.

In constraint logic programs the basic components of a problem are stated as *constraints*. The problem as a whole is then represented by putting the various constraints together by means of *rules*.

Definition 1.3.6 (constraint rule, goal, programs). *Consider a particular instance of the CLP(X) framework.*

- *A constraint rule is of the form*

  $$A : -c, B_1, \ldots, B_n.$$

  where $n \geq 0$*,* c *is a constraint over the class* X*, and* A *and all* B_i *s are atoms over the Herbrand Universe.* A *is called the head of the rule, and* $c, B_1, \ldots, B_n$ *the body. If* $n = 0$ *then such a constraint rule can also be called a fact.*
- *A goal is a constraint rule without the head, i.e of the form* $: : -c.$ *or* $: -c, B_1, \ldots, B_n.$
- *A constraint logic program is a finite set of constraint rules.*

Let us now describe how the CLP system works to decide if a given goal is a logical consequence of the program; i.e., how the operational semantics of logic programming, which we assume is given by SLD-resolution, has to be extended in order to deal with constraints instead of (or, better, besides) substitutions.

Definition 1.3.7 (derivation step). *Given a CLP(X) program P, a derivation step takes a goal*

$$: -(c, A_1, \ldots, A_n)$$

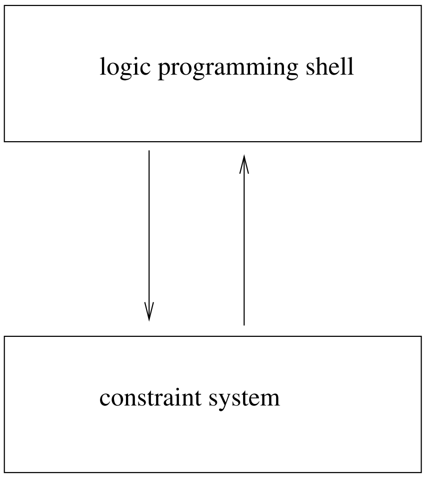


Fig. 1.4. The CLP framework

and a (variant of a) constraint rule

$$A : -c', B_1, \dots, B_m.$$

in P. If $(c \cup c') \cup (A_i = A)$ *is satisfiable, then the derivation step returns a new goal*

$$: -((c \cup c') \cup (A = A_i), A_1, \dots, A_{i-1}, B_1, \dots B_m, A_{i+1}, \dots A_n)$$

Definition 1.3.8 (derivation, answer constraint). *A derivation is a sequence of derivation steps. Derivation sequences are* successful *(and are called refutations) when the last goal therein contains only constraints. These* answer constraints *constitute the output of a CLP program.*

The satisfiability of $A_i = A$ is the problem of finding, if there is one, an mgu, and thus, it is a task which can be accomplished by the logic programming engine that underlies the CLP system. On the contrary, to check the satisfiability of $(c \cup c')$ we need to use the particular constraint solver corresponding to the chosen structure X. If, for example, we are considering arithmetic inequality constraints over the reals, then the constraint solver could be a simplex algorithm. Thus, at each derivation step two constraint solvers are needed: one for the Herbrand universe and one for the chosen class of other constraints.

The fact that CLP can be interpreted as the possibility of using logic programming as a logic "shell" where to formally reason about constraints and solve constraint problems intuitively implies that logic programming is not only an instance of CLP, but it is always "present" in the CLP framework. More precisely, since a constraint is simply a relation which must hold among a collection of objects, then term equalities, which are handled in logic programming while performing unification, are a special case of constraints. Therefore, it is safe to say that any logic program is a CLP program where the only allowed constraints are term equalities. On the other hand, it is also true that any CLP instance needs to perform unifications, and thus term equalities must always be among the allowed constraints.

1.4 Non-crisp Constraints

All this constraint application, however, have some obvious limitations, mainly due to the fact that they do not appear to be very flexible when trying to represent real-life scenarios where the knowledge is neither completely available nor crisp. In fact, in such situations, the ability to state whether an instantiation of values to variables is allowed or not, is not sufficient or sometimes not even possible. For these reasons, it is natural to try to extend the CSP formalism in this direction.

For example [91,165,167], CSPs have been extended with the ability to associate with each tuple, or to each constraint, a level of preference, and with the possibility of combining constraints using min-max operations. This extended formalism was called Fuzzy CSPs (FCSPs). Other extensions concern the ability to model incomplete knowledge of the real problem [96], to solve over-constrained problems [108], and to represent cost optimization problems [24, 31].

The constraints in classical constraint satisfaction problems are crisp, i.e., they either allow a tuple (of values of involved variables) or not. In some real-life applications, this is not a desired feature and, therefore, some alternative approaches to constraint satisfaction were proposed to enable non-crisp constraints, probabilities or weights respectively. These extensions can be used to solve optimization problems as well as over-constrained problems as they allow relaxing of constraints.

1.4.1 Partial Constraint Satisfaction Problems

The Partial Constraint Satisfaction (PCSP) [108] scheme of Freuder and Wallace is an interesting extension of CSP, which allows the relaxation and optimization of problems via the weakening of the original CSP.

A theory of Partial Constraint Satisfaction Problems (PCSPs) were developed to weaken systems of constraints which have no solutions (over-constrained systems), or for which finding a solution would take too long. Instead of searching for a solution to a complex problem, we consider searching for a simpler problem that we know we can solve.

Definition 1.4.1 (Partial Constraint Satisfaction Problems). *The notion of a partial CSP is formalized by considering three components:* $\langle (P, U), (PS, \leq), (M, (N, S)) \rangle$ *where:*

- $(PS, \leq)$ *is a problem space with* PS *a set of problems and* $\leq$ *a partial order over problems,*
- $P \in PS$ *is a constraint satisfaction problem (CSP),* U *is a set of 'universes' i.e. a set of potential values for each of the variables in* P*,*
- M *is a distance function over the problem space, and* (N, S) *are* N*ecessary and* S*ufficient bounds on the distance between the given problem* P *and some solvable member of the problem space* PS

A solution to a $PCSP$ *is a problem* P' *from the problem space and its solution, where the distance between* P *and* P' *is less than* N*. If the distance between* P *and* P' *is minimal, then this solution is optimal.*

1.4.2 Constraint Hierarchies

Constraint hierarchies [61] were introduced for describing over-constrained systems of constraints by specifying constraints with hierarchical strengths or preferences. It allows one to specify declaratively not only the constraints that are required to hold, but also weaker, so called soft constraints at an arbitrary but finite number of strengths. Weakening the strength of constraints helps in finding a solution of a previously over-constrained system of constraints. Intuitively, the hierarchy does not permit the weakest constraints to influence the result.

Moreover, constraint hierarchies allow "relaxing" of constraints with the same strength by applying, for example a weighted-sum, least-squares or similar comparators.

Definition 1.4.2 (Constraint Hierarchies). *In constraint hierarchies, one uses* labeled constraints *which are constraints labeled with a strength or preference. Then, a* constraint hierarchy *is a (finite) multiset of labeled constraints. Given a constraint hierarchy H, let H_0 denote the set of required constraints in H, with their labels removed. In the same way, we define the sets $H_1, H_2, \ldots$ for levels$1, 2, \ldots$.*

A solution to a constraint hierarchy H will consist of instantiations for variables in H, that satisfies the constraints in H respecting the hierarchy. Formally (U, V are instantiations):

$$S_{H,0} = \{U \mid \textit{for each } c \textit{ in } H_0,\ c \textit{ is satisfied by } U\}$$
$$S_H = \{U \mid U \in S_{H,0} \textit{ and } \forall V \in S_{H,0},\ V \textit{ is not better than } U \textit{ respecting } H\}.$$

The set $S_{H,0}$ contains instantiations satisfying required constraints in H and S_H, called a solution set *for H, is a subset of $S_{H,0}$ containing all instantiations which are not worse than other instantiations in $S_{H,0}$.*

1.4.3 Fuzzy, Probabilistic and Valued CSPs

The Fuzzy [91, 165, 167], Probabilistic [96] and Valued CSPs [174] use values to express fuzzyness, probability or several preference values in the problem. In particular, Fuzzy CSPs are a very significant extension of CSPs since they are able to model partial constraint satisfaction (PCSP [108]) and also prioritised constraints (Constraint hierarchies [61]), that is, constraints with different levels of importance .

We will give a detailed description of the fuzzy, probabilistic and VCSP framework in the next chapter, when will also show how to represent all this non-crisp extension in our framework.

1.5 Overview of the Book

In this book we describe a constraint solving framework where most of these extensions, as well as classical CSPs, can be cast. The main idea is based on the observation that a semiring (that is, a domain plus two operations satisfying certain properties) is all that is needed to describe many constraint satisfaction schemes. In fact, the domain of the semiring provides the levels of consistency (which can be interpreted as either cost, degrees of preference, probabilities, or others), and the two operations define a way to combine constraints together. More precisely, we define the notion of constraint solving over any semiring. Specific choices of the semiring will then give rise to different instances of the framework, which may correspond to known or new constraint solving schemes.

With the definition of the new framework, we also need to extend the classical AI local consistency techniques [104, 105, 138, 139, 150, 152] that have proven to be very effective when approximating the solution of a problem in order to be applied in the semiring framework. We may say that all of them are variants or extensions of two basic algorithms, which started the field. One is the *arc consistency* algorithm (used for the first time in [193] but studied and named later in [138], where we may say that inconsistency is eliminated by each unary constraint (i.e., involving only one variable); or, that each unary constraint (i.e., connecting only one variable, say v) is propagated to all variables attached to the variable v by some binary constraint. The other pioneering algorithm is the *path-consistency* algorithm (which was proposed in [150]), where inconsistency is eliminated by each binary constraint; or, in other words, where each binary constraint connected to variables v and v' is propagated to all other variables attached to v and v' by some binary constraint. Then, a more general one is the *k-consistency* algorithm scheme [104], where inconsistency is eliminated by each set of constraints involving k variables (it is easy to see that arc-consistency is 2-consistency and that path-consistency is 3-consistency).

We generalize these techniques to our framework, and we provide sufficient conditions over the semiring operations which assure that they can also be fruitfully applied to the considered scheme. By "fruitfully applicable" we mean that 1) the algorithm terminates and 2) the resulting problem is equivalent to the given one and it does not depend on the nondeterministic choices made during the algorithm. The advantage of the SCSP framework is that one can hopefully see one's own constraint solving paradigm as an instance of SCSP over a certain semiring, and can inherit the results obtained for the general scheme. In particular, one can immediately see whether a local consistency and/or a dynamic programming technique can be applied.

Also the study of generalized versions of AI algorithms for the SCSP framework seems to be very important. In particular, the *partial local consistency* algorithms seem to be efficient and useful.

Moreover, we study the behavior of a class of algorithms able to eliminate from a semiring-based constraint problem some solutions that are not *interesting* (where *interesting* might mean optimal) and we will study their application to SCSPs. To solve an SCSP problem in a faster way, we try to "simplify" it by changing the structure of the semiring. The solutions obtained in the easier semiring have to be mapped back and used to solve the original problem. In order to capture some interesting properties we use a notion of *abstract interpretation* to represent this mapping. The idea is to take an SCSP problem P and map it over a new problem P' which is easier to handle and solve. As soon as we have solved (or partially solved) P' we have to map back the solutions (or the partial solutions) over the original problem. Depending on the properties of the semiring operations, we can perform some simplification steps over the original problem.

Next we demonstrate how to extend the logic programming paradigm to deal with semiring based constraints. One of the ways to program with constraint is to dip the constraints in the logic programming theory obtaining the CLP framework. Programming in CLP means choosing a constraint system for a specific class of constraints (for example, linear arithmetic constraints, or finite domain constraints) and embedding it into a logic programming engine. This approach is very flexible because one can choose among many constraint systems without changing the overall programming language, and it has been shown to be very successful in specifying and solving complex problems in terms of constraints of various kinds. It can, however, only handle classical constraint solving. Thus, it is natural to try to extend the CLP formalism in order to be able to also handle SCSP problems. We will call such an extension SCLP (for Semiring-based CLP).

In passing from CLP to SCLP languages, we replace classical constraints with the more general SCSP constraints. By doing this, we also have to modify the notions of interpretation, model, model intersection, and others, since we have to take into account the semiring operations and not the usual CLP operations. For example, while CLP interpretations associate a truth value (either *true* or *false*) to each ground atom, here, ground atoms must be given one of the elements of the semiring. Also, while in CLP the value associated to an existentially quantified atom is the *logical or* among the truth values associated to each of its instantiations, here we have to replace the *or* with another operation which refers to one of the semiring ones. The same reasoning applies to the value of a conjunction of atomic formulas: instead of using the logical *and* we have to use a specific operation of the semiring.

After describing the syntax of SCLP programs, we define three equivalent semantics for such languages: model-theoretic, fix-point, and operational. These semantics are conservative extensions of the corresponding ones for Logic Programming (LP), since by choosing a particular semiring (the one with just two elements, *true* and *false*, and the logical *and* and *or* as the two semiring operations) we get exactly the LP semantics. The extension is in some cases predictable but it possesses some crucial new features. We also show the equivalence of the three semantics. In particular, we prove that, given the set of all refutations starting from a given goal, it is possible to derive the declarative meaning of both the existential closure of the goal and its universal closure. Moreover, we investigate the decidability of the semantics of SCLP programs and we prove that if a goal has a semiring value greater than a certain value in the semiring, then we can discover this in finite time.

Classical Operational Research (OR) problems and their embedding in the SCLP framework are also studied. The embedding seems to be able to 1) give a new semantics w.r.t. the algorithmic one and 2) give some hints towards an extension of the SCLP framework to handle multilevel and multisorted problems. More precisely, the "levels of preference" together with a constraint framework can be used fruitfully to describe and solve optimization problems. In fact, solving a problem does not mean finding "a solution", but finding the best one (in the sense of the semiring). This intuition suggests we investigate around the operational research field in order to find out how to describe and solve classical problems that usually are solved using OR techniques.

In the last part of the book we describe some new results related to the semiring framework. An extension of the concurrent constraint language (cc) is presented. The language is able to deal with soft constraints instead of with the crisp ones. The idea of thresholds is added to the new language, and a new "branch&bound" semantics is given.

Using the introduced language we show how to use the SCSP framework to discovery security problems in the network. In particular we study what happens while a protocol session is executed. To do this we first model the policy of the protocol and the initial state of the network as SCSPs, and then we compare the policy SCSP with the initial one, modified by the protocol session.

Then a notion of Neighborhood Interchangeability for soft constraints is introduced. As in the crisp case, detecting interchangeabilities among domain values improve the performance of the solver. Interchangeability can also be used as a basis for search heuristics, solution adaptation and abstraction techniques

1.5.1 Structure of Subsequent Chapter

In Chapter 1 we introduce some background material: some basic notions related to Constraint Satisfaction Problems (CSPs) and to the Constraint Logic Programming (CLP) framework.

In Chapter 2 we introduce the Soft CSP (SCSP) framework. The framework is based on the notion of semiring, that is, a set plus two operations. Different instantiations of the two operations give rise to different existing frameworks (Fuzzy CSP, Probabilistic, Classical, VCSP, etc.). Each of these frameworks satisfies specific properties that are useful in testing the applicability of the constraint preprocessing and solving techniques described in Chapter 3.

In Chapter 3 the properties studied in the previous chapter are applied to the different frameworks in order to prove the applicability of soft local consistency and dynamic programming techniques. These techniques are defined starting from the classical ones, by using the semiring operations. We also generalize the local consistency technique by reducing the needed properties of the semiring operators and by using different functions in the local consistency steps. Some ideas on the efficient applicability of these techniques are given, by looking at the graph structure of the problem.

In Chapter 4 classical techniques of abstraction are applied to SCSPs. The problem is mapped in an "easier" one, some solution related information is found and mapped back to the original problem. The soft local consistency techniques are also proven to be useful in the abstract framework to find the set of optimal solutions or one of its approximations.

In Chapter 5 a more structured view of the SCSP framework is given: domains and semirings are used as basic elements for building high order functions representing solution or preprocessing algorithms.

In chapter 6 the CLP framework is extended to deal with soft constraints. We define the Soft CLP (SCLP) framework and we give an operational, model-theoretic and fix-point semantics. Some equivalence and properties of the semantics are proven.

In Chapter 7 we show how the ability of the CLP language to deal with soft constraints can be used to represent and solve in a declarative fashion OR problems. As an example we consider general shortest path problems over partially ordered criteria and we show how the SCLP framework can easily represent and solve them. Moreover, we give a semantics to a class of SCLP programs by using a classical shortest-path algorithm.

In Chapter 8 the extension of the concurrent constraint language (cc) able to deal with soft constraints is presented. We first give a functional formalization of the semiring framework that will also be used in the next two chapter. Then syntax and semantics of the new language is given.

In Chapter 9 we describe the notion of soft Neighborhood Interchangeability. As in the crisp case interchangeabilities are very rare. For this reason we extend the interchangeability definition with a notion of threshold and degradation.

In Chapter 10 we try to enlarge the applicability domain of the constraint framework, by defining a mapping between a network of agents executing a protocol and an SCSP. Some security properties of the network are translated into specific properties of the SCSP solutions.

In Chapter 11 we give some final remarks and some directions for future research.

1.5.2 The Origin of the Chapters

Many chapters of this book are based on already published papers, jointly with Giampaolo Bella, Philippe Codognet, Boi Faltings, Helene Fargier, Rossella Gennari, Yan Georget, Ugo Montanari, Nicoleta Neagu, Elvinia Riccobene, Francesca Rossi, Thomas Schiex and Gerard Verfaillie. In particular:

– The basic definitions and properties of the soft constraints' framework described in Chapter 2 appear in [45,47]; moreover the mapping between SCSPs and VCSPs is based on the ideas developed in [38,39];
– The solutions and preprocessing techniques described in Chapter 3 are instead introduced in [45,47] and extended in [34,43,44,54];
– Chapter 4 describe the abstraction framework and develops the ideas presented in [33,35,36];
– Chapter 5 is only based on some preliminary work presented in [49];
– The logic language to program with soft constraints of Chapter 6 is introduced in [46,53]; an extended version of the work appeared in [50];
– Chapter 7 is based on the ideas developed in [48,51];
– The soft concurrent constraint framework described in Chapter 8 appeared in [52];
– Chapter 9 is based on the ideas developed in [37,55].
– The use of soft constraints for security protocol quantitative analysis introduced in Chapter 10 is based on the works presented in [17,18,19].

2. Soft Constraint Satisfaction Problems

Three Rings for the Elven-kings under the sky,
Seven for the Dwarf-lords in their halls of stone,
Nine for the Mortal Men doomed to die,
One for the Dark Lord on his dark throne.
In the Land of Mordor where the Shadows lie.
One Ring to rule them all, One Ring to find them,
One Ring to bring them all and in the darkness bind them.
In the Land of Mordor where the Shadows lie.
Tolkien, J.R.R. The Lord of the Rings

Overview

We introduce a general framework for constraint satisfaction and optimization where classical CSPs, fuzzy CSPs, weighted CSPs, partial constraint satisfaction, and others can be easily cast. The framework is based on a semiring structure, where the carrier set of the semiring specifies the values to be associated with each tuple of values of the variable domain, and the two semiring operations ($+$ and $\times$) model constraint projection and combination respectively.

Classical constraint satisfaction problems (CSPs) [139, 150] are a very expressive and natural formalism to specify many kinds of real-life problems. In fact, problems ranging from map coloring, vision, robotics, job-shop scheduling, VLSI design, and so on, can easily be cast as CSPs and solved using one of the many techniques that have been developed for such problems or subclasses of them [104, 105, 138, 140, 150].

They also, however, have evident limitations, mainly due to the fact that they do not appear to be very flexible when trying to represent real-life scenarios where the knowledge is neither completely available nor crisp. In fact, in such situations, the ability of stating whether an instantiation of values to variables is allowed or not, is not enough or sometimes not even possible. For these reasons, it is natural to try to extend the CSP formalism in this direction.

For example, in [91,165,167] CSPs were extended with the ability to associate with each tuple, or to each constraint, a level of preference, and with the possibility of combining constraints using min-max operations. This extended formalism were called Fuzzy CSPs (FCSPs). Other extensions concern the ability to model incomplete knowledge of the real problem [96], to solve over-constrained problems [108], and to represent cost optimization problems [31].

In this chapter we describe a constraint solving framework where all such extensions, as well as classical CSPs, can be cast. We do not, however, relax the assumption of a finite domain for the variables of the constraint problems. The main idea is based on the observation that a semiring (that is, a domain plus two operations satisfying certain properties) is all that is needed to describe many constraint satisfaction schemes. In fact, the domain of the semiring provides the levels of consistency (which can be interpreted as cost, or degrees of preference, or probabilities, or others), and the two operations define a way to combine constraints together. More precisely, we define the notion of constraint solving over any semiring. Specific choices of the semiring will then give rise to different instances of the framework, which may correspond to known or new constraint solving schemes.

The notion of semiring for constraint solving was used also in [114]. However, the use of such a notion is completely different from ours. In fact, in [114] the semiring domain (hereby denoted by A) is used to specify the domain of the variables, while here we always assume a finite domain (hereby denoted by D) and A is used to model the values associated with the tuples of values of D.

The chapter is organized as follows. Section 2.1 defines c-semirings and their properties. Then Section 2.2 introduces constraint problems over any semirings and the associated notions of solution and consistency. Then Section 2.3 studies several instances of the SCSP framework, and in particular studies the differences between the framework presented in [174] and ours.

The basic definitions and properties of the soft constraints' framework described in this chapter appear in [45,47]; moreover the mapping between SCSPs and VCSPs [174] is based on the ideas developed in [38,39].

2.1 C-semirings and Their Properties

We extend the classical notion of constraint satisfaction to allow also for 1) non-crisp statements and 2) a more general interpretation of the operations on such statements. This allows us to model both classical CSPs and several extensions of them (like fuzzy CSPs [91,167], partial constraint satisfaction [108], and so on). To formally do that, we associate a semiring with the standard notion of constraint problem, so that different choices of the semiring represent different concrete constraint satisfaction schemes. In fact, such a semiring will give us both the domain for the non-crisp statements and also the allowed operations on them.

Definition 2.1.1 (semiring). *A semiring is a tuple* $\langle A, sum, \times, \mathbf{0}, \mathbf{1}\rangle$ *such that*

- A *is a set and* $\mathbf{0}, \mathbf{1} \in A$*;*
- *sum, called the additive operation, is a commutative (i.e.,* $sum(a,b) = sum(b,a)$*) and associative (i.e.,* $sum(a, sum(b,c)) = sum(sum(a,b),c)$*) operation with* $\mathbf{0}$ *as its unit element (i.e.,* $sum(a,\mathbf{0}) = a = sum(\mathbf{0},a)$*);*
- $\times$*, called the multiplicative operation, is an associative operation such that* $\mathbf{1}$ *is its unit element and* $\mathbf{0}$ *is its absorbing element (i.e.,* $a \times \mathbf{0} = \mathbf{0} = \mathbf{0} \times a$*);*

- $\times$ *distributes over sum (i.e., for any* $a, b, c \in A$, $a \times sum(b, c) = sum((a \times b), (a \times c))$*).*

In the following we will consider semirings with additional properties of the two operations. Such semirings will be called *c-semiring*, where "c" stands for "constraint", meaning that they are the natural structures to be used when handling constraints.

Definition 2.1.2 (c-semiring). *A c-semiring is a tuple* $\langle A, +, \times, \mathbf{0}, \mathbf{1}\rangle$ *such that*

- A *is a set and* $\mathbf{0}, \mathbf{1} \in A$*;*
- $+$ *is defined over (possibly infinite) sets of elements of* A *as follows*[1]*:*
  - *for all* $a \in A$, $\sum(\{a\}) = a$*;*
  - $\sum(\emptyset) = \mathbf{0}$ *and* $\sum(A) = \mathbf{1}$*;*
  - $\sum(\bigcup A_i, i \in I) = \sum(\{\sum(A_i), i \in I\})$ *for all sets of indices* I *(*flattening property*).*
- $\times$ *is a binary, associative and commutative operation such that* $\mathbf{1}$ *is its unit element and* $\mathbf{0}$ *is its absorbing element;*
- $\times$ *distributes over* $+$ *(i.e., for any* $a \in A$ *and* $B \subseteq A$, $a \times \sum(B) = \sum(\{a \times b, b \in B\})$*).*

Operation $+$ is defined over any set of elements of A, also over infinite sets. This will be useful later in proving Theorem 2.1.4. The fact that $+$ is defined over *sets* of elements, and not *pairs* or *tuples*, automatically makes such an operation commutative, associative, and idempotent. Moreover, it is possible to show that $\mathbf{0}$ is the unit element of $+$; in fact, by using the flattening property we get $\sum(\{a, \mathbf{0}\}) = \sum(\{a\} \cup \emptyset) = \sum(\{a\}) = a$. This means that a c-semiring is a semiring (where the *sum* operation is $+$) with some additional properties.

It is also possible to prove that $\mathbf{1}$ is the absorbing element of $+$. In fact, by flattening and by the fact that we set $\sum(A) = \mathbf{1}$, we get $\sum(\{a, \mathbf{1}\}) = \sum(\{a\} \cup A) = \sum(A) = \mathbf{1}$.

Let us now consider the advantages of using c-semirings instead of semirings. First, the idempotency of the $+$ operation is needed in order to define a partial ordering $\leq_S$ over the set A, which will enable us to compare different elements of the semiring. Such partial order is defined as follows: $a \leq_S b$ iff $a + b = b$. Intuitively, $a \leq_S b$ means that b is "better" than a, or, from another point of view, that, between a and b, the $+$ operation chooses b. This ordering will be used later to choose the "best" solution in our constraint problems.

Theorem 2.1.1 ($\leq_S$ is a partial order). *Given any c-semiring* $S = \langle A, +, \times, \mathbf{0}, \mathbf{1}\rangle$*, consider the relation* $\leq_S$ *over* A *such that* $a \leq_S b$ *iff* $a + b = b$*. Then* $\leq_S$ *is a partial order.*

Proof. We have to prove that $\leq_S$ is reflexive, transitive, and antisymmetric:

[1] When $+$ is applied to a two-element set, we will use the symbol $+$ in infix notation, while in general we will use the symbol $\sum$ in prefix notation.

– Since $+$ is idempotent, we have that $a + a = a$ for any $a \in A$. Thus, by definition of $\leq_S$, we have that $a \leq_S a$. Thus $\leq_S$ is reflexive.
– Assume $a \leq_S b$ and $b \leq_S c$. This means that $a + b = b$ and $b + c = c$. Thus $c = b + c = (a + b) + c = a + (b + c) = a + c$. Note that here we also used the associativity of $+$. Thus $\leq_S$ is transitive.
– Assume that $a \leq_S b$ and $b \leq_S a$. This means that $a + b = b$ and $b + a = a$. Thus $a = b + a = a + b = b$. Thus $\leq_S$ is antisymmetric.

The fact that $\mathbf{0}$ is the unit element of the additive operation implies that $\mathbf{0}$ is the minimum element of the ordering. Thus, for any $a \in A$, we have $\mathbf{0} \leq_S a$.

It is important to note that both the additive and the multiplicative operations are monotone on such an ordering.

Theorem 2.1.2 ($+$ and $\times$ are monotone over $\leq_S$). *Given any c-semiring $S = \langle A, +, \times, \mathbf{0}, \mathbf{1} \rangle$, consider the relation $\leq_S$ over A. Then $+$ and $\times$ are monotone over $\leq_S$. That is, $a \leq_S a'$ implies $a + b \leq_S a' + b$ and $a \times b \leq_S a' \times b$.*

Proof. Assume $a \leq_S a'$. Then, by definition of $\leq_S$, $a + a' = a'$.

Thus, for any b, $a' + b = a + a' + b$. By idempotency of $+$, we also have $a' + b = a + a' + b = a + a' + b + b$, which, by commutativity of $+$, becomes $a' + b = (a + b) + (a' + b)$. By definition of $\leq_S$, we have $a + b \leq_S a' + b$.

Also, from $a + a' = a'$ derives that, for any b, $a' \times b = (a' + a) \times b =$(by distributiveness) $(a' \times b) + (a \times b)$. This means that $(a \times b) \leq_S (a' \times b)$.

The commutativity of the $\times$ operation is desirable when such an operation is used to combine several constraints. In fact, were it not commutative, it would mean that different orders of the constraints would give different results.

Since $\mathbf{1}$ is also the absorbing element of the additive operation, then $a \leq_S \mathbf{1}$ for all a. Thus $\mathbf{1}$ is the maximum element of the partial ordering. This implies that the $\times$ operation is *intensive*, that is, that $a \times b \leq_S a$. This is important since it means that combining more constraints leads to a worse (w.r.t. the $\leq_S$ ordering) result.

Theorem 2.1.3 ($\times$ is intensive). *Given any c-semiring $S = \langle A, +, \times, \mathbf{0}, \mathbf{1} \rangle$, consider the relation $\leq_S$ over A. Then $\times$ is intensive, that is, $a, b \in A$ implies $a \times b \leq_S a$.*

Proof. Since $\mathbf{1}$ is the unit element of $\times$, we have $a = a \times \mathbf{1}$. Also, since $\mathbf{1}$ is the absorbing element of $+$, we have $\mathbf{1} = \mathbf{1} + b$. Thus $a = a \times (\mathbf{1} + b)$. Now, $a \times (\mathbf{1} + b)$ = { by distributiveness of $\times$ over $+$ } $(a \times \mathbf{1}) + (a \times b)$ = {$\mathbf{1}$ unit element of $\times$} $a + (a \times b)$. Thus we have $a = a + (a \times b)$, which, by definition of $\leq_S$, means that $(a \times b) \leq_S a$.

In the following we will sometimes need the $\times$ operation to be closed on a certain finite subset of the c-semiring.

Definition 2.1.3 (AD-closed). *Given any c-semiring $S = \langle A, +, \times, \mathbf{0}, \mathbf{1} \rangle$, consider a finite set $AD \subseteq A$. Then $\times$ is AD-closed if, for any $a, b \in AD$, $(a \times b) \in AD$.*

We will now show that c-semirings can be assimilated to complete lattices. Moreover, we will also sometimes need to consider c-semirings where $\times$ is idempotent, which we will show equivalent to distributive lattices. See [80] for a deep treatment of lattices.

Definition 2.1.4 (lub, glb, (complete) lattice [80]). *Consider a partially ordered set S and any subset I of S. Then we define the following:*

- *an upper bound (resp., lower bound) of I is any element x such that, for all $y \in I$, $y \leq x$ (resp., $x \leq y$);*
- *the least upper bound (lub) (resp., greatest lower bound (glb)) of I is an upper bound (resp., lower bound) x of I such that, for any other upper bound (resp., lower bound) x' of I, we have that $x \leq x'$ (resp., $x' \leq x$).*

A lattice is a partially ordered set where every subset of two elements has a lub and a glb. A complete lattice is a partially ordered set where every subset has a lub and a glb.

We will now prove a property of partially ordered sets where every subset has the lub, which will be useful in proving that $\langle A, \leq_S \rangle$ is a complete lattice. Notice that when every subset has the lub, then also the empty set has the lub. Thus, in partial orders with this property, there is always a global minimum of the partial order (which is the lub of the empty set).

Lemma 2.1.1 (lub $\Rightarrow$ glb). *Consider a partial order $\langle A, \leq \rangle$ where there is the lub of every subset I of A. Then there exists the glb of I as well.*

Proof. Consider any set $I \subseteq A$, and let us call $LB(I)$ the set of all lower bounds of S. That is, $LB(I) = \{x \in A \mid \text{for all } y \in I,\ x \leq_S y\}$. Then consider $a = lub(LB(I))$. We will prove that a is the glb of I.

Consider any element $y \in I$. By definition of $LB(I)$, we have that all elements x in $LB(I)$ are smaller than y, thus y is an upper bound of $LB(I)$. Since a is by definition the smallest among the upper bounds of $LB(I)$, we also have that $a \leq_S y$. This is true for all elements y in I. Thus a is a lower bound of I, which means that it must be contained in $LB(I)$. Thus we have found an element of A which is the greatest among all lower bounds of I.

Theorem 2.1.4 ($\langle A, \leq_S \rangle$ is a complete lattice). *Given a c-semiring $S = \langle A, +, \times, \mathbf{0}, \mathbf{1} \rangle$, consider the partial order $\leq_S$. Then $\langle A, \leq_S \rangle$ is a complete lattice.*

Proof. To prove that $\langle A, \leq_S \rangle$ is a complete lattice it is enough to show that every subset of A has the lub. In fact, by Lemma 2.1.1 we would get that each subset of A has both the lub and the glb, which is exactly the definition of a complete lattice.

We already know by Theorem 2.1.1 that $\langle A, \leq_S \rangle$ is a partial order. Now, consider any set $I \subseteq A$, and let us set $m = \sum(I)$ and $n = lub(I)$. Given any element $x \in I$, we have that $x + m =$(by flattening) $\sum(\{x\} \cup I) = \sum(I) = m$. Therefore, by definition of $\leq_S$, we have that $x \leq_S m$. Thus also $n \leq_S m$, since m is an upper bound of I and n by definition is the least among the upper bounds.

On the other hand, we have that
$m + n =$ (by definition of sum)
$\sum(\{m\} \cup \{n\}) =$ (by flattening and since $m = \sum(I)$)
$\sum(I \cup \{n\}) =$ (by giving an alternative definition of the set $I \cup \{n\}$)
$\sum(\bigcup_{x \in I}(\{x\} \cup \{n\})) =$ (by flattening)
$\sum(\{\sum(\{x\} \cup \{n\}), x \in I\}) =$ (since $x \leq_S n$ and thus $\sum(\{x\} \cup \{n\}) = n$)
$\sum(\{n\}) = n$.
Thus we have proven that $m \leq_S n$, which, together with the previous result (that $n \leq_S m$) yields $m = n$. In other words, we proved that $\sum(I) = lub(I)$ for any set $I \subseteq A$. Thus every subset of A, say I, has a least upper bound (which coincides with $\sum(I)$). Thus $\langle A, \leq_S \rangle$ is a lub-complete partial order.

Note that the proof of the previous theorem also says that the $+$ operation coincides with the lub of the lattice $\langle A, \leq_S \rangle$.

Theorem 2.1.5 ($\times$ idempotent). *Given a c-semiring $S = \langle A, +, \times, \mathbf{0}, \mathbf{1} \rangle$, consider the corresponding complete lattice $\langle A, \leq_S \rangle$. If $\times$ is idempotent, then we have that:*

1. *$+$ distributes over $\times$;*
2. *$\times$ coincides with the glb operation of the lattice;*
3. *$\langle A, \leq_S \rangle$ is a distributive lattice.*

Proof.

1. $(a + b) \times (a + c) = \{$since $\times$ distributes over $+\}$
   $((a + b) \times a) + ((a + b) \times c)) = \{$same as above$\}$
   $((a \times a) + (a \times b)) + ((a + b) \times c)) = \{$by idempotency of $\times\}$
   $(a + (a \times b)) + ((a + b) \times c)) = \{$by intensivity of $\times$ and definition of $\leq_S\}$
   $a + ((a + b) \times c)) = \{$since $\times$ distributes over $+\}$
   $a + ((c \times a) + (c \times b)) = \{$by intensivity of $\times$ and definition of $\leq_S\}$
   $a + (c \times b)$.
2. Assume that $a \times b = c$. Then, by intensivity of $\times$ (see Theorem 2.1.3), we have that $c \leq_S a$ and $c \leq_S b$. Thus c is a lower bound for a and b. To show that it is a glb, we need to show that there is no other lower bound c' such that $c \leq_s c'$. Assume that such c' exists. We now prove that it must be $c' = c$:
   $c' = \{$since $c \leq_S c'\}$
   $c' + c = \{$since $c = a \times b\}$
   $c' + (a \times b) = \{$since $+$ distributes over $\times$, see previous point$\}$
   $(c' + a) \times (c' + b) = \{$since we assumed $c' \leq_S a$ and $c' \leq_S b$, and by definition of $\times\}$
   $a \times b = \{$by assumption$\}$
   c.
3. This comes from the fact that $+$ is the lub, $\times$ is the glb, and $\times$ distributes over $+$ by definition of semiring (the distributiveness in the other direction is given by Lemma 6.3 in [80], or can be seen in the first point above).

Note that, in the particular case in which $\times$ is idempotent and $\leq_S$ is total, we have that $a + b = max(a, b)$ and $a \times b = min(a, b)$.

2.2 Constraint Systems and Problems

We will now define the notion of constraint system, constraint, and constraint problem, which will be parametric w.r.t. the notion of c-semiring just defined. Intuitively, a constraint system specifies the c-semiring $\langle A, +, \times, \mathbf{0}, \mathbf{1}\rangle$ to be used, the set of all variables and their domain D.

Definition 2.2.1 (constraint system). *A constraint system is defined as a tuple $CS = \langle S, D, V\rangle$, where S is a c-semiring, D is a finite set, and V is an ordered set of variables.*

Now, a constraint over a given constraint system specifies the involved variables and the "allowed" values for them. More precisely, for each tuple of values (of D) for the involved variables, a corresponding element of A is given. This element can be interpreted as the tuple's weight, or cost, or level of confidence, or other.

Definition 2.2.2 (constraint). *Given a constraint system $CS = \langle S,\ D,\ V\rangle$, where $S = \langle A,\ +,\ \times,\ \mathbf{0},\ \mathbf{1}\rangle$, a constraint over CS is a pair $\langle def, con\rangle$, where*

- *$con \subseteq V$, it is called the* type *of the constraint;*
- *$def : D^k \rightarrow A$ (where k is the cardinality of con) is called the* value *of the constraint.*

A constraint problem is then just a set of constraints over a given constraint system, plus a selected set of variables (thus a *type*). These are the variables of interest in the problem, i.e., the variables of which we want to know the possible assignments' compatibly with all the constraints.

Definition 2.2.3 (constraint problem). *Consider any constraint system $CS = \langle S, D, V\rangle$. A constraint problem P over CS is a pair $P = \langle C, con\rangle$, where C is a set of constraints over CS and $con \subseteq V$. We also assume that $\langle def_1, con'\rangle \in C$ and $\langle def_2, con'\rangle \in C$ implies $def_1 = def_2$. In the following we will write SCSP to refer to such constraint problems.*

Note that the above condition (that there are no two constraints with the same type) is not restrictive. In fact, if in a problem we had two constraints $\langle def_1, con'\rangle$ and $\langle def_2, con'\rangle$, we could replace both of them with a single constraint $\langle def, con'\rangle$ with $def(t) = def_1(t) \times def_2(t)$ for any tuple t. Similarly, if C were a multiset, e.g. if a constraint $\langle def, con'\rangle$ had n occurrences in C, we could replace all its occurrences with the single constraint $\langle def', con'\rangle$ with

$$def'(t) = \underbrace{def(t) \times \ldots \times def(t)}_{n \text{ times}}$$

for any tuple t. This assumption, however, implies that the operation of union of constraint problems is not just set union, since it has to take into account the possible presence of constraints with the same type in the problems to be combined (at most one in each problem), and, in that case, it has to perform the just-described constraint replacement operations.

When all variables are of interest, like in many approaches to classical CSP, *con* contains all the variables involved in any of the constraints of the given problem, say P. Such a set, called $V(P)$, is a subset of V that can be recovered by looking at the variables involved in each constraint. That is, $V(P) = \bigcup_{\langle def, con' \rangle \in C} con'$.

As for classical constraint solving, SCSPs as defined above also can be graphically represented via labeled hypergraphs where nodes are variables, hyperarcs are constraints, and each hyperarc label is the definition of the corresponding constraint (which can be seen as a set of pairs ⟨ tuple, value ⟩). The variables of interest can then be marked in the graph.

Note that the above definition is parametric w.r.t. the constraint system CS and thus w.r.t. the semiring S. In the following we will present several instantiations of such a framework, and we will show them to coincide with known and also new constraint satisfaction systems.

In the SCSP framework, the values specified for the tuples of each constraint are used to compute corresponding values for the tuples of values of the variables in *con*, according to the semiring operations: the multiplicative operation is used to combine the values of the tuples of each constraint to get the value of a tuple for all the variables, and the additive operation is used to obtain the value of the tuples of the variables in the type of the problem. More precisely, we can define the operations of *combination* ($\otimes$) and *projection* ($\Downarrow$) over constraints. Analogous operations were originally defined for fuzzy relations in [196], and used for fuzzy CSPs in [91]. Our definition is, however, more general since we do not consider a specific c-semiring (like that which we will define for fuzzy CSPs later) but a general one.

To define the operators of combination ($\otimes$) and projection ($\Downarrow$) over constraints we need the definition of tuple projection (which is given here in a more general way w.r.t. that given in Definition 1.3.2 where it was used only for the definition of CSP solutions).

Definition 2.2.4 (tuple projection). *Given a constraint system $CS = \langle S, D, V \rangle$ where V is totally ordered via ordering $\prec$, consider any k-tuple*[2] *$t = \langle t_1, \ldots, t_k \rangle$ of values of D and two sets $W = \{w_1, \ldots, w_k\}$ and $W' = \{w'_1, \ldots, w'_m\}$ such that $W' \subseteq W \subseteq V$ and $w_i \prec w_j$ if $i \leq j$ and $w'_i \prec w'_j$ if $i \leq j$. Then the projection of t from W to W', written $t \downarrow^{W}_{W'}$, is defined as the tuple $t' = \langle t'_1, \ldots, t'_m \rangle$ with $t'_i = t_j$ if $w'_i = w_j$.*

Definition 2.2.5 (combination). *Given a constraint system $CS = \langle S, D, V \rangle$, where $S = \langle A, +, \times, \mathbf{0}, \mathbf{1} \rangle$, and two constraints $c_1 = \langle def_1, con_1 \rangle$ and $c_2 =$*

[2] Given any integer k, a k-tuple is just a tuple of length k. Also, given a set S, an S-tuple is a tuple with as many elements as the size of S.

$\langle def_2, con_2\rangle$ over CS, their combination, written $c_1 \otimes c_2$, is the constraint $c = \langle def, con\rangle$ with

$$con = con_1 \cup con_2$$

and

$$def(t) = def_1(t \downarrow^{con}_{con_1}) \times def_2(t \downarrow^{con}_{con_2}).$$

Since $\times$ is both commutative and associative, so too is $\otimes$. Thus this operation can be easily extended to more than two arguments, say $C = \{c_1, \ldots, c_n\}$, by performing $c_1 \otimes c_2 \otimes \ldots \otimes c_n$, which we will sometimes denote by $\bigotimes C$.

Definition 2.2.6 (projection). *Given a constraint system $CS = \langle S, D, V\rangle$, where $S = \langle A, +, \times, \mathbf{0}, \mathbf{1}\rangle$, a constraint $c = \langle def, con\rangle$ over CS, and a set I of variables ($I \subseteq V$), the projection of c over I, written $c \Downarrow_I$, is the constraint $\langle def', con'\rangle$ over CS with*

$$con' = I \cap con$$

and

$$def'(t') = \Sigma_{\{t \mid t\downarrow^{con}_{I\cap con} = t'\}} def(t).$$

Proposition 2.2.1 ($c \Downarrow_I = c \Downarrow_{I\cap con}$). *Given a constraint $c = \langle def, con\rangle$ over a constraint system CS and a set I of variables ($I \subseteq V$), we have that $c \Downarrow_I = c \Downarrow_{I\cap con}$.*

Proof. Follows trivially from Definition 2.2.6.

A useful property of the projection operation is the following.

Theorem 2.2.1 ($c \Downarrow_{S_1}\Downarrow_{S_2} = c \Downarrow_{S_2}$). *Given a constraint c over a constraint system CS, we have that $c \Downarrow_{S_1}\Downarrow_{S_2} = c \Downarrow_{S_2}$ if $S_2 \subseteq S_1$.*

Proof. To prove the theorem, we have to show that the two constraints $c_1 = c \Downarrow_{S_1}\Downarrow_{S_2}$ and $c_2 = c \Downarrow_{S_2}$ coincide, that is, they have the same *con* and the same *def*. Assume $c = \langle def, con\rangle$, $c_1 = \langle def_1, con_1\rangle$, and $c_2 = \langle def_2, con_2\rangle$. Now, $con_1 = S_2 \cap (S_1 \cap con)$. Since $S_2 \subseteq S_1$, we have that $con_1 = S_2 \cap con$. Also, $con_2 = S_2 \cap con$, thus $con_1 = con_2$. Consider now def_1. By Definition 2.2.6, we have that $def_1(t_1) = \Sigma_{\{t' \mid t'\downarrow^{S_1}_{S_2} = t_1\}} \Sigma_{\{t \mid t\downarrow^{con}_{S_1} = t'\}} def(t)$, which, by associativity of $+$, is the same as $\Sigma_{\{t \mid t\downarrow^{con}_{S_2} = t_1\}} def(t)$, which coincides with $def_2(t_1)$ by Definition 2.2.6.

We will now prove a property which can be seen as a generalization of the distributivity property in predicate logic, which we recall is $\exists x.(p \wedge q) = (\exists x.p) \wedge q$ if x not free in q (where p and q are two predicates). The extension we prove for our framework is given by the fact that $\otimes$ can be seen as a generalized $\wedge$ and $\Downarrow$ as a variant[3] of $\exists$. This property will be useful later in Section 3.6.

[3] Note, however, that the operator corresponding to $\exists x$ is given by $\Downarrow^{W}_{W-\{x\}}$.

Theorem 2.2.2. *Given two constraints* $c_1 = \langle def_1, con_1 \rangle$ *and* $c_2 = \langle def_2, con_2 \rangle$ *over a constraint system* CS*, we have that* $(c_1 \otimes c_2) \Downarrow_{(con_1 \cup con_2)-x} = c_1 \Downarrow_{con_1 - x} \otimes c_2$ *if* $x \cap con_2 = \emptyset$*.*

Proof. Let us set $c = (c_1 \otimes c_2) \Downarrow_{(con_1 \cup con_2)-x} = \langle def, con \rangle$ and $c' = c_1 \Downarrow_{con_1 - x} \otimes c_2 = \langle def', con' \rangle$. We will first prove that $con = con'$: $con = (con_1 \cup con_2) - x$ and $con' = (con_1 - x) \cup con_2$, which is the same as con if $x \cap con_2 = \emptyset$, as we assumed. Now we prove that $def = def'$. By definition,

$$def(t) = \Sigma_{\{t' \mid t' \downarrow^{con_1 \cup con_2}_{con_1 \cup con_2 - x} = t\}} (def_1(t' \downarrow^{con_1 \cup con_2}_{con_1}) \times def_2(t' \downarrow^{con_1 \cup con_2}_{con_2})).$$

Now, $def_2(t' \downarrow^{con_1 \cup con_2}_{con_2})$ is independent from the summation, since x is not involved in c_2. Thus it can be taken out. Also, $t' \downarrow^{con_1 \cup con_2}_{con_2}$ can be substituted by $t \downarrow^{con_1 \cup con_2}_{con_2}$, since t' and t must coincide on the variables different from x. Thus we have:

$$(\Sigma_{\{t' \mid t' \downarrow^{con_1 \cup con_2}_{con_1 \cup con_2 - x} = t\}} def_1(t' \downarrow^{con_1 \cup con_2}_{con_1})) \times def_2(t \downarrow^{con_1 \cup con_2 - x}_{con_2}).$$

Now, the summation is done over those tuples t' that involve all the variables and coincide with t on the variables different from x. By observing that the elements of the summation are given by $def_1(t' \downarrow^{con_1 \cup con_2}_{con_1})$, and thus they only contain variables in con_1, we can conclude that the result of the summation does not change if it is done over the tuples involving only the variables in con_1 and still coinciding with t over the variables in con_1 that are different from x. Thus we get:

$$(\Sigma_{\{t_1 \mid t_1 \downarrow^{con_1}_{con_1 - x} = t \downarrow^{con_1 \cup con_2 - x}_{con_1 - x}\}} def_1(t_1)) \times def_2(t \downarrow^{con_1 \cup con_2 - x}_{con_2}).$$

It is easy to see that this formula is exactly the definition of def'.

Using the properties of $\times$ and $+$, it is easy to prove that: $\otimes$ is associative and commutative; $\otimes$ is monotone over $\sqsubseteq_S$. Moreover, if $\times$ is idempotent $\otimes$ is idempotent.

Using the operations of combination and projection, we can now define the notion of solution of an SCSP. In the following we will consider a fixed constraint system $CS = \langle S, D, V \rangle$, where $S = \langle A, +, \times, \mathbf{0}, \mathbf{1} \rangle$.

Definition 2.2.7 (solution). *Given a constraint problem* $P = \langle C, con \rangle$ *over a constraint system* CS*, the solution of* P *is a constraint defined as* $Sol(P) = (\bigotimes C) \Downarrow_{con}$*.*

In words, the solution of an SCSP is the constraint induced on the variables in *con* by the whole problem. Such constraint provides, for each tuple of values of D for the variables in *con*, a corresponding value of A. Sometimes, it is enough to know just the best value associated with such tuples. In our framework, this is still a constraint (over an empty set of variables), and will be called the best level of consistency of the whole problem, where the meaning of "best" depends on the ordering $\leq_S$ defined by the additive operation.

Definition 2.2.8 (best level of consistency). *Given an SCSP $P = \langle C, con \rangle$, we define $blevel(P) \in S$ such that $\langle blevel(P), \emptyset \rangle = (\bigotimes C) \Downarrow_{\emptyset}$. Moreover, we say that P is consistent if $\mathbf{0} <_S blevel(P)$, and that P is α-consistent if $blevel(P) = \alpha$.*

Informally, the best level of consistency gives us an idea of how much we can satisfy the constraints of the given problem. Note that $blevel(P)$ does not depend on the choice of the distinguished variables, due to the associative property of the additive operation. Thus, since a constraint problem is just a set of constraints plus a set of distinguished variables, we can also apply function *blevel* to a set of constraints only. More precisely, $blevel(C)$ will mean $blevel(\langle C, con \rangle)$ for any *con*.

Note that $blevel(P)$ can also be obtained by first computing the solution and then projecting such a constraint over the empty set of variables, as the following proposition shows.

Proposition 2.2.2. *Given an SCSP P, we have that $Sol(P) \Downarrow_{\emptyset} = \langle blevel(P), \emptyset \rangle$.*

Proof. $Sol(P) \Downarrow_{\emptyset}$ coincides with $((\bigotimes C) \Downarrow_{con}) \Downarrow_{\emptyset}$ by definition of $Sol(P)$. This coincides with $(\bigotimes C) \Downarrow_{\emptyset}$ by Theorem 2.2.1, thus it is the same as $\langle blevel(P), \emptyset \rangle$ by Definition 2.2.8.

Another interesting notion of solution, more abstract than the one defined above, but sufficient for many purposes, is the one that does not consider those tuples whose associated value is worse (w.r.t. $\leq_S$) than that of other tuples.

Definition 2.2.9 (abstract solution). *Given an SCSP $P = \langle C, con \rangle$, consider $Sol(P) = \langle def, con \rangle$. Then the abstract solution of P is the set $ASol(P) = \{\langle t, v \rangle \mid def(t) = v$ and there is no t' such that $v \leq_S def(t')\}$.*

Note that, when the $\leq_S$ ordering is a total order (that is, when, for any a and b, $a + b$ is either a or b), then the abstract solution contains only those tuples whose associated value coincides with $blevel(P)$. In general, instead, an incomparable set of tuples is obtained, and thus $blevel(P)$ is just an upper bound of the values associated with the tuples.

By using the ordering $\leq_S$ over the semiring, we can also define a corresponding ordering on constraints with the same type.

Definition 2.2.10 (constraint ordering). *Consider two constraints c_1, c_2 over CS, and assume that $con_1 = con_2$ and $|con_1| = k$. Then $c_1 \sqsubseteq_S c_2$ if and only if, for all k-tuples t of values from D, $def_1(t) \leq_S def_2(t)$.*

Notice that, if $c_1 \sqsubseteq_S c_2$ and $c_2 \sqsubseteq_S c_1$, then $c_1 = c_2$.

Theorem 2.2.3 ($\sqsubseteq_S$ is a partial order). *The relation $\sqsubseteq_S$ over the set of constraints over CS is a partial order.*

Proof. By definition, such a relation is antisymmetric. Also, it is easy to see that it is reflexive and transitive.

The notion of constraint ordering, and the fact that the solution is a constraint, can also be useful to define an ordering on problems.

Definition 2.2.11 (problem ordering and equivalence). *Consider two SCSPs $P_1 = \langle C_1, con \rangle$ and $P_2 = \langle C_2, con \rangle$ over CS. Then $P_1 \sqsubseteq_P P_2$ if $Sol(P_1) \sqsubseteq_S Sol(P_2)$. If $P_1 \sqsubseteq_P P_2$ and $P_2 \sqsubseteq_P P_1$, then they have the same solution. Thus we say that P_1 and P_2 are equivalent and we write $P_1 \equiv P_2$.*

Theorem 2.2.4 ($\sqsubseteq_P$ is a preorder and $\equiv$ is an equivalence). *The relation $\sqsubseteq_P$ over the set of constraint problems over CS is a preorder. Moreover, $\equiv$ is an equivalence relation.*

Proof. It is trivial to see that $\sqsubseteq_P$ is reflexive and transitive, due to the definition of constraint ordering $\sqsubseteq_S$. Thus $\sqsubseteq_P$ is a preorder. From this it derives that $\equiv$ is reflexive and transitive as well. Moreover, it is trivially symmetric.

The ordering $\sqsubseteq_P$ can also be used for ordering sets of constraints, since a set of constraint is just a problem where *con* contains all the variables.

It is interesting now to note that, as in the classical CSP case, also the SCSP framework is monotone. That is, if some constraints are added, the solution of the new problem precedes that of the old one in the ordering $\sqsubseteq_S$. In other words, the new problem precedes the old one w.r.t. the preorder $\sqsubseteq_P$.

Theorem 2.2.5 (monotonicity). *Consider two SCSPs $P_1 = \langle C_1, con \rangle$ and $P_2 = \langle C_1 \cup C_2, con \rangle$ over CS. Then $P_2 \sqsubseteq_P P_1$ and $blevel(P_2) \leq_S blevel(P_1)$.*

Proof. $P_2 \sqsubseteq_P P_1$ follows from the intensivity of $\times$ and the monotonicity of $+$. The same holds also for $blevel(P_2) \leq_S blevel(P_1)$, since both Sol and $blevel(P)$ are obtained by projecting over a subset of the variables (which is always empty in the case of $blevel(P)$).

When one is interested in the abstract solution rather than in the solution of an SCSP, then the above notion of monotonicity is lost. In fact, it is possible that by adding some constraints the best level of consistency gets worse, while the set of tuples in the abstract solution grows. See for example Figure 2.1, where there is a problem P_1 with one constraint and both variables as type, and a problem P_2 with the same constraint plus another one, and the two leftmost variables as type. Assume also that $\times$ is *min* and that $+$ is *max*. As we will see later, this amounts to considering the fuzzy constraint satisfaction framework (FCSP) (with the set of reals as carrier set instead of the set $[0, 1]$. Now, it is easy to see that $blevel(P_1) = 3$ and $blevel(P_2) = 2$. The abstract solution of P_1 contains the tuples $\langle a, a \rangle$ and $\langle b, a \rangle$, while that of P_2 contains also $\langle a, b \rangle$ and $\langle b, b \rangle$. Thus, adding one constraint in this case makes the best level of consistency worse while enlarging the set of tuples in the abstract solution.

Another notion of monotonicity holds, however, but only when the best level is not changed by the addition of some constraints and the order $\leq_S$ is total.

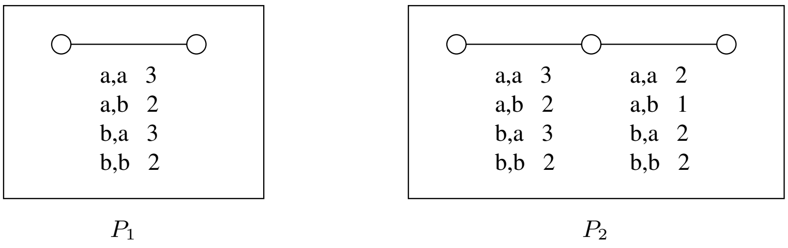


Fig. 2.1. Two SCSPs

Theorem 2.2.6 (subset-monotonicity vs. Asol). *Consider two SCSPs* $P_1 = \langle C_1, con\rangle$ *and* $P_2 = \langle C_1 \cup C_2, con\rangle$ *over* CS*, and assume that* $\leq_S$ *is total and that* $blevel(P_1) = blevel(P_2)$*. Then* $Asol(P_2) \subseteq Asol(P_1)$*.*

Proof. If $\leq_S$ is total, then the abstract solution is a set of tuples with associated value exactly the best level. Take a tuple t in $ASol(P_2)$. This means that the value associated with t in P_2 is $blevel(P_2)$. Then, by Theorem 2.2.5, the value associated with t in P_1, say v, is such that $blevel(P_2) \leq_S v$. By assumption, $blevel(P_1) = blevel(P_2)$. If $v \neq blevel(P_2)$, then this assumption is violated. Thus it must be $v = blevel(P_2) = blevel(P_1)$. Thus t is also in $Asol(P_1)$. Therefore $Asol(P_2) \subseteq Asol(P_1)$.

2.3 Instances of the Framework

We will now show how several known, and also new, frameworks for constraint solving may be seen as instances of the SCSP framework. More precisely, each of such frameworks corresponds to the choice of a specific constraint system (and thus of a semiring). This means that we can immediately know whether one can inherit the properties of the general framework by just looking at the properties of the operations of the chosen semiring, and by referring to the theorems in the previous sections. This is interesting for known constraint solving schemes, because it puts them into a single unifying framework and it justifies in a formal way many informally taken choices, but it is especially significant for new schemes, for which one does not need to prove all the properties that it has (or not) from scratch.

Note that the constraint systems of different instances differ only in the choice of the semiring. Therefore, in the following we will only specify the semiring that has to be chosen to obtain a particular instance of the SCSP framework.

2.3.1 Classical CSPs

As described in Section 1.3 a classical CSP problem [139, 150] is just a set of variables and constraints, where each constraint specifies the tuples that are allowed for the involved variables. Assuming the presence of a subset of distinguished variables, the solution of a CSP consists of a set of tuples that represent the assignments of the distinguished variables, which can be extended to total assignments (for all the values) while satisfying all the constraints.

Since constraints in CSPs are crisp, that is, they either allow a tuple or not, we can model them via a semiring domain with only two values, say 1 and 0: allowed tuples will have associated the value 1, and not-allowed ones the value 0. Moreover, in CSPs, constraint combination is achieved via a join operation among allowed tuple sets. This can be modeled here by assuming the multiplicative operation to be the logical *and* (and interpreting 1 as true and 0 as false). Finally, to model the projection over some of the variables (for example the distinguished ones), as the k-tuples (assuming we project over k variables) for which there exists a consistent extension to an n-tuple, it is enough to assume the additive operation to be the logical *or*. Therefore, a CSP is just an SCSP where the semiring is

$$S_{CSP} = \langle \{0,1\}, \vee, \wedge, 0, 1 \rangle.$$

Theorem 2.3.1. *$S_{CSP} = \langle \{0,1\}, \vee, \wedge, 0, 1 \rangle$ is a c-semiring.*

Proof. It is easy to see that it is a semiring. Thus, we will only prove the additional properties needed in a c-semiring:

– the additive operation must be idempotent; it is trivially true since the additive operation is $\vee$, and $a \vee a = a$ for any $a \in \{0,1\}$;
– the multiplicative operation must be commutative; trivially from the definition of $\wedge$;
– 1 is the absorbing element of the additive operation, that is $1 \vee a = 1$, which is trivially true.

The ordering $\leq_S$ here reduces to $0 \leq_S 1$.

Example 2.3.1. Consider the CSP P in Figure 2.2a), where the distinguished variables are marked and the constraints are arcs labeled by the allowed tuples. The solution of such a CSP is the singleton set containing tuple $\langle a, a \rangle$ (which means $x = a$ and $y = a$). This CSP can be seen as an SCSP P' over the semiring S_{CSP} in Figure 2.2b). It can then be shown, by using the semiring operations, that the solution of P and that of P' coincide (modulo the semiring domain value). To compute the solution of P', we have to assign a value to each tuple for the distinguished variables. This is done by first considering all 3-tuples and assigning a value to them, and then by projecting (via the $\vee$ operation) such values over the distinguished variables. To assign a value to a 3-tuple, one has to combine (via the $\wedge$ operation) the values of all the subtuples corresponding to subsets of variables which are connected by some constraint. In Figure 2.3, for each 3-tuple we have written the values of its subtuples: subtuples are pointed out by underlining them, and subtuples of one element only are not indicated, since they all have value 1. Then the value for the tuple itself can be seen to the right as a combination of the subtuple values (again, for simplicity of the picture one-value subtuples are not considered since they always have value 1), and the value for each 2-tuple which is a projection of some 3-tuples onto the variables x and y can be seen to the left, as a combination of the values of the two tuples which have the same projection over x and y. It is easy to see that a 3-tuple gets the value 1 only when it satisfies all the constraints (due to the definition of $\wedge$), and that a 2-tuple gets a value 1 when there exists a possible extension to a 3-tuple whose value is 1 (due to the definition of $\vee$). From Figure 2.3, one can see that the only group of tuples that gets value 1 is the first one, thus the solution of the CSP is the projection of this group of tuples onto x and y, that is, $x = a$ and $y = a$.

It is easy to see, also from the example above, that the chosen c-semiring allows us to faithfully represent any given CSP. That is, taken the given CSP P and the SCSP P' obtained by applying a suitable transformation to P, the solutions of P and P' are the same modulo a similar transformation.

It is possible to check that an alternative way to represent CSPs in the SCSP framework is by using the following c-semiring: $\langle \wp(\{a\}), \bigcup, \bigcap, \emptyset, \{a\} \rangle$, where $\wp(S)$ is the powerset of a set S, and a is any value.

2.3.2 Fuzzy CSPs (FCSPs)

Fuzzy CSPs (FCSPs) [91,165,167] extend the notion of classical CSPs by allowing non-crisp constraints, that is, constraints which associate a preference level with each tuple of values. Such level is always between 0 and 1, where 1 represents the best value (that is, the tuple is allowed) and 0 the worst one (that is, the tuple is not allowed). The solution of a fuzzy CSP is then defined as the set of tuples of values (for all the variables) which have the maximal value. The way they associate a value with an n-tuple is by minimizing the values of all its subtuples. The reason for such a max-min framework relies on the attempt to maximize the value of the least preferred tuple.

Fuzzy CSPs are already a very significant extension of CSPs. In fact, they are able to model partial constraint satisfaction (as described in Section 1.4.1), so as to get a solution even when the problem is overconstrained, and also prioritised constraints, that is, constraints with different levels of importance (see Section 1.4.2).

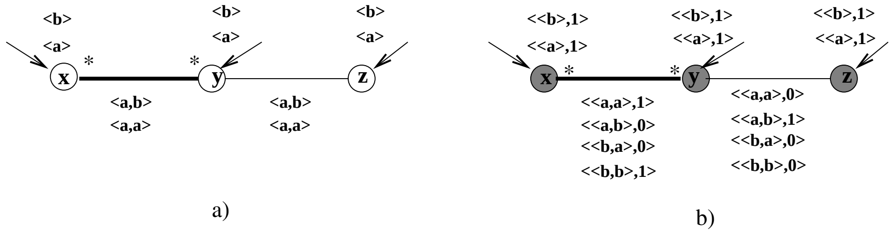


Fig. 2.2. A CSP and the corresponding SCSP

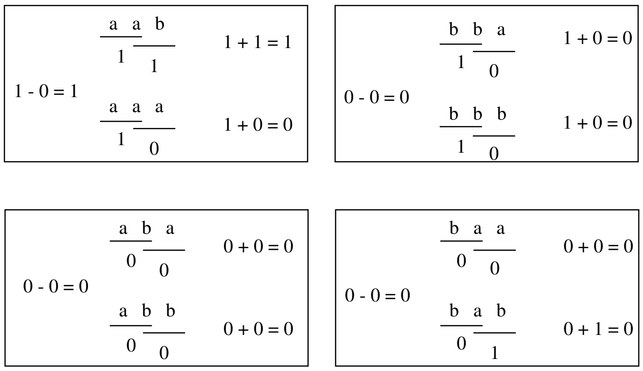


Fig. 2.3. Combination and projection in classical CSPs

Definition 2.3.1 (Fuzzy CSPs). *A Fuzzy Constraint Satisfaction Problem (FCSP) consist of:*

- *a set of variables $X = \{x_1, \ldots, x_n\}$,*
- *for each variable x_i, a finite set D_i of possible values (its domain),*
- *a set of constraints; each constraint c is defined by a function fuzzy level $fl(c, A)$, that assigns a real number between 0 and 1 to each tuple A of domain values.*

A solution to a FCSP is an assignment of a value from its domain to every variable such that the expression $min\{fl(c, A) \mid c$ is a constraint in FCSP$\}$ is maximized among all possible assignments A of domain values.

It is easy to see that fuzzy CSPs can be modeled in our framework by choosing the semiring

$$S_{FCSP} = \langle \{x \mid x \in [0, 1]\}, max, min, 0, 1\rangle.$$

Theorem 2.3.2. *S_{FCSP} is a c-semiring.*

Proof. It is easy to see that it is a semiring. Thus we will only prove the additional properties that are needed in a c-semiring:

- the additive operation must be idempotent; this is trivially true since the additive operation is max, and $max(a, a) = a$ for any a;
- the multiplicative operation must be commutative; trivially from the definition of min;
- 1 is the absorbing element of the additive operation, that is $max(1, a) = 1$, which is true since $0 \leq a \leq 1$.

The ordering $\leq_S$ here reduces to the $\leq$ ordering on reals. It is also easy to see that any FCSP P can be rewritten as an SCSP P' over the semiring S_{FCSP} such that $sol(P) = sol(P')$ (modulo a suitable transformation between the problems and the solutions).

Fuzzy CSPs are much closer to our framework than classical CSPs, since they already introduce the notion of preference levels. This means that they have to generalize the notions of constraint combination and projection as well, so as to get the appropriate definition of solution. We are, however, obviously much more general in that we do not make any assumption on the semiring operations.

2.3.3 Probabilistic CSPs (Prob-CSPs)

Probabilistic CSPs (Prob-CSPs) [96] were introduced to model those situations where each constraint c has a certain probability $p(c)$, independent from the probability of the other constraints, to be part of the given problem (actually, the probability is not of the constraint, but of the situation which corresponds to the constraint: saying that c has probability p means that the situation corresponding to c has probability p of occurring in the real-life problem).

This allows one to also reason about problems which are only partially known. The probability levels on constraints then gives to each instantiation of all the variables, a probability that it is a solution of the real problem. This is done by first associating with each subset of constraints the probability that it is in the real problem (by multiplying the probabilities of the involved constraints), and then by summing up all the probabilities of the subsets of constraints where the considered instantiation is a solution. Alternatively, the probability associated with an n-tuple t can also be seen as the probability that all constraints that t violates are indeed in the real problem. This is just the product of all $1 - p(c)$ for all c violated by t. Finally, the aim is to get those instantiations with the maximum probability.

Definition 2.3.2 (Probabilistic CSPs). *A Probabilistic Constraint Satisfaction Problem (Prob-CSP) consists of:*

- *a set of variables $X = \{x_1, \ldots, x_n\}$,*
- *for each variable x_i, a finite set D_i of possible values (its domain),*
- *a set of constraints restricting the values that the variables can simultaneously take; each constraint c is also assigned a certain probability $p(c)$ to be part of the given problem.*

There are two alternative approaches to define the solution to a Prob-CSP. Again, the solution is an assignment of a value from its domain to every variable such that

1. *the expression Sum$\{$Product$\{p(c) \mid c$ is a constraint in $S\} \mid S$ is a subset of the set of all constraints satisfied by $A\}$ is maximized among all possible assignments A of values, or*
2. *the expression Product$\{(1-p(c)) \mid c$ is a constraint in FCSP violated by $A\}$ is maximized among all possible assignments A of values.*

The relationship between Prob-CSPs and SCSPs is complicated by the fact that Prob-CSPs contain crisp constraints with probability levels, while SCSPs contain non-crisp constraints. That is, we associate values with tuples, and not to constraints. It is still possible however to model Prob-CSPs, by using a transformation which is similar to that proposed in [91] to model prioritised constraints via soft constraints in the FCSP framework. More precisely, we assign probabilities to tuples instead of constraints. Consider any constraint c with probability $p(c)$, and let t be any tuple of values for the variables involved in c. Then $p(t) = 1$ if t is allowed by c, otherwise $p(t) = 1 - p(c)$. The reasons for such a choice are as follows: if a tuple is allowed by c and c is in the real problem, then t is allowed in the real problem; this happens with probability $p(c)$; if instead c is not in the real problem, then t is still allowed in the real problem, and this happens with probability $1 - p(c)$. Thus, t is allowed in the real problem with probability $p(c) + 1 - p(c) = 1$. Consider instead a tuple t which is not allowed by c. Then it will be allowed in the real problem only if c is not present; this happens with probability $1 - p(c)$.

To give the appropriate value to an n-tuple t, given the values of all the smaller k-tuples, with $k \leq n$ and which are subtuples of t (one for each constraint), we just perform the product of the value of such subtuples. By the manner in which values have been assigned to tuples in constraints, this coincides with the product of all $1 - p(c)$ for all c violated by t. In fact, if a subtuple violates c, then by construction its value is $1 - p(c)$; if instead a subtuple satisfies c, then its value is 1. Since 1 is the unit element of $\times$, we have that $1 \times a = a$ for each a. Thus we get $\Pi(1 - p(c))$ for all c that t violates.

As a result, the semiring corresponding to the Prob-CSP framework is

$$S_{prob} = \langle \{x \mid x \in [0,1]\}, max, \times, 0, 1 \rangle.$$

Theorem 2.3.3. *S_{prob} is a c-semiring.*

Proof. It is a semiring. To be also a c-semiring, we need the following:

- the additive operation must be idempotent; this is trivially true since the additive operation is max, and $max(a, a) = a$ for any a;
- the multiplicative operation must be commutative; trivially from the definition of $\times$;
- 1 is the absorbing element of the additive operation, that is $max(1, a) = 1$, which is true since $0 \leq a \leq 1$.

The associated ordering $\leq_S$ here reduces to $\leq$ over reals.

It is easy to see that any Prob-CSP P can be rewritten as a n SCSP P' over the semiring S_{prob}, such that $sol(P) = sol(P')$ (modulo a suitable transformation between the problems and the solutions). Note that the fact that P' is α-consistent means that in P there exists an n-tuple which has probability α to be a solution of the real problem.

2.3.4 Weighted CSPs

While fuzzy CSPs associate a level of preference with each tuple in each constraint, in weighted CSPs (WCSPs) tuples come with an associated cost. This allows one to model optimization problems where the goal is to minimize the total cost (time, space, number of resources, and so on) of the proposed solution. Therefore, in WCSPs the cost function is defined by summing up the costs of all constraints (intended as the cost of the chosen tuple for each constraint). Thus, the goal is to find the n-tuples (where n is the number of all the variables) which minimize the total sum of the costs of their subtuples (one for each constraint).

Another way to understand the difference between WCSPs and FCSPs is that FCSPs have an egalitarianistic approach to optimization problems (that is, they aim at maximizing the overall level of consistency while balancing the levels of all constraints), while WCSPs have an utilitarianistic approach (that is, they aim at getting the minimum cost globally, even though some constraints may be neglected and thus present a big cost) [154]. We believe that both approaches present advantages and drawbacks, and thus one may be preferred to the other, depending on the real-life situation to be modeled.

According to the informal description of WCSPs given above, the associated semiring is

$$S_{WCSP} = \langle \mathcal{R}^+, \min, +, +\infty, 0 \rangle.$$

Theorem 2.3.4. *S_{WCSP} is a c-semiring.*

Proof. It is easy to see that it is a semiring. To be a c-semiring, we need:

– the additive operation must be idempotent; this is trivially true since the additive operation is min, and $min(a, a) = a$ for any a;
– the multiplicative operation must be commutative; trivially from the definition of $+$;
– 0 is the absorbing element of the additive operation, that is $min(0, a) = 0$, which is true since $a \geq 0$.

The associated ordering $\leq_S$ reduces here to $\geq$ over the reals. This means that a value is preferred to another one if it is smaller.

Note that the same properties hold also for the semirings $\langle \mathcal{Q}^+, min, +, +\infty, 0\rangle$ and $\langle \mathcal{Z}^+, min, +, +\infty, 0\rangle$ (which can be proved to be c-semirings).

2.3.5 Egalitarianism and Utilitarianism

As noted above, the FCSP and the WCSP systems can be seen as two different approaches to giving a meaning to the notion of optimization. In fact, the two models correspond on two definitions of social welfare in utility theory [154]: *egalitarianism*, which maximizes the minimal individual utility, and *utilitarianism*, which maximizes the sum of the individual utilities. FCSPs are based on the egalitarian approach, while WCSPs are based on utilitarianism.

In this section we will show how our framework also allows for the combination of these two approaches. In fact, we will construct an instance of the SCSP framework where the two approaches coexist, and allow us to discriminate among solutions which otherwise would result indistinguishable. More precisely, we can first compute the solutions according to egalitarianism (that is, using a max-min computation as in FCSPs), and then discriminate more among them via utilitarianism (that is, using a max-sum computation as in WCSPs).

The resulting semiring is the following:

$$S_{ue} = \langle \{\langle l, k \rangle \mid l \in [0,1] \text{ and } k \in \mathcal{R}^-\}, \underline{max}, \underline{min}, \langle 0, -\infty \rangle, \langle 1, 0 \rangle \rangle$$

where $\underline{max}$ and $\underline{min}$ are defined as follows:

- $\langle l_1, k_1 \rangle \underline{max} \langle l_2, k_2 \rangle = \begin{cases} \langle l_1, max(k_1, k_2) \rangle & \text{if } l_1 = l_2 \\ \langle l_1, k_1 \rangle & \text{if } l_1 > l_2 \end{cases}$
- $\langle l_1, k_1 \rangle \underline{min} \langle l_2, k_2 \rangle = \begin{cases} \langle l_1, k_1 + k_2 \rangle & \text{if } l_1 = l_2 \\ \langle l_2, k_2 \rangle & \text{if } l_1 > l_2 \end{cases}$

That is, the domain of the semiring contains pairs of values: the first element is used to reason via the max-min approach, while the second one is used to further discriminate via the max-sum approach. The operation $\underline{\text{min}}$ (which is the multiplicative operation of the semiring, and thus is used to perform the constraint combination) takes two elements of the semiring, say $a = \langle a_1, a_2 \rangle$ and $b = \langle b_1, b_2 \rangle$) and returns the one with the smallest first element (if $a_1 \neq b_1$), otherwise it returns the pair that has the sum of the second elements as its second element (that is, $\langle a_1, a_2 + b_2 \rangle$). The operation $\underline{\text{max}}$ performs a max on the first elements if they are different (thus returning the pair with the maximum first element), otherwise (if $a_1 = b_1$) it performs a max on the second elements (thus returning a pair which has such a max as second element, that is, $\langle a_1, max(a_2, b_2) \rangle$). More abstractly, we can say that, if the first elements of the pairs differ, then the $\underline{\text{max}}$-$\underline{\text{min}}$ operations behave like a normal max-min, otherwise they behave like max-sum. This can be interpreted as the fact that, if the first elements coincide, it means that the max-min criterion cannot discriminate enough, and thus the max-sum criterion is used.

One can show that S_{ue} is a c-semiring.

2.3.6 Set-Based CSPs

An interesting class of instances of the SCSP framework that are based on set operations like union and intersection is the one which uses the c-semiring

$$S_{set} = \langle \wp(A), \bigcup, \bigcap, \emptyset, A \rangle$$

where A is any set. It is easy to see that S_{set} is a c-semiring. Also, in this case the order $\leq_{S_{set}}$ reduces to set inclusion (in fact, $a \leq b$ iff $a \cup b = b$), and therefore it is partial in general. Furthermore, $\times$ is $\bigcap$ in this case, and thus it is idempotent.

A useful application of such SCSPs based on a set occurs when the variables of the problem represent processes, the finite domain D is the set of possible states of such processes, and A is the set of all time intervals over reals. In this case, a value t associated to a pair of values $\langle d_1, d_2 \rangle$ for variables x and y can be interpreted as the set of all time intervals in which process x can be in state d_1 and process y in state d_2. Therefore, a solution of any SCSP based on the c-semiring S_{set} consists of a tuple of process states, together with (the set of) the time intervals in which such system configuration can occur. This description of a system can be helpful, for example, to discover if there are time intervals in which a deadlock is possible.

2.3.7 Valued Constraint Problems

The framework [174] described in this subsection is based on an ordered monoid structure (that is, an ordered domain plus one operation satisfying some properties). The values of the domain are interpreted as levels of violation (which can be interpreted as cost, or degrees of preference, or probabilities, or others) and can be combined using the monoid operator. Specific choices of the monoid will then give rise to different instances of the framework.

Elements of the set of the monoid, called valuations, are associated with each constraint and the monoid operation (denoted $\circledast$) is used to assign a valuation to each assignment, by combining the valuations of all the constraints violated by the assignment. The order on the monoid is assumed to be total and the problem considered is always to minimize the combined valuation of all violated constraints. The reader should note that the choice of associating one valuation to each constraint rather than to each tuple (as in the SCSP framework) is done only for the sake of simplicity and is not fundamentally different.

In this section, a classical CSP is defined by a set $V = \{v_1, \ldots, v_n\}$ of variables, each variable v_i having an associated finite domain d_i. A constraint $c = (V_c, R_c)$ is defined by a set of variables $V_c \subseteq V$ and a relation R_c between the variables of V_c *i.e.*, a subset of the Cartesian product $\prod_{v_i \in V_c} d_i$. A CSP is denoted by $\langle V, D, C\rangle$, where D is the set of the domains and C the set of the constraints. A solution of the CSP is an assignment of values to the variables in V such that all the constraints are satisfied: for each constraint $c = (V_c, R_c)$, the tuple of the values taken by the variables of V_c belongs to R_c.

Valuation Structure. In order to deal with over-constrained problems, it is necessary to be able to express the fact that a constraint may eventually be violated. To achieve this, we annotate each constraint with a mathematical item which we call a *valuation*. Such valuations will be taken from a set E equipped with the following structure:

Definition 2.3.3 (Valuation structure). *A* valuation structure *is defined as a tuple $\langle E, \circledast, \succ\rangle$ such that:*

- *E is a set, whose elements are called valuations, which is totally ordered by $\succ$, with a maximum element noted $\top$ and a minimum element noted $\bot$;*
- *$\circledast$ is a commutative, associative closed binary operation on E that satisfies:*
  - Identity*: $\forall a \in E, a \circledast \bot = a$;*
  - Monotonicity*: $\forall a, b, c \in E, (a \succcurlyeq b) \Rightarrow \big((a \circledast c) \succcurlyeq (b \circledast c)\big)$;*
  - Absorbing element*: $\forall a \in E, (a \circledast \top) = \top$.*

Valued CSP. A valued CSP is then simply obtained by annotating each constraint of a classical CSP with a valuation denoting the impact of its violation or, equivalently, of its rejection from the set of constraints.

Definition 2.3.4 (Valued CSP). *A* valued CSP *is defined by a classical CSP $\langle V, D, C\rangle$, a valuation structure $S = (E, \circledast, \succ)$, and an application φ from C to E. It is denoted by $\langle V, D, C, S, \varphi\rangle$. $\varphi(c)$ is called the valuation of c.*

An assignment A of values to some variables $W \subset V$ can now be simply evaluated by combining the valuations of all the violated constraints using $\circledast$:

Definition 2.3.5 (Valuation of assignments). *In a Valued CSP $\mathcal{P} = \langle V, D, C, S, \varphi\rangle$* the valuation of an assignment *A of the variables of $W \subset V$ is defined by:*

$$\mathcal{V}_{\mathcal{P}}(A) = \underset{\substack{c \in C, V_c \subset W \\ A \text{ violates } c}}{\circledast} [\varphi(c)]$$

The semantics of a VCSP is a distribution of valuations on the assignments of V (potential solutions). The problem considered is to find an assignment A with a *minimum* valuation. The valuation of such an optimal solution will be called the CSP valuation. It provides a *gradual* notion of inconsistency, from $\bot$, which corresponds to consistency, to $\top$, for complete inconsistency.

From SCSPs to VCSPs. The SCSP and the VCSP framework are similar, but, however, they are not completely equivalent: only if one assumes a total order on the semiring set, it will be possible to define appropriate mappings to pass from one of them to the other.

In this section we compare the two approaches. Some more details related to the two approaches can be found in [38, 39].

We will consider SCSPs $\langle C, con\rangle$ where con involves all variables. Thus we will omit it in the following. An SCSP is, therefore, just a set of constraints C over a constraint system $\langle S, D, V\rangle$, where S is a c-semiring $S = \langle A, +, \times, \mathbf{0}, \mathbf{1}\rangle$, and D is the domain of the variables in V. Moreover, we will assume that the $+$ operation induces an order $\leq_S$ which is total. This means that $+$ corresponds to max, or, in other words, that it always chooses the value which is closer to $\mathbf{1}$.

Given an SCSP, we will now show how to obtain a corresponding VCSP, where by correspondence we mean that they associate the same value with each variable assignment, and that they have the same solution.

Definition 2.3.6. *Given an SCSP P with constraints C over a constraint system $\langle S, D, V\rangle$, where S is a c-semiring $S = \langle A, +, \times, \mathbf{0}, \mathbf{1}\rangle$, we obtain the VCSP $P' = \langle V, D, C', S', \varphi\rangle$, where $S' = \langle E, \circledast, \succ\rangle$, where $E = A$, $\circledast = \times$, and $\prec = >_S$. (Note that $\top = \mathbf{0}$ and $\bot = \mathbf{1}$.)*

For each constraint $c = \langle def, con\rangle \in C$, we obtain a set of constraints $c'_1, \ldots, c'_k$, where k is the cardinality of the range of def. That is, $k = |\{a$ s.t. $\exists t$ with $def(t) = a\}|$. Let us call $a_1, \ldots, a_k$ such values, and let us call T_i the *set of all tuples t such that $def(t) = a_i$. All the constraints c_i involve the same variables, which are those involved in c. Then, for each $i = 1, \dots, k$, we set $\varphi(c_i') = a_k$, and we define c_i' in such a way that the tuples allowed are those not in T_i. We will write $P' = sv(P)$. Note that, by construction, each tuple t is allowed by all constraints $c_1, \dots, c_k$ except the constraint c_i such that $\varphi(c_i) = def(t)$.*

Example 2.3.2. Consider an SCSP which contains the constraint $c = \langle def, con \rangle$, where $con = \{x, y\}$, and $def(\langle a, a \rangle) = l_1$, $def(\langle a, b \rangle) = l_2$, $def(\langle b, a \rangle) = l_3$, $def(\langle b, b \rangle) = l_1$. Then, the corresponding VCSP will contain the following three constraints, all involving x and y:

– c_1, with $\varphi(c_1) = l_1$ and allowed tuples $\langle a, b \rangle$ and $\langle b, a \rangle$;
– c_2, with $\varphi(c_2) = l_2$ and allowed tuples $\langle a, a \rangle$, $\langle b, a \rangle$ and $\langle b, b \rangle$;
– c_3, with $\varphi(c_3) = l_3$ and allowed tuples $\langle a, a \rangle$, $\langle a, b \rangle$ and $\langle b, b \rangle$.

Figure 2.4 shows both c and the three constraints c_1, c_2, and c_3.

First we need to make sure that the structure we obtain via the definition above is indeed a valued CSP. Then we will prove that it is equivalent to the given SCSP.

Theorem 2.3.5 (from c-semiring to valuation structure). *If we consider a c-semiring $S = \langle A, +, \times, \mathbf{0}, \mathbf{1} \rangle$ and the structure $S' = \langle E, \circledast, \succ \rangle$, where $E = A$, $\circledast = \times$, and $\prec = >_S$ (obtained using the transformation in Definition 2.3.6), then S' is a valuation structure.*

Proof. First, $\succ$ is a total order, since we assumed that $\leq_S$ is total and that $\prec = >_S$. Moreover, $\top = \mathbf{0}$ and $\bot = \mathbf{1}$, from the definition of $\leq_S$. Then, since $\circledast$ coincides with $\times$, it is easy to see that it is commutative, associative, monotone, and closed. Moreover, $\mathbf{1}$ (that is, $\bot$) is its unit element and $\mathbf{0}$ (that is, $\top$) is its absorbing element.

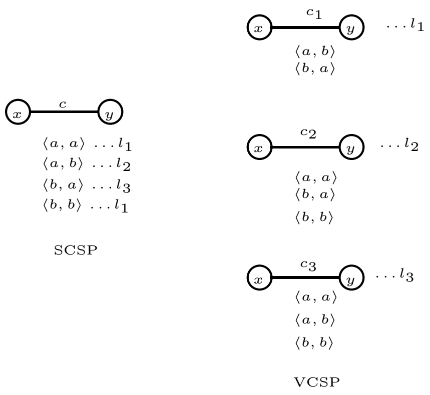


Fig. 2.4. From SCSPs to VCSPs

Theorem 2.3.6 (equivalence between SCSP and VCSP). *Consider an SCSP P and the corresponding VCSP problem P'. Consider also an assignment t to all the variables of P, with associated value $a \in A$. Then $\mathcal{V}_{P'}(t) = a$.*

Proof. Note first that P and P' have the same set of variables. In P, the value of t is obtained by multiplying the values associated with each subtuple of t, one for each constraint of P. Thus, $a = \prod\{def_c(t_c)$, for all constraints $c = \langle def_c, con_c \rangle$ in C and such that t_c is the projection of t over the variables of $c\}$. Now, $\mathcal{V}_{P'}(t) = \prod\{\varphi(c')$ for all $c' \in C'$ such that the projection of t over the variables of c' violates $c'\}$. It is easy to see that the number of values multiplied in this formula coincides with the number of constraints in C, since, as noted above, each tuple violates only one of the constraints in C' that have been generated because of the presence of the constraint $c \in C$. Thus, we just have to show that, for each $c \in C$, $def_c(t_c) = \varphi(c_i)$, where c_i is the constraint violated by t_c. But this is easy to show by what we have noted above. In fact, we have defined the translation from SCSP to VCSP in such a way that the only constraint of the VCSP violated by a tuple is exactly the one whose valuation coincides with the value associated with the tuple in the SCSP.

Note that SCSPs that do not satisfy the restrictions imposed at the beginning of the section, that is, that *con* involves all the variables and that $\leq_S$ is total, do not have a corresponding VCSP.

Corollary 2.3.1 (same solution). *Consider an SCSP P and the corresponding VCSP problem P'. Then P and P' have the same solution.*

Proof. It follows from Theorem 2.3.6, from the fact that P uses $+ = max$ which goes towards $\mathbf{1}$, that P' uses min which goes towards $\bot$, that $\mathbf{1} = \bot$, and that a solution is just one of the total assignments.

From VCSPs to SCSPs. Here we will define the opposite translation, which allows one to get an SCSP from a given VCSP.

Definition 2.3.7. *Given the VCSP $P = \langle V, D, C, S, \varphi \rangle$, where $S = \langle E, \circledast, \succ \rangle$, we will obtain the SCSP P' with constraints C' over the constraint system $\langle S', D, V \rangle$, where S' is the c-semiring $\langle E, +, \circledast, \top, \bot \rangle$, and $+$ is such that $a+b = a$ iff $a \preccurlyeq b$. It is easy to see that $\geq_S = \preccurlyeq$. For each constraint $c \in C$ with allowed set of tuples T, we define the corresponding constraint $c' = \langle con', def' \rangle \in C'$ such that con' contains all the variables involved in c and, for each tuple $t \in T$, $def'(t) = \bot$, otherwise $def'(t) = \varphi(c)$. We will write $P' = vs(P)$.*

Example 2.3.3. Consider a VCSP which contains a binary constraint c connecting variables x and y, for which it allows the pairs $\langle a, b \rangle$ and $\langle b, a \rangle$, and such that $\varphi(c) = l$. Then, the corresponding SCSP will contain the constraint $c' = \langle con, def \rangle$, where $con = \{x, y\}$, $def(\langle a, b \rangle) = def(\langle b, a \rangle) = \bot$, and $def(\langle a, a \rangle) = def(\langle b, b \rangle) = l$. Figure 2.5 shows both c and the corresponding c'.

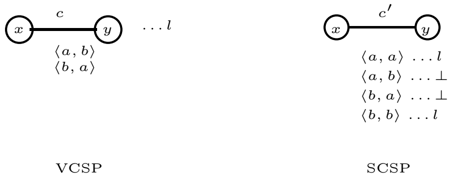


Fig. 2.5. From VCSP to SCSP

Again, we need to make sure that the structure we obtain via the definition above is indeed a semiring-based CSP. Then we will prove that it is equivalent to the given VCSP.

Theorem 2.3.7 (from valuation structure to c-semiring). *If we consider a valuation structure $\langle E, \circledast, \succ \rangle$ and the structure $S = \langle E, +, \circledast, \top, \bot \rangle$, where $+$ is such that $a + b = a$ iff $a \preccurlyeq b$ (obtained using the transformation in Definition 2.3.7), then S is a c-semiring.*

Proof. Since $\succ$ is total, $+$ is closed. Moreover, $+$ is commutative by definition, and associative because of the transitivity of the total order $\succ$. Furthermore, **0** is the unit element of $+$, since it is the top element of $\succ$. Finally, $+$ is idempotent because of the reflexivity of $\succ$, and **1** is the absorbing element of $+$ since **1** $= \bot$. Operation $\times$ of S coincides with $\circledast$. Thus it is closed, associative, and commutative, since $\circledast$ is so. Also, $\top$ is its absorbing element and $\bot$ is its identity (from corresponding properties of $\circledast$). The distributivity of $\circledast$ over $+$ can easily be proved. For example, consider $a, b, c \in E$, and assume $b \preccurlyeq c$. Then $a \circledast (b+c) = a \circledast b$ (by definition of $+$) $= (a \circledast b) + (a \circledast c)$ (by the definition of $+$ and the monotonicity of $\circledast$). The same reasoning applies to the case where $c \preccurlyeq b$.

Theorem 2.3.8 (equivalence between VCSP and SCSP). *Consider a VCSP problem P and the corresponding SCSP P'. Consider also an assignment t to all the variables of P. The value associated with such an assignment is $A = \mathcal{V}_P(t) = \circledast\{\varphi(c)$ for all $c \in C$ such that the projection of t over the variables of c violates $c\}$. Instead, the value associated with the same assignment in P' is $B = \circledast\{def_{c'}(t_{c'})$, for all constraints $c' = \langle def_{c'}, con_{c'} \rangle$ in C' and such that $t_{c'}$ is the projection of t over the variables of $c'\}$. Then, $A = B$.*

Proof. The values multiplied to produce A are as many as the constraints violated by t; instead, the values multiplied to produce B are as many as the constraints in C'. By construction, however, each tuple t_c involving the variables of a constraint $c \in C$ has been associated, in P', with a value that is either $\varphi(c)$ (if t_c violates c), or $\bot$ (if t_c satisfies c). Thus, the contribution of t_c to the value of B is important only if t_c violated c in P, because $\bot$ is the unit element for $\circledast$. Thus A and B are obtained by the same number of significant values. Now we have to show that such values are the same. But this is easy, since we have defined the translation in such a way that each tuple for the variables of c is associated with the value $\varphi(c)$ exactly when it violates c.

Normal Forms and Equivalences. Note that, while passing from an SCSP to a VCSP the number of constraints in general increases, in the opposite direction the number of constraints remains the same. This can also be seen in Example 2.3.2 and 2.3.3. This means that, in general, going from an SCSP P to a VCSP P' and then from the VCSP P' to the SCSP P'', we do not get $P = P''$. In fact, for each constraint c in P, P'' will have in general several constraints $c_1, \ldots, c_k$ over the same variables as c. It is easy to see, however, that $c_1 \otimes \cdots \otimes c_k = c$, and thus P and P'' associate the same value with each variable assignment.

Example 2.3.4. Figure 2.6 shows how to pass from an SCSP to the corresponding VCSP (this part is the same as in Example 1), and then again to the corresponding SCSP. Note that the starting SCSP and the final one are not the same. In fact, the latter has three constraints between variables x and y, while the former has only one constraint. One can see, however, that the combination of the three constraints yields the starting constraint.

Consider now the opposite cycle, that is, going from a VCSP P to an SCSP P' and then from P' to a VCSP P''. In this case, for each constraint c in P, P'' has two constraints: one is c itself, and the other one is a constraint with associated value $\perp$. This means that violating such a constraint has cost $\perp$, which, in other words, means that this constraint can be eliminated without changing the behaviour of P'' at all.

Example 2.3.5. Figure 2.7 shows how to pass from a VCSP to the corresponding SCSP (this part is the same as in Example 2), and then again to the corresponding VCSP. Note that the starting VCSP and the final one are not the same. In

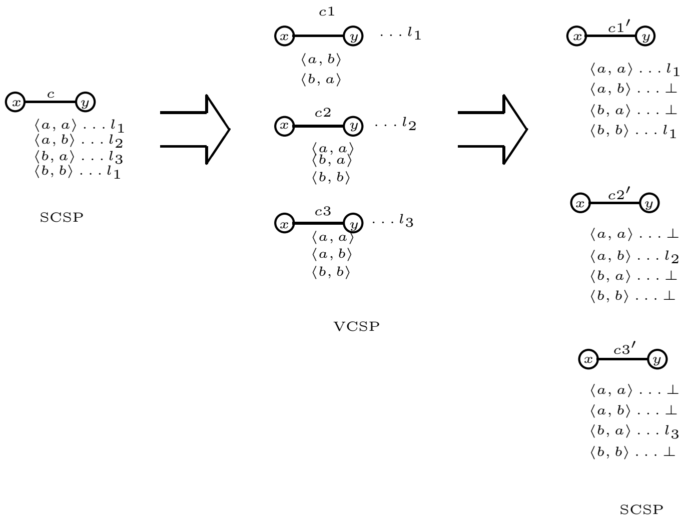


Fig. 2.6. From SCSP to VCSP and back to SCSP again

fact, the latter one has two constraints between variables x and y. One is the same as the one in the starting VCSP, while the other one has the value $\perp$ associated with it. This means that violating such constraint yields a cost of value $\perp$.

Let us now define normal forms for both SCSPs and VCSPs, as follows. For each VCSP P, its normal form is the VCSP $P' = nfv(P)$ which is obtained by deleting all constraints c such that $\varphi(c) = \perp$. It is easy to see that P and P' are equivalent.

Definition 2.3.8. *Consider $P = \langle V, D, C, S, \varphi\rangle$, a VCSP where $S = \langle E, \circledast, \succ\rangle$. Then P is said to be in normal form if there is no $c \in C$ such that $\varphi(c) = \perp$. If P in not in normal form, then it is possible to obtain a unique VCSP $P' = \langle V, D, C - \{c \in C \mid \varphi(c) = \perp\}, S, \varphi\rangle$, denoted by $P' = nfv(P)$, which is in normal form.*

Theorem 2.3.9 (normal form). *For any VCSP P, P and $nfv(P)$ are equivalent.*

Proof. The theorem follows from the fact that $\forall a, (\perp \circledast a) = a$ and from the definitions of $\mathcal{V}_P(A)$ and $\mathcal{V}_{P'}(A)$.

Also, for each SCSP P, its normal form is the SCSP $P' = nfs(P)$, which is obtained by combining all constraints involving the same set of variables. Again, this is an equivalent SCSP.

Definition 2.3.9. *Consider any SCSP P with constraints C over a constraint system $\langle S, D, V\rangle$, where S is a c-semiring $S = \langle A, +, \times, \mathbf{0}, \mathbf{1}\rangle$. Then, P is in normal form if, for each subset W of V, there is at most one constraint $c = \langle def, con\rangle \in C$ such that $con = W$. If P is not in normal form, then it is possible to obtain a unique SCSP P', as follows. For each $W \subseteq V$, consider the set $C_W \subseteq C$, which contains all the constraints involving W. Assume $C_W = \{c_1, \ldots, c_n\}$. Then, replace C_W with the single constraint $c = \bigotimes C_W$. P', denoted by $nfs(P)$, is in normal form.*

Theorem 2.3.10 (normal forms). *For any SCSP P, P and $nfs(P)$ are equivalent.*

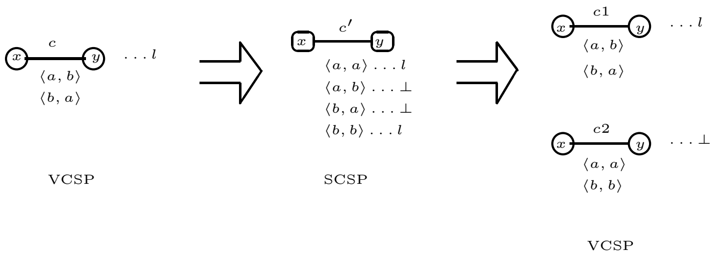


Fig. 2.7. From VCSP to SCSP, and to VCSP again

Proof. It follows from the associative property of $\times$.

Even though, as noted above, the transformation from an SCSP P to the corresponding VCSP P' and then again to the corresponding SCSP P'' does not necessarily yield $P = P''$, we will now prove that there is a strong relationship between P and P''. In particular, we will prove that the normal forms of P and P'' coincide. The same holds for the other cycle, where one passes from a VCSP to an SCSP and then to a VCSP again.

Theorem 2.3.11 (same normal form 1). *Given any SCSP P and the corresponding VCSP $P' = sv(P)$, consider the SCSP P'' corresponding to P', that is, $P'' = vs(P')$. Thus nfs(P) =nfs(P'').*

Proof. We will consider one constraint at a time. Take any constraint c of P. With the first transformation (to the VCSP P'), we get as many constraints as the different values associated with the tuples in c. Each of the constraints, say c_i, is such that $\varphi(c_i)$ is equal to one of such values, say l_i, and allows all tuples that do not have value l_i in c. With the second transformation (to the SCSP P''), for each of the c_i, we get a constraint c_i', where tuples which are allowed by c_i have value $\perp$ while the others have value l_i. Now, if we apply the normal form to P'', we combine all the constraints c_i', getting one constraint that is the same as c, since, given any tuple t, it is easy to see that t is forbidden by exactly one of the c_i. Thus, the combination of all c_i' will associate with t a value, which is the one associated with the unique c_i that does not allow t.

Theorem 2.3.12 (same normal form 2). *Given any VCSP problem P and the corresponding SCSP $P' = vs(P)$, consider the VCSP P'' corresponding to P', that is, $P'' = sv(P')$. Then we have that nfv(P) =nfv(P'').*

Proof. We will consider one constraint at a time. Take any constraint c in P, and assume that $\varphi(c) = l$ and that c allows the set of tuples T. With the first transformation (to the SCSP P'), we get a corresponding constraint c' where tuples in T have value $\perp$ and tuples not in T have value l. With the second transformation (to the VCSP P''), we get two constraints: c_1, with $\varphi(c_1) = \perp$, and c_2, with $\varphi(c_2) = l$ and which allows the tuples of c' with value $\perp$. It is easy to see that $c_2 = c$. Now, if we apply the normal form to both P and P'', which implies the deletion of all constraints with value $\perp$, we get exactly the same constraint. This reasoning applies even if the starting constraint has value $\perp$. In fact, in this case the first transformation will give us a constraint where all tuples have value $\perp$, and the second one gives us a constraint with value $\perp$, which will be deleted when obtaining the normal form.

The statements of the above two theorems can be summarized by the two diagrams represented in Figure 2.8. Note that in such diagrams each arrow represents one of the transformations defined above, and all problems in the same diagram are equivalent (by the theorems proved previously in this section).

$$\begin{tikzcd}
VCSP \arrow[r, "vs"] \arrow[d, "nfv"] & SCSP \arrow[d, "sv"] \\
VCSP & VCSP \arrow[l, "nfv"]
\end{tikzcd}$$

$$\begin{tikzcd}
SCSP \arrow[r, "sv"] \arrow[d, "nfs"] & VCSP \arrow[d, "vs"] \\
SCSP & SCSP \arrow[l, "nfs"]
\end{tikzcd}$$

Fig. 2.8. Diagrams representing the thesis of Theorem 2.3.11 and 2.3.11

2.3.8 N-dimensional C-semirings

Choosing an instance of the SCSP framework means specifying a particular c-semiring. This, as discussed above, induces a partial order, which can be interpreted as a (partial) guideline for choosing the "best" among different solutions. In many real-life situations, however, one guideline is not enough, since, for example, it could be necessary to reason with more than one objective in mind, and thus choose solutions which achieve a good compromise w.r.t. all such goals.

Consider, for example, a network of computers, where one would like to both minimize the total computing time (thus the cost) and also to maximize the work of the least used computers. Then, in our framework, we would need to consider two c-semirings, one for cost minimization (see Section 2.3.4 on weighted CSPs), and another one for work maximisation (see Section 2.3.2 on fuzzy CSPs). Then one could work first with one of these c-semirings and then with the other one, trying to combine the solutions which are the best for each of them. A much simpler approach, however, consists of combining the two c-semirings and only then work with the resulting structure. The nice property is that such a structure is a c-semiring itself, thus all the techniques and properties of the SCSP framework can be used for such a structure as well.

More precisely, the way to combine several c-semirings and getting another c-semiring simply consists of vectorizing the domains and operations of the combined c-semirings.

Definition 2.3.10 (composition of c-semirings). *Given n c-semirings $S_i = \langle A_i, +_i, \times_i, \mathbf{0}_i, \mathbf{1}_i \rangle$, for $i = 1, \ldots, n$, let us define the structure $Comp(S_1, \ldots, S_n) = \langle \langle A_1, \ldots, A_n \rangle, +, \times, \langle \mathbf{0_1}, \ldots, \mathbf{0_n} \rangle, \langle \mathbf{1_1} \cdots \mathbf{1_n} \rangle \rangle$. Given $\langle a_1, \ldots, a_n \rangle$ and $\langle b_1, \ldots, b_n \rangle$ such that $a_i, b_i \in A_i$ for $i = 1, \ldots, n$, $\langle a_1, \ldots, a_n \rangle + \langle b_1, \ldots, b_n \rangle = \langle a_1 +_1 b_1, \ldots, a_n +_n b_n \rangle$, and $\langle a_1, \ldots, a_n \rangle \times \langle b_1, \ldots, b_n \rangle = \langle a_1 \times_1 b_1, \ldots, a_n \times_n b_n \rangle$.*

We will now prove that by composing c-semirings we get another c-semiring. This means that all the properties of the framework defined in the previous sections hold.

Theorem 2.3.13 (a c-semiring composition is a c-semiring). *Given n c-semirings $S_i = \langle A_i, +_i, \times_i, \mathbf{0_i}, \mathbf{1_i} \rangle$, for $i = 1, \ldots, n$, the structure $Comp(S_1, \ldots, S_n)$ is a c-semiring.*

Proof. It is easy to see that all the properties required for being a c-semiring hold for $Comp(S_1, \ldots, S_n)$, since they directly follow from the corresponding properties of the component semirings S_i.

According to the definition of the ordering $\leq_S$ (in Section 2.1), such an ordering for $S = Comp(S_1, \ldots, S_n)$ is as follows. Given $\langle a_1, \ldots, a_n \rangle$ and $\langle b_1, \ldots, b_n \rangle$ such that $a_i, b_i \in A_i$ for $i = 1, \ldots, n$, we have $\langle a_1, \ldots, a_n \rangle \leq_S \langle b_1, \ldots, b_n \rangle$ if and only if $\langle a_1 +_1 b_1, \ldots, a_n +_n b_n \rangle = \langle b_1, \ldots, b_n \rangle$. Since the tuple elements are completely independent, $\leq_S$ is in general a partial order, even if each of the $\leq_{S_i}$ is a total order. This means (see Section 2.2) that the abstract solution of a problem over such a semiring in general contains an incomparable set of tuples, none of which has $blevel(P)$ as its associated value. Therefore, if one wants to reduce the number of "best" tuples (or to get just one), one has to specify some priorities among the orderings of the component c-semirings.

Notice that, although the c-semiring discussed in Section 2.3.5 may seem a composition of two c-semirings, as defined in Definition 2.3.10, this is not so, since the behaviour of each of the two operations on one of the two elements of each pair (we recall that that semiring is a set of pairs) is not independent from the behaviour of the same operation on the other element of the pair. Thus, it is not possible to define two independent operations (that is, $+_1$ and $+_2$, or $\times_1$ and $\times_2$), as needed by Definition 2.3.10. For example, operation $\underline{max}$ returns the maximum of the second elements of two pairs *only when* the first elements of the pairs are equal.

2.4 Conclusions

In this chapter we have introduced the SCSP framework and we have shown how it can be instantiated to represent several *non-crisp* constraint formalisms. We have described the different semirings that are needed to represent fuzziness, probabilities and optimizations. Furthermore, we have shown how to build new semirings by combining the existing ones.

The structure that we have defined in this framework will be the starting point for the results of the next chapters. In particular, in the next chapter we will review several solution techniques used in the classical frameworks, and we will prove their applicability to SCSPs, by looking only to the properties of the semiring operations.

3. Towards SCSPs Solutions

Overview

Local consistency algorithms, as usually used for classical CSPs, can be exploited in the SCSP framework as well, provided that certain conditions on the semiring operations are satisfied. We show how the SCSP framework can be used to model both old and new constraint solving and optimization schemes, thus allowing one to both formally justify many informally taken choices in existing schemes, and to prove that local consistency techniques can also be used in newly defined schemes. We generalize to soft constraints the approximation techniques usually used for local consistency in classical constraint satisfaction and programming. The theoretical results show that this is indeed possible without losing the fundamental properties of such techniques (and the experimental results (on partial arc-consistency) [111] show that this work can help develop more efficient implementations for logic-based languages working with soft constraints). Then, we consider dynamic programming-like algorithms, and we prove that these algorithms can always be applied to SCSPs, and have a linear time complexity when the given SCSPs can be provided with a parsing tree of bounded size. Finally, we provide several instances of SCSPs which show the generality and also the expressive power of the framework.

In classical CSPs, so-called local consistency techniques [104,105,138,139,140, 150,152] have been proven to be very effective when approximating the solution of a problem. In this chapter we study how to generalize these techniques to our framework, and we provide sufficient conditions over the semiring operations which assure that they can also be fruitfully applied to the considered scheme. Here, by "fruitfully applicable" we mean that 1) the algorithm terminates and 2) the resulting problem is equivalent to the given one and it does not depend on the nondeterministic choices made during the algorithm. In particular, such conditions rely mainly on having an idempotent operator (the $\times$ operator of the semiring).

The advantage of our framework, that we call SCSP (for Semiring-based CSP), is that one can hopefully see one's own constraint solving paradigm as an instance of SCSP over a certain semiring, and can inherit the results obtained for the general scheme. In particular, one can immediately see whether a local consistency technique can be applied. In fact, our sufficient conditions, which are related to the chosen semiring, guarantee that the above basic properties of local consistency hold. Therefore, if they are satisfied, local consistency can safely be applied. Otherwise, it means that we cannot be sure, in general, that local consistency will be meaningful in the chosen instance.

In this chapter we consider several known and new constraint solving frameworks, casting them as instances of SCSP, and we study the possibility of applying local consistency algorithms. In particular, we confirm that CSPs enjoy all the properties needed to use such algorithms, and that they can also be applied to FCSPs. Moreover, we consider probabilistic CSPs [96] and we see that local consistency algorithms might be meaningless for such a framework, as well as for weighted CSPs (since the above-cited sufficient conditions do not hold).

We also define a suitable notion of dynamic programming over SCSPs, and we prove that it can be used over any instance of the framework, regardless of the properties of the semiring. Furthermore, we show that when it is possible to provide a whole class of SCSPs with parsing trees of bounded size, then the SCSPs of the class can be solved in linear time.

To complete our study of the local consistency algorithms, we extend to the soft case some of the partial solving techniques that have proved useful for classical constraints. In fact, such techniques are widely used in current constraint programming systems like `clp(FD)` [72], Ilog Solver [163] and CHIP [5].

To do this, we define a class of Partial Local Consistency (PLC) algorithms, which perform less pruning than (complete) local consistency but can, nevertheless, be rather useful in reducing the search. Some instances of these algorithms are able to reduce the domains of the variables in constant time, and can work with a convenient representation of the domains of the variables. Using PLC together with labeling, the aim is to find the solutions of a problem with a better practical complexity bound. Moreover we study the termination condition in the SCSP framework of the general local consistency algorithms whose rules have to be only monotone and extensive (we eliminate the hypothesis of idempotency).

The chapter is organized as follows. Section 3.1 introduces the concept of local consistency for SCSPs and gives sufficient conditions for the applicability of the local consistency algorithms. Section 3.2 proves the applicability, or not, of the local consistency algorithms to several instances. Then in Section 3.3 we show some properties of the soft version of the Arc Consistency, and in Section 3.4 we describe some approximated algorithms of local consistency. Moreover Section 3.5 describes local consistency algorithms as a set of rules and gives sufficient conditions for the termination of the soft version of the GI schema [10, 11, 12]. Finally Section 3.6 describes a dynamic programming algorithm to solve SCSPs that can be applied to any instance of our framework, without any condition.

The firsts two section of this chapter appeared as part of [45, 47]. Moreover Section 3.4, 3.3 and 3.5, already appeared respectively in [54], [34] and [43, 44].

3.1 Soft Local Consistency

Computing any of the previously defined notions (the best level of consistency, the solution, and the abstract solution) is an NP-hard problem. Thus, it can be convenient in many cases to approximate such notions. In classical CSPs, this is done using the so-called local consistency techniques [104,105,132,152]. The main idea is to choose some subproblems in which to eliminate local inconsistency, and then iterate such elimination in all the chosen subproblems until stability. The most widely know local consistency algorithms are arc-consistency [138], where subproblems contain just one constraint, and path-consistency [150], where subproblems are triangles (that is, they are complete binary graphs over three variables). The concept of k-consistency [104], where subproblems contain all constraints among subsets of k chosen variables, generalizes them both. All subproblems are, however, of the same size (which is k). In general, one may imagine several algorithms where, instead, the considered subparts of the SCSP have different sizes. This generalization of the k-consistency algorithms were described in [152] for classical CSPs. Here we follow the same approach as in [152], but we extend the definitions and results presented there to the SCSP framework, and we show that all the properties still hold, provided that certain properties of the semiring are satisfied.

Applying a local consistency algorithm to a constraint problem means making explicit some implicit constraints, thus possibly discovering inconsistency at a local level. In classical CSPs, this is crucial, since local inconsistency implies global inconsistency. We will now show that such a property also holds for SCSPs.

Definition 3.1.1 (local inconsistency). *Consider an SCSP $P = \langle C, con \rangle$. Then we say that P is locally inconsistent if there exists $C' \subseteq C$ such that $blevel(C') = \mathbf{0}$.*

Theorem 3.1.1 (necessity of local consistency). *Consider an SCSP P which is locally inconsistent. Then it is not consistent.*

Proof. We have to prove that $blevel(P) = \mathbf{0}$. We know that P is locally inconsistent. That is, there exists $C' \subseteq C$ such that $blevel(C') = \mathbf{0}$. By Theorem 2.2.5, we have that $blevel(P) \leq_S blevel(C')$, thus $blevel(P) \leq_S \mathbf{0}$. Since $\mathbf{0}$ is the minimum in the ordering $\leq_S$, then we immediately have that $blevel(P)$ must be $\mathbf{0}$ as well.

In the SCSP framework, we can be even more precise, and relate the best level of consistency of the whole problem (or, equivalently, of the set of all its constraints) to that of its subproblems, even though such a level is not $\mathbf{0}$. In fact, it is possible to prove that if a problem is α-consistent, then all its subproblems are β-consistent, where $\alpha \leq_S \beta$.

Theorem 3.1.2 (local and global α-consistency). *Consider a set of constraints C over CS, and any subset C' of C. If C is α-consistent, then C' is β-consistent, with $\alpha \leq_S \beta$.*

Proof. The proof is similar to the previous one. If C is α-consistent, it means, by definition, that $blevel(C) = \alpha$. Now, if we take any subset C' of C, by Theorem 2.2.5 we have that $blevel(C) \leq_S blevel(C')$. Thus $\alpha \leq_S blevel(C')$.

In order to define the subproblems to be considered by a local consistency algorithm, we use the notion of local consistency rules. The application of such a rule consists of solving one of the chosen subproblems. To model this, we need the additional notion of *typed location*. Informally, a typed location is just a location l (as in ordinary store-based programming languages) which has a set of variables con as type, and thus can only be assigned a constraint $c = \langle def, con \rangle$ with the same type. In the following we assume to have a location for every set of variables, and thus we identify a location with its type. Note that at any stage of the computation the store will contain a constraint problem with only a finite number of constraints, and thus the relevant locations will always be in a finite number.

Definition 3.1.2 (typed location). *A typed location l is a set of variables.*

Definition 3.1.3 (value of a location). *Given a CSP $P = \langle C, con \rangle$, the value $[l]_P$ of the location l in P is defined as the constraint $\langle def, l \rangle \in C$ if it exists, as $\langle 1, l \rangle$ otherwise. Given n locations $l_1, \ldots, l_n$, the value $[\{l_1, \ldots, l_n\}]_P$ of this set of locations in P is defined instead as the set of constraints $\{[l_1]_P, \ldots, [l_n]_P\}$.*

Definition 3.1.4 (assignment). *An assignment is a pair $l := c$ where $c = \langle def, l \rangle$. Given a CSP $P = \langle C, con \rangle$, the result of the assignment $l := c$ is the problem $[l := c](P)$ defined as:*

$$[l := c](P) = \langle \{\langle def', con' \rangle \in C \mid con' \neq l\} \cup c, con \rangle.$$

Thus an assignment $l := c$ is seen as a function from constraint problems to constraint problems, which modifies a given problem by changing just one constraint, the one with type l. The change consists in replacing such a constraint with c. If there is no constraint of type l, then constraint c is added to the given problem.

Definition 3.1.5 (local consistency rule). *Given an SCSP $P = \langle C, con \rangle$, a local consistency rule r for P is defined as $r = l \leftarrow L$, where L is a set of locations, l is a location, and $l \notin L$.*

Applying r to P means assigning to location l the constraint obtained by solving the subproblem of P containing the constraints specified by the locations in $L \cup \{l\}$.

Definition 3.1.6 (rule application). *Given a local consistency rule $r = l \leftarrow L$ and a constraint problem P, the result of applying r to P is*

$$[l \leftarrow L](P) = [l := Sol(\langle [L \cup \{l\}]_P, l \rangle)](P).$$

Since a rule application is defined as a function from problems to problems, the application of a sequence S of rules to a problem is easily provided by function composition. Thus we have that $[r_1; S](P) = [S]([r_1](P))$.

Notice for example that $[l \leftarrow \emptyset](P) = P$.

In other words, a *local consistency rule*, written also r_l^L, is a function r_l^L which, taken any problem P and , returns $r_l^L(P) = [l := Sol(\langle [L]_P, l\rangle)](P)$.

Theorem 3.1.3 (equivalence for rules). *Given an SCSP P and a rule r for P, we have that $P \equiv [r](P)$ if $\times$ is idempotent.*

Proof. Assume that $P = \langle C, con\rangle$, and let $r = l \leftarrow L$. Then we have $P' = [l \leftarrow L](P) = [l := Sol(\langle [L \cup \{l\}]_P, l\rangle)](P) = \langle \{\langle def', con'\rangle \in C \mid con' \neq l\} \cup Sol(\langle [L \cup \{l\}]_P,\ l\rangle), con\rangle$. Let now $C(r) = [L \cup \{l\}]_P$, and $C' = C - C(r)$. Then P contains the constraints in the set $C' \cup C(r)$, while P' contains the constraints in $C' \cup (C(r) - c) \cup ((\bigotimes C(r)) \Downarrow_l)$, where $c = \langle def, l\rangle = [l]_P$. In fact, by definition of solution, $Sol(\langle [L \cup \{l\}]_P, l\rangle)$ can be written as $(\bigotimes C(r)) \Downarrow_l$. Since the set C' is present in both P and P', we will not consider it, and we will instead prove that the constraint $c_{pre} = \bigotimes C(r)$ coincides with $c_{post} = (\bigotimes(C(r) - \{c\})) \otimes ((\bigotimes C(r)) \Downarrow_l)$. In fact, if this is so, then $P \equiv P'$.

First of all, it is easy to see that c_{pre} and c_{post} have the same type $\bigcup(L \cup \{l\})$. Let us now consider the definitions of such two constraints. Let us set $c_i = [l_i]_P = \langle def_i, l_i\rangle$ for all $l_i \in L$, and assume $L = \{l_1, \ldots, l_n\}$. Thus, $C(r) = \{c, c_1 \ldots, c_n\}$. We have that, taken any tuple t of length $\mid \bigcup(L \cup \{l\}) \mid$, $def_{pre}(t) = def(t \downarrow_l) \times (\Pi_i def_i(t \downarrow_{l_i}))$, while $def_{post}(t) = (\Pi_i def_i(t \downarrow_{l_i})) \times \Sigma_{t' \mid t'\downarrow_l = t\downarrow_l}(def(t' \downarrow_l) \times (\Pi_i def_i(t' \downarrow_{l_i})))$. Now, since the sum is done over all t' such that $t' \downarrow_l = t \downarrow_l$, we have that $def(t' \downarrow_l) = def(t \downarrow_l)$. Thus $def(t' \downarrow_l)$ can be taken out from the sum since it appears the same in each factor. Thus we have $def_{post}(t) = (\Pi_i def_i(t \downarrow_{l_i})) \times def(t \downarrow_l) \times \Sigma_{t' \mid t'\downarrow_l = t\downarrow_l}(\Pi_i def_i(t' \downarrow_{l_i}))$. Consider now $\Sigma_{t' \mid t'\downarrow_l = t\downarrow_l}(\Pi_i def_i(t' \downarrow_{l_i}))$: one of the t' such that $t' \downarrow_l = t \downarrow_l$ must be equal to t. Thus the sum can be written also as $(\Pi_i def_i(t \downarrow_{l_i}) + \Sigma_{t' \mid t'\downarrow_l = t\downarrow_l, t' \neq t}(\Pi_i def_i(t' \downarrow_{l_i}))$. Thus we have $def_{post}(t) = (\Pi_i def_i(t \downarrow_{l_i})) \times def(t \downarrow_l) \times (\Pi_i def_i(t \downarrow_{l_i}) + \Sigma_{t' \mid t'\downarrow_l = t\downarrow_l, t' \neq t}(\Pi_i def_i(t' \downarrow_{l_i})))$.

Let us now consider any two elements a and b of the semiring. Then it is easy to see that $a \times (a + b) = a$. In fact, by intensivity of $\times$ (see Theorem 2.1.3), we have that $a \times c \leq_S a$ for any c. Thus, if we choose $c = a + b$, $a \times (a + b) \leq_S a$. On the other hand, since $a + b$ is the lub of a and b, we have that $a + b \geq_S a$. Thus, by monotonicity of $\times$, we have $a \times (a + b) \geq_S a \times a$. Now, since we assume that $\times$ is idempotent, we have that $a \times a = a$, thus $a \times (a + b) \geq_S a$. Therefore $a \times (a + b) = a$. This result can be used in our formula for $def_{post}(t)$, by setting $a = (\Pi_i def_i(t \downarrow_{l_i}))$ and $b = \Sigma_{t' \mid t'\downarrow_l = t\downarrow_l, t' \neq t}(\Pi_i def_i(t' \downarrow_{l_i}))$. In fact, by replacing $a \times (a + b)$ with a, we get $def_{post}(t) = (\Pi_i def_i(t \downarrow_{l_i})) \times def(t \downarrow_l)$, which coincides with $def_{pre}(t)$.

Definition 3.1.7 (stable problem). *Given an SCSP P and a set R of local consistency rules for P, P is said to be stable w.r.t. R if, for each $r \in R$, $[r](P) = P$.*

A local consistency algorithm consists of the application of several rules to the same problem, until stability of the problem w.r.t. all the rules. At each step of the algorithm, only one rule is applied. Thus the rules will be applied in a certain order, which we call a strategy.

Definition 3.1.8 (strategy). *Given a set R of local consistency rules for an SCSP, a strategy for R is an infinite sequence $S \in R^{\infty}$. A strategy S is fair if each rule of R occurs in S infinitely often.*

Definition 3.1.9 (local consistency algorithm). *Given an SCSP P, a set R of local consistency rules for P, and a fair strategy S for R, a local consistency algorithm $lc(P, R, S)$ applies to P the rules in R in the order given by S. Thus, if the strategy is $S = s_1 s_2 s_3 \ldots$, the resulting problem is*

$$P' = [s_1; s_2; s_3; \ldots](P)$$

The algorithm stops when the current SCSP is stable w.r.t. R.

Note that this formulation of local consistency algorithms extends the usual one for k-consistency algorithms [104], which can be seen as local consistency algorithms where all rules in R are of the form $l \leftarrow L$, where $L = \{l_1, \ldots, l_n\}$ and $\mid \bigcup_{i=1,\ldots,n} l_i \mid = \mid l \mid = k - 1$. That is, exactly $k - 1$ variables are involved in the locations[1] in L.

Arc-consistency [29, 122] (AC) is an instance of local consistency where one propagates only the domains of the variables. With our notations, this type of algorithm deals only with rules of the form $\{x\} \leftarrow \{\{x, y_1, \ldots, y_n\}, \{y_1\}, \ldots, \{y_n\}\}$. In fact, an arc-consistency rule considers a constraint, say over variables $x, y_1, \ldots, y_n$, and all unary constraints over these variables, and combines all these constraints to get some information (by projecting) over one of the variables, say x[2]. Let us call AC this set of rules. In the following, we will write $ac(P)$ to denote $lc(P, AC, S)$.

When a local consistency algorithm terminates[3], the result is a new problem that has the graph structure of the initial one plus, possibly, new arcs representing newly introduced constraints (we recall that constraints posing no restrictions on the involved variables are usually not represented in the graph structure), but where the definition of some of the constraints has been changed. More precisely, assume $R = \{r_1, \ldots, r_n\}$, and $r_i = l_i \leftarrow L_i$ for all $i = 1, \ldots, n$. Then the algorithm uses, among others, n typed locations l_i of type con_i. Assume also that each of such typed locations has value def_i when the algorithm terminates. Consider also the set of constraints $C(R)$ that have type con_i. That is, $C(R) = \{\langle def, con \rangle$ such that $con = con_i$ for some i between 1 and $n\}$. Informally, these are the constraints that may have been modified (via the typed location mechanism) by the algorithm. Then, if the initial SCSP is $P = \langle C, con \rangle$, the resulting SCSP is P' $= lc(P, R, S) = \langle C', con \rangle$, where $C' = C - C(R) \cup (\bigcup_{i=1,\ldots,n} \langle def_i, con_i \rangle)$.

[1] We recall that locations are just sets of variables.

[2] Actually, this is a generalized form of arc-consistency, since originally arc-consistency was defined for binary constraints only [138].

[3] We will consider the termination issue later.

In classical CSPs, any local consistency algorithm enjoys some important properties. We now will study these same properties in the SCSP framework, and point out the corresponding properties of the semiring operations that are sufficient for them to hold. The desired properties are as follows:

1. any local consistency algorithm returns a problem which is equivalent to the given one;
2. any local consistency algorithm terminates in a finite number of steps;
3. the strategy, if fair, used in a local consistency algorithm does not influence the resulting problem.

Theorem 3.1.4 (equivalence). *Consider an SCSP P and the problem $P' = lc(P, R, S)$. Then $P \equiv P'$ if the multiplicative operation of the semiring ($\times$) is idempotent.*

Proof. By Definition 3.1.9, a local consistency algorithm is just a sequence of applications of local consistency rules. Since by Theorem 3.1.3 we know that each rule application returns an equivalent problem, by induction we can conclude that the problem obtained at the end of a local consistency algorithm is equivalent to the given one.

Theorem 3.1.5 (termination). *Consider any SCSP $P = \langle C, con \rangle$ over the constraint system $CS = \langle S, D, V \rangle$ and the set $AD = \bigcup_{\langle def, con \rangle \in C} R(def)$, where $R(def) = \{a \mid \exists t \text{ with } def(t) = a\}$. Then the application of a local consistency algorithm to P terminates in a finite number of steps if AD is contained in a set I which is finite and such that $+$ and $\times$ are I-closed.*

Proof. Each step of a local consistency algorithm may change the definition of one constraint by assigning a different value to some of its tuples. Such value is strictly worse (in terms of $\leq_S$) since $\times$ is intensive. Moreover, it can be a value which is not in AD but in $I - AD$. If the state of the computation consists of the definitions of all constraints, then at each step we get a strictly worse state (in terms of $\sqsubseteq_S$). The sequence of such computation states, until stability, has finite length, since, by assumption, I is finite and thus the value associated with each tuple of each constraint may be changed at most $|I|$ times.

An interesting special case of the above theorem occurs when the chosen semiring has a finite carrier set A. In fact, in that case the hypotheses of the theorem hold with $I = A$.

Corollary 3.1.1. *Consider any SCSP P over the constraint system $CS = \langle S, D, V \rangle$, where $S = \langle A, +, \times, \mathbf{0}, \mathbf{1} \rangle$, and A is finite. Then the application of a local consistency algorithm to P terminates in a finite number of steps.*

Proof. Easily holds with $I = AD$ by Theorem 3.1.5.

We will now prove that, if $\times$ is idempotent, no matter which strategy is used during a local consistency algorithm, the result is always the same problem.

Theorem 3.1.6 (order-independence). *Consider an SCSP P and two different applications of the same local consistency algorithm to P, producing respectively the SCSPs $P' = lc(P, R, S)$ and $P'' = (P, R, S')$. Then $P' = P''$ if the multiplicative operation of the semiring ($\times$) is idempotent.*

Proof. Each step of the local consistency algorithm, which applies one of the local consistency rules in R, say r, were defined as the application of a function $[r]$ that takes an SCSP P and returns another problem $P' = [r](P)$, and that may change the definition of the constraint connecting the variables in l (if the applied rule is $r = l \leftarrow L$). Thus the whole algorithm may be seen as the repetitive application of the functions $[r]$, for all $r \in R$, until no more change can be done. If we can prove that each $[r]$ is a closure operator [80], then classical results on chaotic iteration [76] allow us to conclude that the problem resulting from the whole algorithm does not depend on the order in which such functions are applied. Closure operators are just functions which are idempotent, monotone, and intensive. We will now prove that each $[r]$ enjoys such properties. We remind that $[r]$ just combines a constraint with the combination of other constraints.

If $\times$ is idempotent, then $[r]$ is idempotent as well, that is, $[r]([r](P)) = [r](P)$ for any P. In fact, combining the same constraint more than once does not change the problem by idempotency of $\times$. Also, the monotonicity of $\times$ (see Theorem 2.1.2) implies that of $[r]$, that is, $P \sqsubseteq P'$ implies $[r](P) \sqsubseteq [r](P')$. Note that, although $\sqsubseteq$ has been defined only between constraints, here we use it among constraint problems, meaning that such a relationship holds among each pair of corresponding constraints in P and P'. Finally, the intensivity of $\times$ implies that $[r]$ is intensive as well, that is, $[r](P) \sqsubseteq P$. In fact, P and $[r](P)$ just differ in one constraint, which, in $[r](P)$, is just the corresponding constraint of P combined with some other constraints.

Note that the above theorems require that $\times$ is idempotent for a local consistency algorithm to be meaningful. There is, however, no requirement over the nature of $\leq_S$. More precisely, $\leq_S$ can be partial. This means that the semiring operations $+$ and $\times$ can be different from *max* and *min* respectively (see note at the end of Section 2.1).

Note also that, by definition of rule application, constraint definitions are changed by a local consistency algorithm in a very specific way. In fact, even in the general case in which the ordering $\leq_S$ is partial, the new values assigned to tuples are always smaller than or equal to the old ones in the partial order. In other words, local consistency algorithms do not "jump" in the partially ordered structure from one value to another one that is unrelated to the first one. More precisely, the following theorem holds.

Theorem 3.1.7 (local consistency and partial orders). *Given an SCSP P, consider any value v assigned to a tuple in a constraint of such problem. Also, given any set R of local consistency rules and strategy S, consider $P' = lc(P, R, S)$, and the value v' assigned to the same tuple of the same constraint in P'. Then we have that $v' \leq_S v$.*

Proof. By definition of rule application (see Definition 3.1.6) the formula defining the new value associated to a tuple of a constraint can be assimilated to $(v \times a) + (v \times b)$, where v is the old value for that tuple and a and b are combinations of values for tuples of other constraints in the graph of the rule. Now, we have $(v \times a) + (v \times b) = v + (a \times b)$ by distributivity of $+$ over $\times$ (see Theorem 2.1.5). Also, $(v \times c) \leq_S v$ for any c by intensivity of $\times$ (see Theorem 2.1.3), thus $v + (a \times b) \leq_S v$.

One could imagine other generalizations with the same desired properties as the ones proven above (that is, termination, equivalence, and order-independence). For example, one could design an algorithm which avoids (or compensates) the effect of cumulation coming from a non-idempotent $\times$. Or also, one could generalize the equivalence property, by allowing the algorithm to return a non-equivalent problem (but, nonetheless, being in a certain relationship with the original problem). Finally, the order-independence property states that any order is fine, as far as the resulting problem is concerned. Another desirable property, related to this, could be the existence of an ordering that makes the algorithm belong to a specific complexity class.

3.2 Applying Local Consistency to the Instances

3.2.1 Classical CSPs

Let us now consider the application of a local consistency algorithm to CSPs. As predictable, we will show that all the classical properties hold. First, $\wedge$ is idempotent. Thus, by Theorem 3.1.4, the problem returned by a local consistency algorithm is equivalent to the given one, and from Theorem 3.1.6, the used strategy does not matter. Also, since the domain of the semiring is finite, by Corollary 3.1.1 any local consistency algorithm terminates in a finite number of steps.

3.2.2 Fuzzy CSPs

Let us now consider the properties that hold in the FCSP framework. The multiplicative operation (that is, *min*) is idempotent. Thus, local consistency algorithms on FCSPs do not change the solution of the given problem (by Theorem 3.1.4), and do not care about the strategy (by Theorem 3.1.6). Moreover, *min* is AD-closed for any finite subset AD of [0,1]. Thus, by Theorem 3.1.5, any local consistency algorithm on FCSPs terminates.

Thus FCSPs, although providing a significant extension to classical CSPs, can exploit the same kind of algorithms. Their complexity will, of course, be different due to the presence of more than two levels of preference. It can be proven. however, that if the actually used levels of preference are p, then the complexity of a local consistency algorithm is just $O(p)$ times greater than that of the corresponding algorithm over CSPs (as also discussed above and in [91]).

An implementation of arc-consistency, suitably adapted to be used over fuzzy CSPs, is given in [173]. No formal properties of its behavior, however, are proven there. Thus our result can be seen as a formal justification of [173].

3.2.3 Probabilistic CSPs

The multiplicative operation of S_{prob} (that is, $\times$) is not idempotent. Thus the result of Theorem 3.1.4 cannot be applied to Prob-CSPs. That is, by applying a local consistency algorithm one is not guaranteed to obtain an equivalent problem. Theorem 3.1.6 cannot be applied as well. Therefore the strategy used by a local consistency algorithm may matter. Also, $\times$ is not closed on any finite superset of any subset of $[0, 1]$. Hence the result of Theorem 3.1.5 cannot be applied to Prob-CSPS. That is, we cannot be sure that a local consistency algorithm terminates.

As a result of these observations, local consistency algorithms should make no sense in the Prob-CSP framework. As a result, we are not sure that their application has the desired properties (termination, equivalence, and order-independence). However, the fact that we are dealing with a c-semiring implies that, at least, we can apply Theorem 3.1.2: if a Prob-CSP problem has a tuple with probability α to be a solution of the real problem, then any subproblem has a tuple with probability at least α to be a solution of a subproblem of the real problem. This can be fruitfully used when searching for the best solution. In fact, if one employs a branch-and-bound search algorithm, and in one branch we find a partial instantiation with probability smaller than α, then we can be sure that such a branch will never lead to a global instantiation with probability α. Thus, if a global instantiation with such a probability has already been found, we can cut this branch. Similar (and possibly better) versions of branch-and-bound for non-standard CSPs may be found in [108, 174].

3.2.4 Weighted CSPs

The multiplicative operation of S_{WCSP} (that is, $+$) is not idempotent. Consequently the results of Section 3.1 on local consistency algorithms cannot be applied to WCSPs. As in Prob-CSPs, however, the fact that we are dealing with a c-semiring implies that, at least, we can apply Theorem 3.1.2: if a WCSP problem has a best solution with cost α, then the best solution of any subproblem has a cost smaller than α. This can be fruitfully used when searching for the best solution in a branch-and-bound search algorithm.

Note that the same properties hold also for the semirings $\langle \mathcal{Q}^+, min, +, +\infty, 0\rangle$ and $\langle \mathcal{Z}^+, min, +, +\infty, 0\rangle$ (which can be proved to be c-semirings).

3.2.5 Egalitarianism and Utilitarianism

One can show that S_{ue} is a c-semiring. However, since it uses a combination of the operations of the semirings S_{FCSP} and S_{WCSP}, and since the multiplicative operation of S_{WCSP} is not idempotent, also the multiplicative operation of S_{ue}, that is, $\underline{min}$, is not idempotent. This means that we cannot guarantee that local consistency algorithms can be used meaningfully in this instance of the framework.

Note that also the opposite choice (that is, first perform a max-sum and then a max-min) can be made, by using a similar c-semiring. Due however to the same reasoning as above, again the multiplicative operation would not be idempotent, thus we cannot be sure that the local consistency techniques possess the properties of Section 3.1.

3.2.6 Set-Based SCSPs

The set-based SCSPs are represented by the c-semiring

$$S_{set} = \langle \wp(A), \bigcup, \bigcap, \emptyset, A \rangle$$

where A is any set. It is easy to see that S_{set} is a c-semiring. Also, in this case the order $\leq_{S_{set}}$ reduces to set inclusion (in fact, $a \leq b$ iff $a \cup b = b$), and, therefore, it is partial in general. Furthermore, $\times$ is $\bigcap$ in this case, and thus it is idempotent. Therefore, the local consistency algorithms possess all the properties stated in Section 3.1 and, therefore, can be applied. We recall that when a local consistency algorithm is applied to an SCSP with a partially ordered semiring, its application changes the constraints such that the new constraints are smaller than the old ones in the ordering $\sqsubseteq_S$.

3.3 About Arc-Consistency in SCSPs

In this section we study the properties and possible advantages of the extension to SCSPs of the most successful of the local consistency techniques, called *arc-consistency* [138, 193], although we claim that all our results can be extended to higher levels of consistency. What arc-consistency (AC) does over a CSP is considering each pair of variables and checking whether there are values for one of these variables, which are inconsistent with all the values for the other one. If so, such inconsistent values are deleted from the CSP, since they cannot participate in any solution. Semiring-based arc-consistency (SAC) works similarly over SCSPs, considering, however, also the semiring values, and the final effect is not a deletion, but a worsening of the semiring value assigned to some domain element of a variable.

Here we study the relationship between AC and SAC over SCSPs, showing that they discover the same inconsistencies. That is, the domain elements whose semiring value gets replaced by the worst value (that is, the minimum of the semiring) are exactly those elements which are deleted by AC in the corresponding CSP, obtained by the given SCSP by forgetting the semiring values. Thus sometimes it may be reasonable to apply just a classical AC instead of SAC, since AC is more efficient, although less informative. This can lead to an original methodology where, instead of preprocessing a given SCSP via SAC and then solving it, we can first obtain the corresponding CSP, then preprocess this CSP via AC (or some other form of local consistency), and then bring back to the initial SCSP the information obtained by AC over the inconsistencies. In particular, if AC discovered an inconsistency, we can safely assign to such an element the worst semiring value. Note that we could also make this method more flexible, by replacing the given SCSP not with a CSP but with a simpler SCSP, that is, an SCSP over a simpler semiring. In this way, there could be a whole range of possible simplifications, in the line of [169], and the results of SAC over the simpler problem could be brought back to the given problem via established techniques like abstract interpretation [79]. In Chapter 4 we will show how the usual Galois connection of abstract interpretation is used to pass from an SCSP to an "easier" one.

Moreover, we consider SCSPs with visible and hidden variables, and we show that, if such SCSPs are SAC, then some hidden variables can be removed without changing the set of solutions and associated semiring values. In particular, we show that all hidden variables of degree 1 can be removed. The reason behind this result is that, by achieving SAC, all the information provided by the variables of degree 1 is put into the other (connected) variables. If such variables are not hidden, then we can still gain from the application of SAC, because they can be instantiated without the need for any backtracking. Moreover, since in a tree all leaves have degree 1, and the other variables get degree 1 once the leaves are removed, tree-shaped SCSPs can be solved without backtracking, provided that we follow an instantiation order that visits the tree top-down. For SCSPs which are not tree-shaped, we can still use this result, on the parts which are tree-shaped. Therefore, such SCSPs can be solved by searching (possibly with backtracking) over the part with cycles, and then by continuing the instantiation process over the tree-shaped parts without backtracking.

These results are extensions of similar results already existing for the CSP case [105, 166]; however, here they are particularly interesting both because of the generality of the SCSP framework, and also because SCSPs usually express constraint optimization, and not just satisfaction, problems.

The Section is organized as follows. Subsection 3.3.1 compares the power of classical arc-consistency with that of semiring-based arc-consistency. Then, Subsection 3.3.2 studies the possibility of removing hidden variables in an arc-consistent SCSP, subsection 3.3.3 shows that tree-shaped SCSPs can be solved easily, and Section 3.3.4 extends this result to SCSPs with cycles with bounded size.

3.3.1 Classical Arc-Consistency vs. Semiring-Based One

Consider an SCSP $P = \langle C, con \rangle$. Let us now construct a corresponding CSP $cut(P)$ by simply forgetting the semiring values associated with all the tuples in all constraints of P, except for those values which are the minimum of the semiring. That is, if a tuple has a semiring value different from the minimum we say that such tuple satisfies the constraint in $cut(P)$, otherwise we say that it does not satisfy it. For example, the CSP in Figure 3.1 is the CSP corresponding to the FCSP of Figure 3.2 according to function *cl*.

Then, let us apply SAC over P and AC over $cut(P)$. As a result of that, some element in the domain of some variables in P will get a smaller semiring value, possibly the minimum of the semiring, and some domain elements of some variables in $cut(P)$ will be eliminated. Then, the main result is the following one:

Theorem 3.3.1 (SAC vs. AC). *In an SAC-consistent SCSP P, if the order $\leq_S$ induced by the $+$ operation is total, then the domain elements which will get the minimum of the semiring in P by applying an SAC algorithm are exactly those that will be eliminated in $cut(P)$ by applying an AC algorithm.*

In fact, consider what happens during the SAC algorithm: for each pair of variables x and y, to see which value has to be given to the domain element t_x for x, that now has value a_x, we have to consider also all the pairs t_{xy} that extend t_x over y. Let us assume there are n of them, and let us call them $t^1_{xy}, \ldots, t^n_{xy}$. Assume also that each t^i_{xy} is given the semiring value a^i_{xy}. Finally, consider all domain elements of y, say $t^1_y, \ldots, t^n_y$, and assume that they now have the semiring values $a^1_y, \ldots, a^n_y$. What we have to do now to perform SAC over this constraint is to compute the following: $\sum_{i=1,\ldots,n}(a_x \times a^i_{xy} \times a^i_y)$. This is the semiring value to be associated with t_x. That is, if t_x has a different semiring value, the problem is not SAC-consistent.

Now the question is: when does this value coincide with the minimum of the semiring? In general we don't know. We know, however, that if the $\leq_S$ order is total, then $a \times b$ is always equal to either a or b, for any a, b in the semiring, and the same is for $a + b$. In fact, the glb and the lub of two elements in a total order is always one of the two elements.

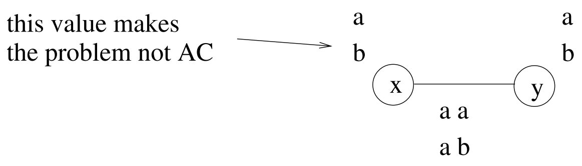


Fig. 3.1. A CSP which is not AC

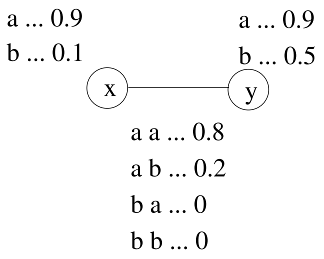


Fig. 3.2. A fuzzy CSP

Thus, the only way that the above formula coincides with the minimum of the semiring is such a minimum appears somewhere in the formula. That is, either a_x, or a^i_{xy} for some i, or a^i_y for some i, is already the minimum of the semiring. In other words, if none of these values coincides with the minimum of the semiring, then we can be sure that the result of the above formula does not coincide with the minimum of the semiring.

Now, by the way we have constructed $cut(P)$, tuples that have the minimum of the semiring do not appear in $cut(P)$. That is, they do not satisfy the corresponding constraint. Therefore, by the way AC works, they will create a local inconsistency and they will cause the deletion of an element from the domain of a variable. More precisely, in $cut(P)$, to see whether element t_x of the domain of x has to be eliminated, we have to look at all pairs t_{xy} that extend t_x over y and that satisfy the constraint between x and y, and for each of such pairs we have to see whether its second element (that is, its projection over y) is in the domain of y or not. If all such elements are not in the domain of y, then t_x has to be eliminated by the domain of x. This means that for all pairs t_{xy} that extend t_x over y, either t_{xy} does not satisfy the constraint between x and y, or the projection of t_{xy} onto y is not in the domain of y. But if this is so, then, by the way $cut(P)$ was constructed, the element t_x in P will get the minimum of the semiring by SAC. The opposite is also easily provable: if t_x gets the minimum of the semiring in P via SAC, then it will be eliminated in $cut(P)$ via AC.

Figure 3.3 shows the FCSP obtained by applying SAC onto the FCSP of Figure 3.2 and the CSP obtained by applying AC onto the CSP of Figure 3.1.

Why is this result interesting? Because applying AC over $cut(P)$ is more efficient, both in space and in time, than applying SAC over P. In fact, domains

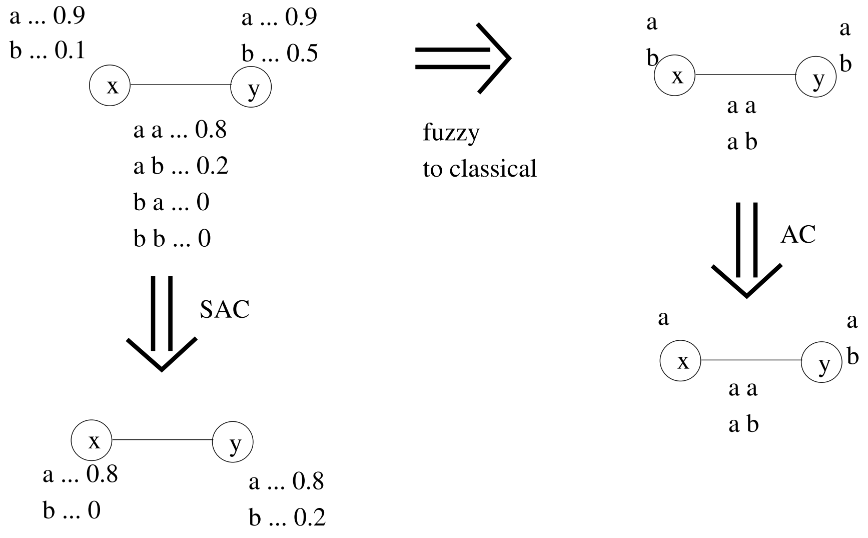


Fig. 3.3. An SAC-consistent FCSP and the corresponding AC-consistent CSP

could have a more compact representation, and each step of the algorithm could deal with a smaller set of tuples (only those that have semiring value *true*).

Therefore, one could use AC instead of SAC if he/she is interested only in discovering local inconsistencies. Of course some information will be lost, because the semiring values that are present in the problem after AC and which are different from the minimum are not as accurate, but in some cases this could be a reasonable trade-off between pruning power and space/time efficiency.

In fact, this approach could also be seen as the extreme case of a flexible abstraction technique, where, given an SCSP over a certain semiring S, we replace SAC over such a semiring with SAC over a simpler semiring, not necessarily S_{CSP} (as we proposed in this section) which is the simplest semiring of all. More details in this direction are given in Chapter 4 and in [33,35,36] where the usual Galois connection of abstract interpretation is used to pass from an SCSP to an "easier" one.

3.3.2 Removing Hidden Variables in SCSPs

An SCSP, as defined above, is a pair $\langle C, con\rangle$, where con is a subset of the variables, and specifies the variables of interest as far as solutions are concerned. Variables in con can then be called the *visible* variables, while all the others are *hidden*.

Since the solution set of such problems is defined as $(\otimes C) \Downarrow_{con}$, this means that a solution is an assignment of the visible variables, with an associated semiring value, such that it can be extended to the hidden variables in a way that the overall assignment (to all the variables) has the same semiring value. Since, however, we do not really care to know the values for the hidden variables, but just that there is a way to instantiate them, in some cases some of these variables can be found to be redundant. This means that their elimination would not change the solutions of the overall problem. This, of course, can be very convenient in terms of the search for one or the best solution, because the depth of the search tree coincides with the number of variables. In particular, this section shows the following:

Theorem 3.3.2 (hidden variable removal). *If we have an SCSP which is SAC-consistent, and if the order $\leq_S$ is total, then all hidden variables with degree 1 (or 0) are redundant and thus can be eliminated, without changing the set of solution-value pairs for the problem.*

This is an extension to SCSP of a result which has been shown already for CSPs [166]. For example, Figure 3.4 shows the case of a CSP where, assuming that the problem is AC, variable v is redundant. Here is, however, more interesting, since we are dealing with optimization and not satisfaction problems.

Let us now prove the statement of the above theorem. Consider an SCSP that is SAC-consistent, and assume that there is a hidden variable, say z, with degree 1. Figure 3.5 shows an example of such an SCSP.

What we have to prove is that, given an instantiation of x and y, which satisfies the constraints between them, for example $\langle x = a, y = a\rangle$, which has

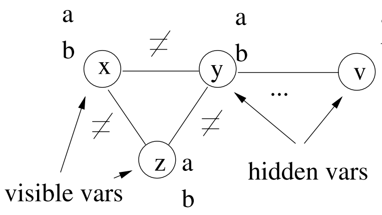


Fig. 3.4. Redundant hidden variables in CSPs

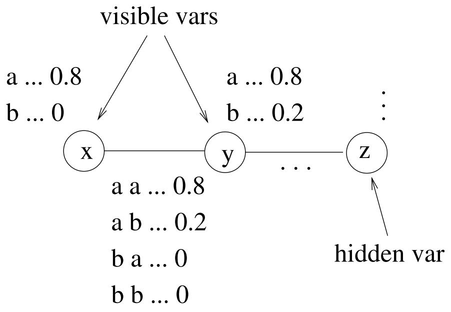


Fig. 3.5. Redundant hidden variables in SCSPs

value 0.8, there is an extension to z, say d, such that $\langle x = a, y = a, z = d\rangle$ has value 0.8 and all other extensions have a worse value. In fact, if this is the case, then we can safely delete z without changing the solution set.

More generally, let us consider an instantiation of the visible variables with a corresponding semiring value, which is given by combining all constraints involving subsets of the visible variables: say it is $\langle d_1, \ldots, d_n\rangle$ (there are n visible variables) with semiring value v. To show that z is redundant, we have to show that such a tuple can be extended to z without making its semiring value worse. In other words, there is a value for z, say d_z, such that the tuple $\langle d_1, \ldots, d_n, d_z\rangle$ has semiring value v.

To compute the semiring value for this tuple, once we have the semiring value v for the shorter tuple, we need to multiply (by using the $\times$ operator of the semiring) v by the semiring value given by c_{xz} (we assume that x is the i-th visible variable) to $\langle x = d_i, z = v\rangle$, say v_{xz}, and also by the semiring value given by c_z to $z = v$, say v_z. Notice that if z is indeed connected to x, this is the only constraint connected to z, otherwise we will just consider a null constraint which assigns the maximum of the semiring to each tuple. Thus, we get $v \times v_{xz} \times v_z$, and we have to prove that this coincides with v. Figure 3.6 shows the three components of this new value for the example in Figure 3.5, to be compared to v.

Now, by the extensivity of $\times$ (that is, $a \times b \leq a$ for any a, b), we know that $(v \times v_{xz} \times v_z) \leq v$. We will now prove that this disequality is in fact an equality, that is, $(v \times v_{xz} \times v_z) = v$. We know that any SAC algorithm enforces the formula

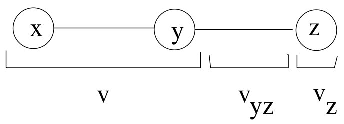


Fig. 3.6. New semiring value for the extended tuple

$c_x = (\otimes\{c_y, c_{xy}, c_x\}) \Downarrow_x$ for each pair of variables x and y. Thus, this also applies to the pair of variables x and z of our example. Therefore, the semiring values associated with elements of the domain of x, such as d_i, have been computed during the SAC algorithm by also considering c_{xz} and c_z. In other words, the information that we are now adding to v, that is, $v_{xz} \times v_z$, is already in v, because of SAC. Of course there are many ways to instantiate z, but if the order $\leq_S$ is total, one of them, that is, the best one, has been inserted in v. By choosing that value for z, by the idempotency of the $\times$ operation, which has to be assumed if we want to be sure that local consistency has the desired properties, we get that $v = v \times v_{xz} \times v_z$. Therefore, there exists a way to extend $\langle d_1, \ldots, d_n \rangle$ to z such that its semiring value is not changed.

Notice also that once some variables are eliminated the degree of other variables may be reduced and thus satisfy the redundancy condition. Thus these other variables can be eliminated as well, until no variable has degree 1.

Notice also that it is not necessary to achieve SAC all over the problem to eliminate some hidden variable, but that it is enough to achieve it on the neighborhood (just two variables) of the variable to eliminate. Also, in such a neighborhood, SAC could be achieved just in one direction, because we have to make sure that a solution involving the visible variables can be extended to the hidden variables, but we do not care about the opposite direction.

Again, we claim that the result of this section can easily be generalized to a higher level of consistency, allowing the elimination of variables of higher degree. That is, the generalization should say that, if an SCSP is k-consistent, then all hidden variables with degree up to $k - 1$ are redundant.

3.3.3 Tree-Shaped SCSPs

The result of the previous section says that, if we have an SCSP with a total order over the semiring values, and if the SCSP is SAC-consistent, then any tuple involving the visible variables can be extended to any hidden variable with degree 1 without changing the semiring value of the tuple.

Consider now the same situation, except that the variable with degree 1 is not hidden, but visible. Still, the extension works just the same, and the only thing that is different here is that we have to constructively find that extension, and we cannot be satisfied with just knowing that it exists. Let us see what we have to do to find the (or *a*) domain element for z (the variable with degree 1), which does not change the semiring value of the given tuple. Since the difference between the semiring value of the new tuple and that of the old one depends only on the constraint connecting z, say c_{xz}, and on the domain of z, say c_z, we just have to look at all pairs of values for x and z such that $x = d_i$ (that is, the value that x has in the short tuple), and for each of such pairs we have to multiply the value given by c_{xz} by that given by c_z. Now, the best of such values is the one to choose. That is, we choose $z = d_z$ if $c_{xz}(d_i, d_z) \times c_z(d_z)$ is the best among all $c_{xz}(d_i, d'_z) \times c_z(d'_z)$ for all d'_z other values for z (where given a constraint $c = \langle def, con\rangle$, $c(d)$ means the application of the definition function to the domain value $d4$, that is the value $def(d)$).

This operation is linear in the domain size, since for each element of the domain of z, say d_z, we just need to compute $c_{xz}(d_i, d_z) \times c_z(d_z)$. More important, if the tuple we started with (that is, $\langle d_1, \ldots, d_n\rangle$) was the best one for the first n variables, then the new tuple, $\langle d_1, \ldots, d_n, d_z\rangle$ is still the best one for the n initial variables plus z. In fact, it is impossible to find a tuple which has a better semiring value for such $n+1$ variables. Otherwise the assumption that the initial tuple was the best one for the n variables will be contradicted.

This discussion leads to the following theorem, which again is an extension of a result already known for classical CSPs [105]:

Theorem 3.3.3 (linear time for trees). *If we have an SCSP with a total order over the semiring values, and such an SCSP is SAC-consistent and tree-shaped, then the optimal solution can be found in time linear in the number of its variables.*

In fact, in a tree-shaped SCSP, all variables have either degree 1 (the leaves) or can get degree 1 when the current leaves are eliminated. Therefore, the extension result just discussed can be applied at every step of the search, provided that we choose an instantiation order that visits the tree from top to bottom. Figure 3.7 shows a tree-shaped SCSP and a possible top-down instantiation order.

That is, at every step during the search, we have a partial solution involving only the previous variables, and we have to extend it to the next variable. By what we said before, this extension can be done without changing the semiring value of the solution, and also without the need to come back to check whether there is a better solution. Therefore, a complete best solution can be found without backtracking.

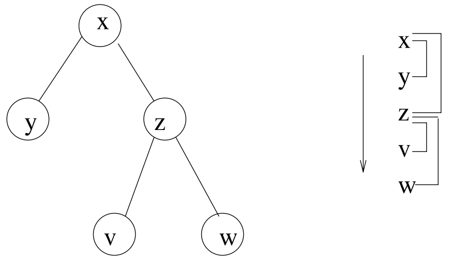


Fig. 3.7. A tree-shaped SCSP and a top-down instantiation order

An interesting remark is that the semiring value for the best solution can be found without even finding the best solution. In fact, since each step does not change the semiring value of the current incomplete solution, the best value of a complete solution coincides with the best value for the first variable in the order. In fact, by choosing the best value for the first variable, we know that we can complete the solution maintaining that value.

Proposition 3.3.1 (best value). *Given an SCSP with a total order over the semiring values, which is SAC-consistent and tree-shaped, the semiring value for the optimal solutions is the best value among those assigned to the domain elements of the first variable in a top-down instantiation order for this problem.*

Even for this result we can easily see that SAC is not really necessary, we just need a directional version of SAC that goes bottom-up in the tree.

3.3.4 Cycle-Cutsets in SCSPs

By stretching the discussion of the previous section even more, we can consider SCSPs, which in general do not have a tree shape, where, however, we identify in some way the part of the SCSP that creates the cycles. In such problems, we can first find the best solution for the difficult part (the one with cycles), and after that we can complete the solution on the rest of the variables, which is tree-shaped. Therefore, the overall search for the best solution will possibly backtrack over the first part of the problem, to find the best solution for that part, but then will not backtrack at all on the rest of the problem. Figure 3.8 shows an SCSP with a cycle over variables x, y, and z, and then a tree shape over the rest of the problem (not completely drawn), and Figure 3.9 shows the search tree for this SCSP.

In classical constraint solving this technique is called *cycle-cutset* [85], since the first part of the problem (that possibly involves some backtracking) is identified by a subset of the variables whose elimination allows to remove all cycles of the problem. Here we can use the same technique and get similar results, which, however, as noted before, deal with optimization instead of satisfaction. Therefore, here we can say something about the time complexity to find not

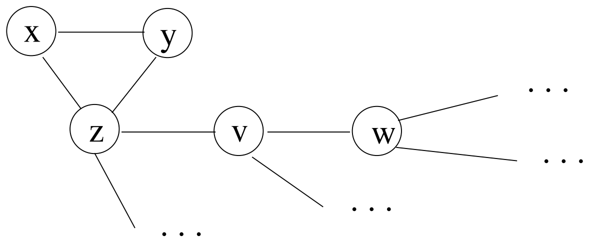


Fig. 3.8. An SCSP with a cycle over x, y, z and a tree shape over the other variables

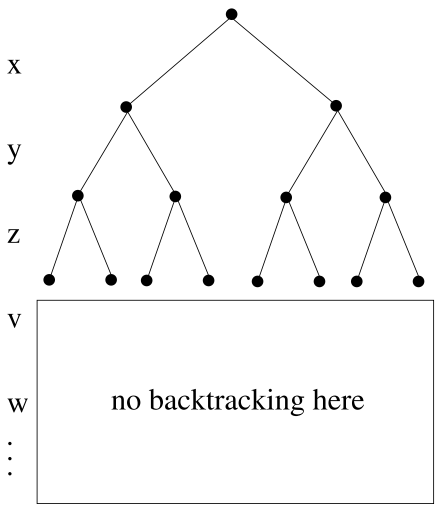


Fig. 3.9. The search tree for the SCSP of Figure 3.8

a solution, but the *best solution*. Moreover, as in the previous section, we have an additional result which allows us to know the semiring value associated with the best solution *before* we actually compute the solution itself. In fact, once all variables in the cycle-cutset have been instantiated and we have found the best solution for them, the semiring value for this partial solution will coincide with the semiring value for the best complete solution.

More interesting results can be obtained if one applies the SAC algorithm not only before the search but also *during* the search, after each instantiation. In fact, instantiating a variable is like eliminating that variable for the problem to be solved, and thus new variables could become of degree 1 with respect to the set considered at the beginning of the search. By maintaining SAC at each step during the search, we could, therefore, exploit the presence of this variables to dynamically augment the set of variables over which the search does not need any backtracking. Again, this idea has been generated by recent results on classical CSPs [170], which, however, we claim can be extended to SCSPs as well.

3.4 Labeling and Partial Local Consistency for SCSPs

The aim of this section is to extend the local consistency algorithms, defined in the previous sections for SCSPs, in order to capture some approximated algorithms which have been proven useful for classical constraints, so much so that they are widely used in current constraint programming systems like `clp(FD)` [72], Ilog Solver [163] and CHIP [5].

The SCSP framework comes with a class of local consistency (LC) algorithms, which are useful to perform a simplification of the problem, cutting away some tuples of domain values that are shown to be useless w.r.t. the solutions of the problem.

This scheme is in theory very useful but experiments show that even the simplest non-trivial LC algorithm (which corresponds to Arc-Consistency [29, 122] extended to SCSPs) has a heavy complexity for most applications. Moreover, one has to apply it several times if used during the labeling phase. Hence, the reduction of its complexity is a crucial point.

To go in this direction, we define a class of Partial Local Consistency (PLC) algorithms, which perform less pruning than (complete) local consistency but can nevertheless be rather useful in reducing the search. Some instances of these algorithms are able to reduce the domains of the variables in constant time, and can work with a convenient representation of the domains of the variables. Using PLC together with labeling, the aim is to find the solutions of a problem with a better practical complexity bound.

From this point of view, our work is related to the one of K. Apt [9]. In fact, he studied a class of filtering functions for classical CSPs, which can express both local consistency and also less pruning notions.

Summarizing, here are our results that appear also in [34]:

1. the extension of the definitions and properties of the labeling procedure from classical CSPs to SCSPs;
2. a general scheme for approximating local consistency in SCSPs, and properties that are sufficient for the correctness of the approximations;
3. some implementations issues in the particular case of arc-consistency;
4. experimental results for partial arc-consistency.

The rest of the section is organized as follows. In Section 3.4.1 we introduce some properties about labelings, we describe their interaction with usual filtering algorithms, and we define the generalization of the labeling procedure to SCSPs. Then, we introduce the notion of rule approximations and study the correctness of such rules in Section 3.4.2 and we discuss some implementations issues in Section 3.4.3.

Another operation.. Besides combination and projection, we need, in this section, another operator, which we will call the *disjunction* operator. Given two constraints $c_1 = \langle def_1, con_1\rangle$ and $c_2 = \langle def_2, con_2\rangle$ over CS, their *disjunction* $c_1 \oplus c_2$ is the constraint $c = \langle def, con\rangle$ with $con = con_1 \cup con_2$ and $def(t) = def_1(t \downarrow^{con}_{con_1}) + def_2(t \downarrow^{con}_{con_2})$.

The informal meaning of the disjunction operator is to give more possibility to (or to enhance the level of preference of) certain instantiations. If we consider the semiring *Bool* which corresponds to classical CSPs, the meaning of having the disjunction of two constraints C_1 and C_2 is to have the possibility of choosing not only the tuples permitted by C_1 but also those permitted by C_2. Figure 3.10 shows an example using the semiring Fuzzy, where, we recall, the operation $+$ is the maximum.

Using the properties of $\times$ and $+$, it is easy to prove that: $\oplus$ is associative, commutative and idempotent; $\otimes$ distributes over $\oplus$; $\oplus$ is monotone over $\sqsubseteq_S$. Moreover, if $\times$ is idempotent: $\oplus$ distributes over $\otimes$.

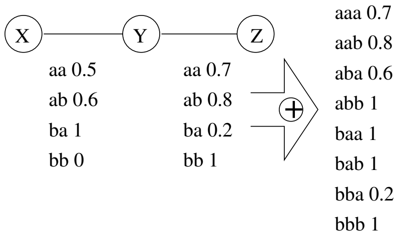


Fig. 3.10. Disjunction example

Let us also give some new definition that will be useful in the following of this section.

Definition 3.4.1 (constraint type). *Given a constraint $c = \langle def, con\rangle$, we call $type(c)$ the set con of the variables of c. Given a constraint set C, $type(C) = \{type(c)/c \in C\}$ and $V(C) = \bigcup_{c\in C} type(c)$ is the set of all the variables appearing in the constraints of C.*

3.4.1 Labeling in SCSPs

Local consistency algorithms, when applied to an SCSP P, can reduce the search space to find its solutions. Solving the resulting problem P', however, is still NP-hard. This scenario is similar to what happens in the classical CSP case, where, after applying a local consistency algorithm, the solution is found by replacing P' with a set of subproblems $\{P_1, \dots, P_n\}$ where some of the variables have been instantiated such that P' is "equivalent" to $\{P_1, \dots, P_n\}$, in the sense that the set of solutions of P' coincides with the union of the sets of solutions of $P_1, \dots, P_n$. Usually, problems P_i are obtained by choosing a variable, say x, and instantiating it to its i-th domain value. Then, the local consistency (usually arc-consistency) algorithm is applied again [121]. By doing this, one hopes to detect the inconsistency of some of the subproblems by detecting their local inconsistency (via the local consistency algorithm). When all the variables of the initial problem are instantiated, arc-consistency becomes complete in the sense that if the problem is not consistent then it is not arc-consistent. Therefore, at that point arc-consistency implies global consistency.

Let us consider, for example, the following problem:

$$X \in [1,3], Y \in [1,2], Z \in [1,2], X \neq Y, Y \neq Z, Z \neq X.$$

Figure 3.11 shows the interaction between AC and the labeling procedure, since each node of the search tree (except the root) shows the CSP obtained after the application of an arc-consistency algorithm. In this particular case we have two solutions (the dashed line indicates the second solution found by the labeling).

We will now study the properties of the labeling procedure when applied to SCSPs.

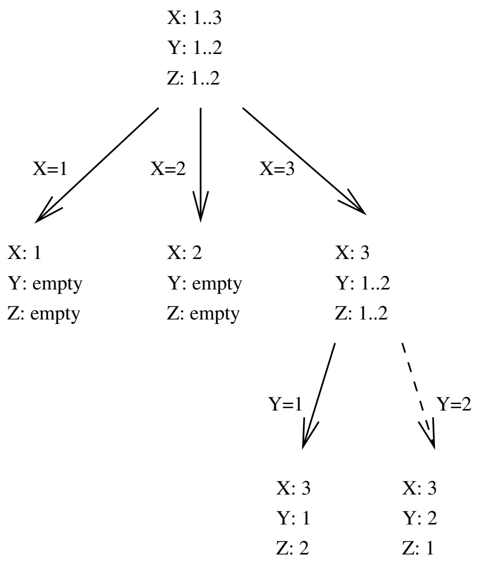


Fig. 3.11. AC and labeling

In the following we assume to work with a constraint system $CS = \langle S, D, V \rangle$, where $S = \langle A, +, \times, \mathbf{0}, \mathbf{1} \rangle$.

Given a set $\mathcal{P} = \{P_1, \cdots, P_n\}$ of SCSPs, the *solution of* $\mathcal{P}$ is the constraint defined as $Sol(\mathcal{P}) = Sol(P_1) \oplus \cdots \oplus Sol(P_n)$.

Given $x \in V$, $d \in D$ and $v \in A$, we call $c_{x,d,v}$ the unary constraint $\langle def, \{x\} \rangle$ where $def(d) = v$ and $def(d') = \mathbf{0}$ if $d' \neq d$. In practice, $c_{x,d,v}$ instantiates x to d with semiring value v. We call this constraint a *simple instantiation constraint.*

Given a total order $\prec$ on V, let $W = \{x_1, \ldots, x_n\}$ be a subset of V, and assume that $x_1 \prec \cdots \prec x_n$. Given an n-tuple of domain values $d = \langle d_1, \ldots, d_n \rangle$ and an n-tuple of semiring values $v = \langle v_1, \ldots, v_n \rangle$, we define: $I_W^{d,v} = \{c_{x_i,d_i,v_i} \mid i \in [1,n]\}$. We write I_W^d for $I_W^{d,\mathbf{1}}$, and we call it a W*-instantiation set.* In practice, I_W^d gives the best semiring value ($\mathbf{1}$) to the assignment of d_i to variable x_i. Instead, $I_W^{d,v}$ gives semiring value v_i to the assignment of d_i to variable x_i.

Given a problem $P = \langle C, con \rangle$, let W be a subset of $V(C)$ and t be a $|W|$-tuple. We call a W*-labeling* of the problem P the problem $P_W^t = \langle C \cup I_W^t, con \rangle$. In words, a W*-labeling* of P adds to P some additional constraints which identify a specific partial labeling for P (partial because it involves only the variables in W). A *complete labeling* of P is thus written $P_{V(C)}^t$. Also, $\mathcal{L}_W(P) = \{P_W^t \mid t \in D^{|W|}\}$ is the set of all W-labelings of P. We call it *the* W-labeling of P.

Figure 3.12 presents a fuzzy SCSP and shows its solution, while Figure 3.13 shows the $\{x, y\}$-labeling corresponding to the original problem and the set of possible partial solutions, whose disjunction gives rise to the complete solution. Notice that the problem set solution coincides with the solution of the problem in Figure 3.12.

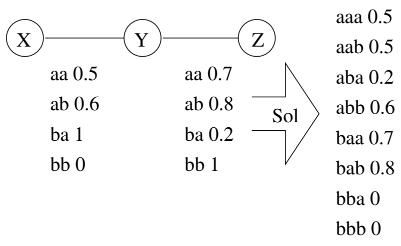


Fig. 3.12. A fuzzy CSP and its solution

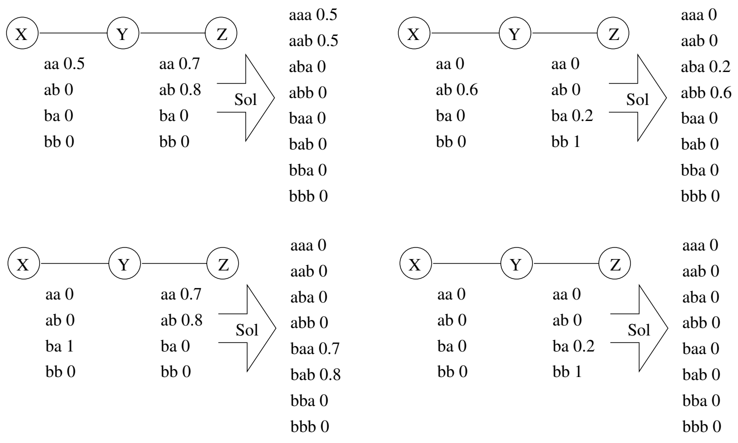


Fig. 3.13. The set of the $\{x, y\}$-labelings corresponding to the problem

This is true also in general; therefore, we can compute the solution of an SCSP by computing the disjunction of the solutions associated with the labeling. This is important, since it means that we can decompose an SCSP and solve it by solving its sub-problems. The following theorem formalizes this informal statement.

Theorem 3.4.1 (disjunction correctness). *Given an SCSP $P = \langle C, con \rangle$, a subset W of $V(C)$, and a set $\{c_1, \ldots, c_n\}$ of constraints over W such that $\bigoplus_i c_i = \langle \mathbf{1}, W \rangle$, we have that $Sol(P) = \bigoplus_i Sol(\langle C \cup \{c_i\}, con \rangle)$. Moreover, $P \equiv \mathcal{L}_W(P)$.*

Proof. By the assumptions, and by the distributivity of $\times$ over $+$, for any tuple t of domain values for all the variables, we get:

We note V the set $V(C)$ and p the size of W. We assume $\bigotimes C = \langle def_C, V \rangle$, $c_i = \langle def_i, W \rangle$ and $\bigoplus_i Sol(\langle C \cup c_i, con \rangle) = \langle def, con \rangle$. We have:

$$def(t) = \sum_i \sum_{\{t' | t' \downarrow^{V \cup W}_{con} = t\}} def_i(t' \downarrow^{V \cup W}_{W}) \times def_C(t' \downarrow^{V \cup W}_{V})$$

Since $W \subseteq V$, we have:

$$def(t) = \sum_i \sum_{\{t' | t' \downarrow^V_{con} = t\}} def_i(t' \downarrow^V_W) \times def_C(t')$$

Now, by distributivity of $\times$ over $+$, we get:

$$def(t) = \sum_{\{t' | t' \downarrow^V_{con} = t\}} def_C(t') \times (\sum_i def_i(t' \downarrow^V_W))$$

Since $\bigoplus_i c_i = \langle \mathbf{1}, W \rangle$, we have $\sum_i def_i(t' \downarrow^V_W) = \mathbf{1}$. Thus:

$$def(t) = \sum_{\{t' | t' \downarrow^V_{con} = t\}} def_C(t')$$

that coincides with $Sol(P)$.

This result allows us to compute the solution of an SCSP using the labeling procedure. Moreover, if all the variables are instantiated, i.e. when the labeling $\mathfrak{L}$ is complete ($W = V(P)$), then the values found for the variables contain all the informations needed to compute (very easily) the exact solution of the problem:

Theorem 3.4.2 (solution of a complete labeling). *We suppose given a problem $P = \langle C, con \rangle$ where $C = \{c_i | i \in [1, n]\}$ and $\forall i \in [1, n], c_i = \langle def_i, con_i \rangle$. Let $P^t_{V(C)} = \langle C \cup I^t_{V(C)}, con \rangle$ be a complete labeling of P. Then $Sol(P^t_{V(C)}) = \langle def, con \rangle$ with:*

- $def(t \downarrow^{V(C)}_{con}) = \prod_{i \in [1,n]} def_i(t \downarrow^{V(C)}_{con_i})$,
- $def(t') = \mathbf{0}$ *if* $t' \neq t \downarrow^{V(C)}_{con}$.

Moreover, $blevel(P^t_{V(C)}) = \prod_{i \in [1,n]} def_i(t \downarrow^{V(C)}_{con_i})$.

Proof. We denote by p the size of $V(C)$. By definition of a complete labeling problem:

$$Sol(P^t_{V(C)}) = ((\bigotimes_{j \in [1,n]} c_j) \otimes (\bigotimes_{k \in [1,p]} c_{x_k, t_k, \mathbf{1}})) \Downarrow_{con}$$

By associativity and idempotency of $\otimes$, we have:

$$Sol(P^t_{V(C)}) = (\bigotimes_{j \in [1,n]} (c_j \otimes (\bigotimes_{k \in [1,p]} c_{x_k, t_k, \mathbf{1}})) \Downarrow_{con}$$

Let $\langle def', V(C) \rangle = c_j \otimes (\bigotimes_{k \in [1,p]} c_{x_k, t_k, \mathbf{1}})$. Then:

- $def'(t) = def_j(t \downarrow^{V(C)}_{con_j})$,
- $def'(t') = \mathbf{0}$ if $t' \neq t$.

The statement of the theorem easily follows.

As in the CSP case ([122]), it is sufficient to maintain AC during the labeling procedure to completely solve the problem (after a complete labeling has been obtained).

Theorem 3.4.3 (correctness of AC). *Given an SCSP $P = \langle C, con \rangle$ and a total order $\prec$ between the variables, assume that $V(C) = \{x_1, \ldots, x_p\}$ with $x_1 \prec \cdots \prec x_p$, and let $P^t_{V(C)} = \langle C \cup I^t_{V(C)}, con \rangle$ be a complete labeling of P. Also, let $P' = ac(P^t_{V(C)})$ be the arc-consistent problem obtained starting from $P^t_{V(C)}$. Then:*

- *for each i in $[1, p]$, the value $[\{x_i\}]_{P'}$ of location x_i in problem P' is a simple instantiation constraint c_{x_i,t_i,v_i},*
- *$blevel(P^t_{V(C)}) = \prod_{i \in [1,p]} v_i$.*

Proof. It is easy to see that the value $[\{x_i\}]_{P'}$ of location x_i in problem P' is still a simple instantiation constraint of the form c_{x_i,t_i,v_i}. Let us now prove that $blevel(P^t_{V(C)}) = \prod_{i \in [1,p]} v_i$. One should first notice that $\forall i \in [1, p], blevel(P^t_{V(C)}) \leq_S v_i$. Thus $blevel(P^t_{V(C)}) \leq \prod_{i \in [1,p]} v_i$. We will now prove that $blevel(P^t_{V(C)}) \geq \prod_{i \in [1,p]} v_i$. With the notations of Theorem 3.4.2, we have to prove that $\prod_{j \in [1,n]} def_j(t \downarrow^{V(C)}_{con_j}) \geq \prod_{i \in [1,p]} v_i$. It is easy to see that $\forall i \in [1, p], \forall j \mid x_i \in con_j, v_i \leq def_j(t \downarrow^{V(C)}_{con_j})$. Thus, we have $\forall i \in [1, p], v_i \leq \prod_{j \mid x_i \in con_j} def_j(t \downarrow^{V(C)}_{con_j})$. The statement of the theorem follows from the idempotency of $\times$.

3.4.2 Partial Local Consistency

In this section we introduce the notion of *approximate local consistency algorithms*, and we give some sufficient conditions for such algorithms to terminate and be correct.

Notice that a local consistency algorithm is already an approximation in itself. In fact, because of its incompleteness, it approximates a complete solution algorithm. Usually, however, local consistency algorithms (as we have defined them in this section and as they are usually used in practice) replace a constraint by the *solution* of a set of other constraints. Here we want to approximate this aspect of such algorithms, by allowing them to replace a constraint by *a superset of the solution* of a set of other constraints. To preserve the nice properties of local consistency, however, we must assure that such a superset is not too big, so as not to introduce any additional solution of the whole problem.

Approximated Rules. The idea of an approximation function is to replace the real computation (that is, the solution of the subproblem defined by a rule) with a simpler one. The following definition states that a “correct” and “complete” approximation function should not enlarge the domains and also it should not lose any solution. Given a rule $r = l \leftarrow L$, an *approximation function* ϕ for r is a function from $type^{-1}(L) \times type^{-1}(l)$ to $type^{-1}(l)$ such that:

- for all constraint set C of type L, for all constraint c of type l, $Sol(\langle C \cup \{c\}, l \rangle) \sqsubseteq_S \phi(C, c) \sqsubseteq_S c$,

– for all constraint sets $\{c_1, \ldots, c_p\}$, $\{c'_1, \ldots, c'_p\}$ of type L, for all constraints c, c' of type l, $(\forall i \in [1,p], c_i \sqsubseteq_S c'_i, c \sqsubseteq_S c') \Rightarrow (\phi(\{c_1, \ldots, c_p\}, c) \sqsubseteq_S \phi(\{c'_1, \ldots, c'_p\}, c'))$.

As trivial examples, one can consider the approximation function that does no approximation (let us call it ϕ_{best}) and also the one that does no domain reduction (ϕ_{worst}).

Example 3.4.1. Let $r = l \leftarrow L$ be a rule. Let ϕ_{best} such that $\forall C \in type^{-1}(L), \forall c \in type^{-1}(l)$, $\phi_{best}(C, c) = Sol(\langle C \cup \{c\}, l\rangle)$. It is easy to see that ϕ_{best} is an approximation function for r. It is the "best" approximation function (in fact, there are no rule that compute an approximation closer to the solution).

Example 3.4.2. Let $r = l \leftarrow L$ be a rule. Let ϕ_{worst} such that $\forall C \in type^{-1}(L)$, $\forall c \in type^{-1}(l)$, $\phi_{worst}(C, c) = c$, It is easy to see that ϕ_{worst} is an approximation function for r. It is the "worst" approximation function (in fact, the rule does not change any information over the constraints).

Given a rule $r = l \leftarrow L$ and an approximation function ϕ for r, the *approximation of rule r by function ϕ* is defined as $r_\phi = l \leftarrow_\phi L$. Given an approximated rule $r_\phi = l \leftarrow_\phi L$, the application of r_ϕ to problem P is defined by $[l \leftarrow_\phi L](P) = [l := \phi([L]_P, [l]_P)](P)$. That is, we apply to P the approximated rules instead of the original ones.

This notion of rule approximation can be used to modify the definition of a constraint without changing the solution of the problem. In fact, as the following theorem states, applying the approximated rule leads to a problem which has better values than the problem obtained by applying the rule. Thus, we cannot lose any solution, since we obtain an SCSP that is "between" the original problem and the problem obtained by using the non-approximated rules, which, we know by definition of local consistency, do not change the solution set.

Theorem 3.4.4. $[l \leftarrow L](P) \sqsubseteq_P [l \leftarrow_\phi L](P) \sqsubseteq P$.

Proof. By definition of rule application $[l \leftarrow L](P) = [l := Sol(\langle [L \cup \{l\}]_P, l\rangle)](P)$ and $[l \leftarrow_\phi L](P) = [l := \phi([L]_P, [l]_P)](P)$. Now, since the problem $[l \leftarrow L](P)$ and the problem $[l \leftarrow_\phi L](P)$ differ only for the constraint over the variable set l and since, by definition of an approximation function, $Sol(\langle C \cup \{c\}, l\rangle) \sqsubseteq_S \phi(C, c)$, the statement obviously holds.

It is easy to verify that ϕ_{best} computes the same problem as the rule itself while ϕ_{worst} does not change the problem.

Theorem 3.4.5 (equivalence for rules). *Given a problem P and an approximated rule r_ϕ, $P \equiv [r_\phi](P)$.*

Proof. From $[r](P) \sqsubseteq_P [r_\phi](P) \sqsubseteq_P P$ and $[r](P) \equiv P$, it follows that $P \equiv [r_\phi](P)$.

Example 3.4.3. Let $r = l \leftarrow L$ be a rule. We have $[l \leftarrow_{\phi_{best}} L](P) = [l \leftarrow L](P)$ and $[l \leftarrow_{\phi_{worst}} L](P) = P$.

We remind the reader that in order to fruitfully apply the local consistency scheme the $\times$ operation of the semiring has to be idempotent ([45, 47]) so in every statement regarding local consistency in the following we assume to have an idempotent $\times$ operation.

Since the application of an *approximated rule* does not change the solution of the problem, we can define a correct and complete semiring-based *partial local consistency* algorithm, by applying, following a given strategy, several approximated rules to a given SCSP.

Given a set of rules R, an *approximation scheme* Φ for R associates to each rule $r = l \leftarrow L$ of R an approximation $\Phi(r) = l \leftarrow_\phi L$. Given a problem P, a set R of local consistency rules, an approximation scheme Φ for R, and a fair strategy S for R, a local consistency algorithm applies to P the Φ-approximations of the rules in R in the order given by S. Thus, if the strategy is $S = s_1 s_2 s_3 \ldots$, the resulting problem is $P' = [\Phi(s_1); \Phi(s_2); \Phi(s_3); \ldots](P)$. The algorithm stops when the current problem is stable w.r.t. R. In that case, we denote by $plc(P, R, S, \Phi)$ the resulting problem.

Proposition 3.4.1 (equivalence). $P \equiv plc(P, R, S, \Phi)$.

Proof. Trivially follows from Theorem 3.4.5 and by transitivity of $\equiv$.

Theorem 3.4.6 (termination). *Consider an SCSP P over a finite semiring. Then the application of a partial local consistency algorithm over P terminates in a finite number of steps.*

Proof. Trivially follows from the intensivity of $\times$ and finiteness of A.

Theorem 3.4.7 (order-independence). *Consider an SCSP P and two different applications of the same partial local consistency algorithm to P, producing respectively the problem $P' = plc(P, R, S, \Phi)$ and $P'' = plc(P, R, S', \Phi)$. Then, $P' = P''$. Thus, we will write $P' = plc(P, R, \Phi)$.*

Proof. It is easy to verify that $[\Phi(r)]$ is intensive and monotone, which is sufficient to apply the chaotic iteration theorem [76].

Partial Arc-Consistency. Given an SCSP P and an approximation scheme Φ for AC, we will write $pac(P, \Phi)$ instead of $plc(P, AC, \Phi)$.

Let ϕ be an approximation function for $r = \{x\} \leftarrow \{\{x, y_1, \ldots, y_n\}, \{y_1\}, \ldots, \{y_n\}\}$ (which, we recall, is the scheme for the AC rules). Let $Y = \{y_1, \ldots, y_n\}$ and $V = \{x\} \cup Y$. We say that ϕ is *instantiation-correct* if, for all constraints c of type V, for all $I_V^{d,v} = \{c_x\} \cup I_Y^{d',v'}$, we have

$$\phi(\{c\} \cup I_Y^{d',v'}, c_x) = Sol(\langle\{c\} \cup I_V^{d,v}, \{x\}\rangle).$$

Informally, ϕ is instantiation-correct if the rule $\Phi(r)$ performs an exact computation as soon as all the variables appearing in such a rule are instantiated. For example, ϕ_{best} is instantiation-correct, but ϕ_{worst} is not.

For approximation functions satisfying the previous definition, partial AC performs the same domain reductions as AC, as soon as all the variables of the problem are instantiated.

Theorem 3.4.8 (correctness of PAC). *Given an SCSP $P = \langle C, con \rangle$, and an approximation scheme Φ for AC that is instantiation-correct, assume that $V(C) = \{x_1, \ldots, x_p\}$ with $x_1 \prec \cdots \prec x_p$. Let $P^t_{V(C)} = \langle C \cup I^t_{V(C)}, con \rangle$ be a complete labeling of P. Let $P'' = pac(P^t_{V(C)}, \Phi)$ and $P' = ac(P^t_{V(C)})$. Then, for each i in $[1, p]$, $[\{x_i\}]_{P'} = [\{x_i\}]_{P''}$.*

Proof. We will study the computation of the two algorithms (ac and pac). Because of the property of order-independence, we can assume that the rules are applied with the same strategy. Φ is correct w.r.t. to the instantiation. Hence, for each step (rule application and approximation of a rule application), the two algorithms compute the same domains (since all the variables are instantiated). Finally, for each i in $[1, p]$, $[\{x_i\}]_{P'} = [\{x_i\}]_{P''}$.

Corollary 3.4.1. *With the notations of Theorem 3.4.8, for each i in $[1, p]$, the value $[\{x_i\}]_{P''}$ of location x_i in problem P'' is a simple instantiation constraint c_{x_i,t_i,v_i}. Moreover, $blevel(P^t_{V(C)}) = \prod_{i \in [1,p]} v_i$.*

Proof. Easily follows from the previous theorem and from Theorem 3.4.3.

3.4.3 Domains

We have seen that it is possible to approximate the computation of AC. We are now going to restrict our attention to certain kinds of approximations for AC, for which there exists a "good" representation for the variable domains. Then, we will adapt the computation of the approximations to these representations.

In the following we will always assume to have a constraint system $CS = \langle S, D, V \rangle$ where D is totally ordered. This means that there must be a way to compare any two elements of the domain D.

We call a *domain* any unary constraint. Given a set of domains E, we are looking for approximations schemes that build domains in E. Because of the properties of approximations, the set E has to satisfy the following three properties:

- for each domain, there must be a bigger domain in E;
- for each domain, there must be a smaller domain in E;
- E has to contain instantiation constraints (by instantiation-correctness); or, equivalently: $E \supseteq \{\langle \mathbf{1}, \{x\} \rangle | x \in V\} \cup \{c_{x,i,v} | x \in V, i \in D, v \in S\}$.

We will call an *approximation domain set* a set satisfying the above properties. The following proposition states that the notion of approximation domain is sufficient for the purpose of this work.

Proposition 3.4.2. *Given an approximation domain set E, there exists an approximation scheme Φ such that the application of an approximation function (corresponding to Φ) builds domains in E.*

Proof. To prove this proposition, just take Φ such that, for each rule $r = \{x\} \leftarrow \{\{x, y_1, \ldots, y_n\}, \{y_1\}, \ldots, \{y_n\}\}$, for each $c \in \mathit{type}^{-1}(\{x, y_1, \ldots, y_n\})$[4], for each $i \in [1, n]$, for each $c_{y_i} \in \mathit{type}^{-1}(\{y_i\})$, for each $c_x \in \mathit{type}^{-1}(\{x\})$, we have $\Phi(r)(\{c, c_{y_1}, \ldots, c_{y_n}\}, c_x) = \mathit{Sol}(\langle\{c, c_x, c_{y_1}, \ldots, c_{y_n}\}, \{x\})$ if c_x and $c_{y_i}(\forall i \in [1, n])$ are instantiations constraints, and $\Phi(r)(\{c, c_{y_1}, \ldots, c_{y_n}\}, c_x) = c_x$ otherwise.

The choice of an approximation domain set is very open. The approximation domains, however, should have a fixed memory requirement. Here we will consider some generalizations of the min-max scheme usually used in current constraint programming.

An *up-down-stair* is a domain $d = (d(1), \ldots, d(n))$ such that there exists $k \in [1, n]$ s.t. $d(1) \leq_S \ldots d(k) \geq_S \ldots d(n)$. We call *m-up-down-stair* any *up-down-stair* domain such that $|d(D)| - 1 \leq m$. We note $S(m)$ the set of m-up-down-stairs.

Note that the 1-up-down-stair corresponds exactly to the min-max scheme in the case where the semiring is the boolean one. Figure 3.14 presents an example of a 3-up-down-stair in the case of the fuzzy semiring.

Note that an m-up-down-stair can be stored using $2m$ integers and m semiring values. A m-up-down-stair is made of m rectangles of decreasing width. Then, for each k between 1 and m, one has to store the interval $[\mathit{inf}_k, \mathit{sup}_k]$ that defines the kth rectangle and the total height of the k first rectangles. For the example of Figure 3.14, we get: $([2, 9], 0.2); ([4, 9], 0.4); ([5, 7], 0.5)$.

Let us now slightly restrict our definition of approximation functions in order to give some insights for their construction. We suppose m given and we will focus on m-up-down-stair domains. Let us consider a constraint c over $V = \{x\} \cup Y$ where $Y = \{y_1, \ldots, y_n\}$ and $y_1 \prec \cdots \prec y_n$. Let us also consider one of the associated AC rules: $\{x\} \leftarrow \{V, \{y_1\}, \ldots, \{y_n\}\}$. We are looking for an approximation function $\phi : \mathit{type}^{-1}(\{V, \{y_1\}, \ldots, \{y_n\}\}) \times \mathit{type}^{-1}(\{x\}) \rightarrow \mathit{type}^{-1}(\{x\})$.

Since we do not consider general local consistency but only AC, only the domains can change; this is the reason why the approximation function should not take the constraint as a parameter. Moreover, because of their presence in many applications, and also for implementation issues, we will focus on approximations functions that intersect the new domain with the old one (more precisely, we will consider the case of a unique intersection function).

Hence, we will focus on the case where ϕ is defined by:

$$\phi(\{c, c_{y_1}, \ldots, c_{y_n}\}, c_x) = \chi(\psi(c_{y_1}, \ldots, c_{y_n}), c_x)$$

where:

- $\psi : S(m) \times \ldots \times S(m) \rightarrow S(m)$ computes a new domain (m-up-down-stair) for variable x;
- $\chi : S(m) \rightarrow S(m)$ intersects[5] the new domain and the old one.

[4] The function $\mathit{type}^{-1}(V)$ gives the constraint(s) c s.t. $\mathit{type}(c) = V$

[5] In the general case, $\otimes$ is not a function from $S(m) \times S(m)$ to $S(m)$ (this is true only in the case where $m = 1$), hence we have to use χ instead.

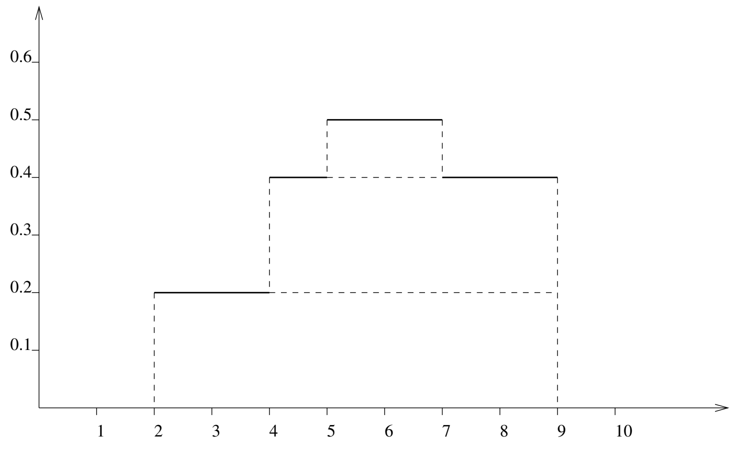


Fig. 3.14. A 3-up-down-stair

In the following, we will assume that the domain of the variable D has the form $[0, \infty]$ where ∞ is the maximal value an integer can take. As a practical and simple example, let us consider the case where $m = 1$. Here $\otimes$ is stable over $S(m)$, hence we take $\chi = \otimes$, which can be implemented by:

$$\chi(([\mathtt{i}_1, \mathtt{s}_1], \mathtt{v}_1), ([\mathtt{i}'_1, \mathtt{s}'_1], \mathtt{v}'_1)) = ([\mathtt{max}(\mathtt{i}_1, \mathtt{i}'_1), \mathtt{min}(\mathtt{s}_1, \mathtt{s}'_1)], \mathtt{v}_1 \times \mathtt{v}'_1).$$

Using 1-up-down-stairs, it is also easy to implement the rules for usual constraints.

Example 3.4.4 (constraint $X \leq Y$). For example, the two rules corresponding to constraint $x \leq y$ can be implemented by:

$$\psi_x(([\mathtt{i}_\mathtt{y}, \mathtt{s}_\mathtt{y}], \mathtt{v}_\mathtt{y})) = ([\mathtt{0}, \mathtt{s}_\mathtt{y}], \mathtt{v}_\mathtt{y})$$

and

$$\psi_y(([\mathtt{i}_\mathtt{x}, \mathtt{s}_\mathtt{x}], \mathtt{v}_\mathtt{x})) = ([\mathtt{s}_\mathtt{x}, \infty], \mathtt{v}_\mathtt{x}).$$

Example 3.4.5 (constraint $X = Y + C$). As another example, the two rules corresponding to constraint $x = y + c$ can be implemented by:

$$\psi_x(([\mathtt{i}_\mathtt{y}, \mathtt{s}_\mathtt{y}], \mathtt{v}_\mathtt{y})) = ([\mathtt{i}_\mathtt{y} + \mathtt{c}, \mathtt{s}_\mathtt{y} + \mathtt{c}], \mathtt{v}_\mathtt{y})$$

and

$$\psi_y(([\mathtt{i}_\mathtt{x}, \mathtt{s}_\mathtt{x}], \mathtt{v}_\mathtt{x})) = ([\mathtt{i}_\mathtt{x} - \mathtt{c}, \mathtt{s}_\mathtt{x} - \mathtt{c}], \mathtt{v}_\mathtt{x}).$$

In the case where $m > 1$, it is also possible to implement χ and the rules associated with usual constraints. This implementation is more complicated since there are different representations corresponding to the same up-down-stair. Hence, one has to define a function that chooses a representation among all the possible ones.

3.5 Constraint Propagation: Generalization and Termination Conditions

As described in Section 3.1 the propagation techniques usually used for classical CSPs have been extended and adapted to deal with soft constraints, provided that certain conditions are met. This has led to a general framework for soft constraint propagation, where at each step a subproblem is solved, as in classical constraint propagation. By studying the properties of this schema, it is proved (see Theorem 3.1.3 and 3.1.6) that such steps can be seen as applications of functions that are monotonic, inflationary, and idempotent over a certain partial order.

On an orthogonal line of research, the concept of constraint propagation over classical constraints has been studied in depth in [10, 11, 12], and a general algorithmic schema (called GI) has been developed. In such a schema, constraint propagation is achieved whenever we have a set of functions that are monotonic and inflationary over a partial order with a bottom.

By studying these two frameworks and comparing them, we noticed that the GI schema can be applied to soft constraints (see Section 3.5.4), since the functions and the order used for soft constraints have all the necessary properties for GI. This is proven in this section by extending the partial order over soft constraint problems defined in Section 2.2.

By analyzing the features of the GI algorithm, we also realized (see Section 3.5.4) that indeed soft constraint propagation can be extended to deal with functions that are not necessarily idempotent. Notice that this is a double generalization: we don't require any longer that each step has to solve a subproblem (it could do some other operation over the problem), nor that it is idempotent. This allows us to model several forms of "approximate" constraint propagation, which were instead not modeled in [47]. Examples are: bounds-consistency for classical constraints [141], and partial soft arc-consistency for soft constraints [34] described in Section 3.4.

These two results allow us to use the GI algorithm schema for performing a generalized form of soft constraint propagation. What is important to study, at this point, is when the resulting GI schema terminates. In fact, if we work with classical constraints over finite domains, it is easy to see that the GI algorithm always terminates. When moving to soft constraints over a semiring, however, even if the variable domain is finite, we could have an infinite behavior due to an infinite number of elements in the semiring. For example, fuzzy constraints have a semiring containing all reals between 0 and 1, and the semiring of weighted constraints contains all the reals, or all the naturals.

In Section 3.5.5 we identify some sufficient conditions for the termination of the GI algorithm over soft constraints. The first, predictable, condition that we consider is the well-foundness of the partial order over soft constraint problems: if the partial order over which the GI algorithm works has chains of finite length, since constraint propagation never goes from one chain to another one, obviously the whole algorithm terminates.

The second condition is in some sense more precise, although less general: when the propagation steps are defined via the two semiring operations, then we can just consider the sub-order over semiring elements obtained by taking the elements initially appearing in the given problem, and closing it under the two operations. In fact, in this case the GI algorithm cannot reach other elements. Therefore, if such a set (or a superset of it) is well-found, the GI algorithm terminates.

Each of these two conditions is sufficient for termination; however, they could be difficult to check, unless the partial order has a well-known structure of which we know the well-foundness. Nevertheless, in a special case we can formally prove that there exists a well-founded set of the shape required by the second condition above, and thus we can automatically deduce termination. This special case is related to the idempotency of the multiplicative operation of the semiring, the one that we use to combine constraints: if this operation is idempotent, then GI terminates. For example, in classical constraints the multiplicative operation is logical *and*, and in fuzzy constraints it is the minimum, thus we can formally prove that the algorithm GI over *any* classical or fuzzy constraint problem *always* terminates, provided that the functions are defined via the two semiring operations.

We believe that the generalizations and termination conditions that we have developed and proven will make soft constraints more widely applicable, and soft constraint propagation more practically usable.

3.5.1 Some Useful Orderings over Semiring Constraints

We now review and modify some of the orderings among semiring elements, constraints, and problems, which have been introduced in the previous chapter; moreover, we also define new orderings that will be used in the next sections.

All the orderings we will consider in this section are derived from the partial order $\leq_S$ over semiring elements, which, we recall, is defined as follows: $a \leq_S b$ iff $a + b = b$. This intuitively means that b is "better" than a.

Definition 3.5.1. *Consider any partial ordering $\langle D, \sqsubseteq \rangle$ and the componentwise ordering $\langle D^n, \sqsubseteq_n \rangle$, with $n \geq 1$, where $\langle d_1, \ldots, d_n \rangle \sqsubseteq_n \langle d'_1, \ldots, d'_n \rangle$ iff $d_i \sqsubseteq d'_i$ for each $i = 1, \ldots, n$. Let f be a function from D^n to D. Then:*

- *f is monotonic iff $\langle d_1, \ldots, d_n \rangle \sqsubseteq_n \langle d'_1, \ldots, d'_n \rangle$ implies $f(\langle d_1, \ldots, d_n \rangle) \sqsubseteq f(\langle d'_1, \ldots, d'_n \rangle)$;*
- *f is inflationary w.r.t. $\sqsubseteq$ iff $d_i \sqsubseteq f(\langle d_1, \ldots, d_n \rangle)$ for every $i = 1, \ldots, n$.*

Given the definition above, it is easy to see that the following results hold when D is the semiring set A and the order considered is $\leq_S$:

- $\leq_S$ is a partial order;
- $\mathbf{0}$ is the minimum;
- $\mathbf{1}$ is the maximum.

– if $\times$ is idempotent, then $\langle A, \leq_S \rangle$ is a distributive lattice where $+$ is the lub and $\times$ is the glb.
– $+$ and $\times$ are monotonic with respect to $\leq_S$;
– $+$ is inflationary with respect to $\leq_S$; instead $\times$ is inflationary with respect to $\geq_S$.

Constraint Orders. From the ordering $\leq_S$ over A, we can also define a corresponding order between constraints (that extend the ordering already defined in the previous chapter). Before introducing the new order we define its domain, namely the set of all possible constraints over a constraint system.

Definition 3.5.2. *Given a semiring $S = \langle A, +, \times, \mathbf{0}, \mathbf{1} \rangle$ and a constraint system $CS = \langle S, D, V \rangle$, we define the* Constraint Universe *related to the constraint system CS as follows: $\mathcal{C}_{CS} = \bigcup_{con \subseteq V} \{\langle def, con \rangle \mid def : D^{|con|} \rightarrow A\}$.*

We will write $\mathcal{C}$ (instead of $\mathcal{C}_{\mathcal{CS}}$) when the constraint system CS is clear from the context.

Definition 3.5.3. *Consider two constraints c_1, c_2 over a constraint system CS; assume that $con_1 \supseteq con_2$ and $|con_1| = k$. Then we write $c_1 \sqsubseteq_S c_2$ if and only if, for all k-tuples t of values from D, $def_1(t) \leq_S def_2(t \downarrow^{con_1}_{con_2})$.*

Loosely speaking, a constraint c_1 is smaller than c_2 in the order $\sqsubseteq_S$ iff it constrains possibly more variables and assigns to each tuple a smaller value with respect to $\leq_S$ than c_2 does.

Theorem 3.5.1 ($\sqsupseteq_S$ is a po). *Given a semiring $S = \langle A, +, \times, \mathbf{0}, \mathbf{1} \rangle$ with $\times$ idempotent and a constraint system $CS = \langle S, D, V \rangle$, we have the following:*

- *the relation $\sqsupseteq_S$ is a partial order over the set $\mathcal{C}_{CS}$;*
- *its bottom is $\langle \mathbf{1}, \emptyset \rangle$, where the 0-arity function $\mathbf{1} : \emptyset \rightarrow A$ is the constant $\mathbf{1}$ of the semiring.*

Proof. We prove our first claim. We need to demonstrate that $\sqsubseteq_S$ is a reflexive, antisymmetric and transitive relation. Reflexivity holds trivially. To prove antisymmetry, suppose that $c_1 \sqsubseteq_S c_2$ and $c_2 \sqsubseteq_S c_1$; this yields that $con_1 = con_2$. Now, for all $t \in D^{|con_1|}$, we have both $def_1(t) \leq_S def_2(t)$ and $def_2(t) \leq_S def_1(t)$, hence $def_1(t) = def_2(t)$ and so $c_1 = c_2$. The transitivity of $\sqsubseteq_S$ follows from the transitivity of $\leq_S$. The other claim immediately follows from the definition of $\sqsubseteq_S$.

We can easily extend the order $\sqsubseteq_S$ over constraints to a new order over *constraint sets* as follows.

Definition 3.5.4. *Consider two sets of constraints C_1, C_2 over a constraint system CS. Suppose furthermore that $C_1 = \{c^1_i : i \in I\}$, $C_2 = \{c^2_j : j \in J\}$, $J \subseteq I$ and that, for every $j \in J$, the relation $c^1_j \sqsubseteq_S c^2_j$ holds. Then we write $C_1 \sqsubseteq_C C_2$.*

The intuitive reading of $C_1 \sqsubseteq_C C_2$ is that C_1 is a problem generally "more constraining" than C_2 is, because C_1 has (possibly) a larger number of "more restrictive" constraints than C_2 has.

Theorem 3.5.2 ($\sqsupseteq_C$ is a partial order). *Given a semiring $S = \langle A, +, \times, \mathbf{0}, \mathbf{1}\rangle$, and a constraint system $CS = \langle S, D, V\rangle$, we have that:*

- *the relation $\sqsupseteq_C$ is a partial order over $\wp(\mathcal{C})$;*
- *the bottom of the relation is $\emptyset$.*

Proof. We only prove the first claim, the other one being straightforward. Reflexivity trivially holds. As far as antisymmetry is concerned, suppose that $C_1 = \{c_i^1\}_{i \in I}$, $C_2 = \{c_j^2\}_{j \in J}$ and both $C_1 \sqsubseteq_C C_2$ and $C_2 \sqsubseteq_C C_1$ hold; this means that $I = J$. Moreover, the following relations hold for every $i \in I$: $c_i^1 \sqsubseteq_S c_i^2$ and $c_i^2 \sqsubseteq_S c_i^1$. Hence $c_i^1 = c_i^2$ for every $i \in I$, because $\sqsubseteq_S$ is a partial order relation, cf. Theorem 3.5.1. Transitivity follows similarly, by exploiting the transitivity of $\sqsubseteq_S$.

So far, we have introduced two partial orders: one between constraints ($\sqsubseteq_S$) and another one between constraint sets ($\sqsubseteq_C$). Local consistency algorithms, however, take constraint problems as input; therefore, we need an ordering relation between problems if we want the `GI` algorithm to be used for soft local consistency.

First we define the set of all problems that can be built over a constraint system CS.

Definition 3.5.5. *Given a semiring $S = \langle A, +, \times, \mathbf{0}, \mathbf{1}\rangle$ and a constraint system $CS = \langle S, D, V\rangle$, we define the* Problem Set Universe *related to the constraint system CS as $\mathcal{P}_{CS} = \{\langle C, con\rangle \mid C \subseteq \mathcal{C}_{CS}, con \subseteq var(C)\}$. When no confusion can arise, we shall simply write $\mathcal{P}$ instead of $\mathcal{P}_{CS}$.*

Definition 3.5.6. *Given a constraint system CS, consider two problems $P_1 = \langle C_1, con_1\rangle$ and $P_2 = \langle C_2, con_2\rangle$ in $\mathcal{P}_{CS}$. We write $P_1 \sqsubseteq_{CP} P_2$ iff $C_1 \sqsubseteq_C C_2$ and $con_2 \subseteq con_1$.*

We now need to define a partially ordered structure that contains all SCSPs that can be generated by enforcing local consistency, starting from a given problem.

Definition 3.5.7. *Consider a constraint system CS and an SCSP P over it. The* up-closure *of P, briefly $P \uparrow$, is the class of all problems P' on CS such that $P \sqsupseteq_{CP} P'$.*

Proposition 3.5.1. *Consider a constraint system CS, an SCSP P over it and its up-closure $P \uparrow$. Then the following statements hold:*

1. *if $P_1 \sqsupseteq_{CP} P_2$ and $P_1 \in P \uparrow$, then $P_2 \in P \uparrow$;*
2. *if $P_1 \sqsupseteq_{CP} P_2$, then $P_2 \uparrow \subseteq P_1 \uparrow$.*

Proof. The proof of the above proposition is an immediate consequence of the previous definition and of the fact that $\sqsubseteq_C$ is transitive, cf. Theorem 3.5.2.

Theorem 3.5.3 ($\sqsupseteq_{CP}$ is a po). *Given a constraint system $CS = \langle S, D, V \rangle$ and a problem P on it, we have:*

- *the relation $\sqsupseteq_{CP}$ is a partial order over $\mathcal{P}_{CS}$;*
- *in particular $\langle P \uparrow, \sqsupseteq_{CP|P\uparrow} \rangle$ is a partial ordering, where $\sqsupseteq_{CP|P\uparrow}$ is the restriction of $\sqsupseteq_{CP}$ to $P \uparrow$; when no confusion can arise, we simply write $\langle P \uparrow, \sqsupseteq_{CP} \rangle$;*
- *the bottom $\bot_{CS}$ of $\langle P \uparrow, \sqsupseteq_{CP} \rangle$ is P.*

Proof. We prove the first claim, the other ones following immediately from the definition of $P \uparrow$ and Proposition 3.5.1. As usual, we only prove that the relation is antisymmetric, because transitivity can be proven similarly and reflexivity trivially holds. Hence, suppose that both $P_1 \sqsubseteq_{CP} P_2$ and $P_2 \sqsubseteq_{CP} P_1$ hold. This means that we have the following relations: $con_2 \subseteq con_1$, $C_1 \sqsubseteq_C C_2$, $con_1 \subseteq con_2$, $C_2 \sqsubseteq_C C_1$. From the two previous relations and Theorem 3.5.2, it follows that $con_1 = con_2$ and $C_1 = C_2$; hence $P_1 = P_2$.

3.5.2 Order-Related Properties of Soft Local Consistency Rules

We remind, from Section 2.2, that two problems P_1 and P_2, which share the same set of variables, are equivalent if they have the same solution set, and we write $P_1 \equiv_P P_2$.

Now we can list some useful properties of soft local consistency rules, which are related to equivalence and to problem ordering. Here we assume that we are given a constraint system C and a rule r on CS:

– (equivalence) $P \equiv_P r_l^L(P)$ if $\times$ is idempotent.
– (inflationarity) $P \sqsupseteq_{CP} r_l^L(P)$. This means that the new semiring values assigned to tuples by the rule application are always smaller than or equal to the old ones with respect to $\leq_S$.
– (monotonicity) Consider two SCSPs $P_1 = \langle C_1, con_1 \rangle$ and $P_2 = \langle C_2, con_2 \rangle$ over CS. If $P_1 \sqsubseteq_{CP} P_2$, then $r(P_1) \sqsubseteq_{CP} r(P_2)$.

It is easy to prove that all the results about local consistency rules hold also for a whole local consistency algorithm. Moreover, we can also prove that the strategy does not influence the result, if it is fair (see Section 3.1).

3.5.3 The Generic Iteration Algorithm

In [11, 12] the Generic Iteration (`GI`) algorithm is introduced to find the least fixpoint of a finite set of functions defined on a partial ordering with bottom. This was then used as an algorithmic schema for classical constraint propagation: each step of constraint propagation was seen as the application of one of these functions. Our idea is to compare this schema with the one used for soft constraints, with the aim of obtaining a new schema which is the most general (that is, it can be applied both to classical and to soft constraints) and has the advantages of both of them.

Given a partial ordering with bottom, say $\langle D, \sqsubseteq, \perp \rangle$, consider now a set of functions $F := \{f_1, \ldots, f_k\}$ on D. The following algorithm can compute the least common fix point of the functions in F.

Generic Iteration Algorithm (GI)

```
$d := \perp$;
$G := F$;
while $G \neq \emptyset$ do
    choose $g \in G$;
    $G := G - \{g\}$;
    $G := G \cup update(G, g, d)$;
    $d := g(d)$
od
```

where for all G, g, d the set of functions $update(G, g, d)$ from F is such that:

A. $\{f \in F - G \mid f(d) = d \wedge f(g(d)) \neq g(d)\} \subseteq update(G, g, d)$;
B. $g(d) = d$ implies $update(G, g, d) = \emptyset$;
C. $g(g(d)) \neq g(d)$ implies $g \in update(G, g, d)$.

Assumption **A** states that $update(G, g, d)$ at least contains all the functions from $F - G$ for which d is a fix point but $g(d)$ is not. So at each loop iteration such functions are added to the set G. In turn, assumption **B** states that no functions are added to G in case the value of d did not change. Note that, even though after the assignment $G := G - \{g\}$ we have $g \in F - G$, still $g \notin \{f \in F - G \mid f(d) = d \wedge f(g(d)) \neq g(d)\}$ holds. So assumption **A** does not provide any information when g is to be added back to G. This information is provided in assumption **C**. On the whole, the idea is to keep in G at least all functions f for which the current value of d is not a fix point.

We now recall the results which state the (partial) correctness of the GI algorithm, cf. [11, 12]:

i. Every terminating execution of the GI algorithm computes in d a common fixpoint of the functions from F.
ii. Suppose that all functions in F are monotonic. Then every terminating execution of the GI algorithm computes in d the least common fixpoint of all the functions from F.
iii. Suppose that all functions in F are inflationary and that D is finite. Then every execution of the GI algorithm terminates.

3.5.4 Generalized Local Consistency for SCSPs via Algorithm GI

In this section we will try to combine the two formalisms described so far (soft constraints and the GI algorithm). Our goal is to exploit the GI algorithm to perform local consistency over soft constraint problems.

GI for Standard Local Consistency over Soft Constraints. The functions that `GI` needs in input are defined on a partial ordering with bottom. In the case of local consistency rules for SCSPs, the partial ordering is $\langle P \uparrow, \sqsupseteq_{CP} \rangle$, and the bottom is the problem P itself, cf. Theorem 3.5.3. Moreover, the local consistency rules (and also the more general local consistency functions) have all the "good" properties that GI needs. Namely, those functions are monotonic and inflationary. Thus, algorithm GI can be used to perform constraint propagation over soft constraint problems. More precisely, we can see that algorithm GI and the local consistency algorithm schema for soft constraints obtain the same result.

Theorem 3.5.4 (GI for soft local consistency rules). *Given an SCSP P over a constraint system CS, consider the SCSP $lc(P, R, S)$ obtained by applying to P a local consistency algorithm using the rules in R and with a fair strategy S. Consider also the partial order $\langle P \uparrow, \sqsupseteq_{CP} \rangle$, and the set of functions R, and apply algorithm GI to such input. Then the output of GI coincides with $lc(P, R, S)$.*

Proof. Since we already know that $lc(P, R, S)$ and *GI* terminate (by the results appeared in [10, 47], and since the computed fixpoint is the same (by previous item *ii.*), their output obviously coincide.

GI for Generalized Local Consistency over Soft Constraints. While all local consistency rules are idempotent (since they solve a subproblem), algorithm GI does not need this property. This means that we can define a generalized notion of local consistency rules for soft constraints, by dropping idempotency.

Definition 3.5.8 (local consistency functions). *Consider an SCSP P over a semiring S. A* local consistency function *for P is a function $f : P \uparrow \rightarrow P \uparrow$ which is monotonic and inflationary over $\sqsupseteq_{CP}$.*

With this definition of a local consistency function we relax two conditions about a local consistency step:

- that it must solve a subproblem;
- that it must be idempotent.

The second generalization has been triggered by the results about the GI algorithm, which have shown that idempotency is not needed for the desired results. Moreover, many practical local consistency algorithms do not exactly solve subproblems, but generate an approximation of the solution (see for example the definition of *bounds consistency* in [141] or the notion of partial soft arc-consistency in [34]). Thus, the first extension allows one to model many more practical propagation algorithms.

Theorem 3.5.5 (GI for soft local consistency functions). *Given a constraint system CS and an SCSP P on it, let us apply the* `GI` *algorithm to the partial order $\langle P \uparrow, \sqsupseteq_{CP} \rangle$ and a finite set R of local consistency functions. Then every terminating execution of the* `GI` *algorithm computes in the output problem P' the least common fixpoint of all the functions from R.*

Proof. Easily follows by item *i.* of the previous section.

What is now important to investigate is when the algorithm terminates. This is particularly crucial for soft constraints, since, even when the variable domains are finite, the semiring may contain an infinite number of elements, which is obviously a source of possible non-termination.

3.5.5 Termination of the GI Algorithm over Soft Constraints

As noted above, the presence of a possibly infinite semiring may lead to a constraint propagation algorithm that does not terminate. In the following we will give several independent conditions which guarantee termination in some special cases.

The first condition is a predictable extension of the one given in Section 3.5.3: instead of requiring the finiteness of the domain of computation, we just require that its chains have finite length, since it is easy to see that constraint propagation moves along a chain in the partial order.

Theorem 3.5.6 (termination 1). *Given a constraint system CS and an SCSP P on it, let us instantiate the* `GI` *algorithm with the po* $\langle P \uparrow, \sqsupseteq_{CP} \rangle$ *and a finite set R of local consistency functions. Suppose that the order* $\sqsupseteq_{CP}$ *restricted to* $P \uparrow$ *is well founded. Then every execution of the* `GI` *algorithm terminates.*

Proof. Easily follows from Theorem 1 in [11,12]. Note that there the author says to need a finite (and not a well found) partial order, but the proof do not uses this restriction.

This theorem can be used to prove termination in many cases. For example, classical constraints over finite domains generate a partial order that is finite (and thus trivially well-found), so the above theorem guarantees termination. Another example occurs when dealing with weighted soft constraints, where we deal with the naturals. Here the semiring is $\langle N, min, +, 0, +\infty \rangle$. Thus we have an infinite order, but well-found.

There are also many interesting cases, however, in which the ordering $\langle P \uparrow, \sqsupseteq_{CP} \rangle$ is not well-found. Consider for instance the case of fuzzy or probabilistic CSPs. For fuzzy CSPs, the semiring is $\langle [0,1], max, min, 0, 1 \rangle$. Thus, the partially ordered structure containing all problems that are smaller than the given one, according to the semiring partial order, is not well-found, since we have all the reals between 0 and a certain element in $[0,1]$. Thus, the above theorem cannot say anything about termination of GI in this case. This does not mean, however, that GI does not terminate, but only that the theorem above cannot be applied. In fact, later we will give another sufficient condition that will guarantee termination in the last two cases as well.

In fact, if we restrict our attention to local consistency functions defined via $+$ and $\times$, we can define another condition on our input problem that guarantees the termination of the GI algorithm; this condition exploits the fact that the local consistency functions are defined by means of the two semiring operations, and the properties of such operations.

Definition 3.5.9 (semiring closure). *Consider a constraint system CS with semiring $S = \langle A, +, \times, \mathbf{0}, \mathbf{1} \rangle$, and an SCSP P on CS. Consider also the set of semiring values appearing in P: $Cl(P) = \bigcup_{\langle def', con' \rangle \in C} \{def'(d) \mid d \in D^{|con'|}\}$. Then, a* semiring closure *of P is any set B such that: $Cl(P) \subseteq B \subseteq A$; B is closed with respect to $+$ and $\times$; $<_S$ restricted to B is well founded.*

Theorem 3.5.7 (termination 2). *Consider a constraint system CS with semiring $S = \langle A, +, \times, \mathbf{0}, \mathbf{1} \rangle$, an SCSP P on it and a finite set of local consistency functions R defined via $+$ and $\times$. Assume there also exists a semiring closure of P. Then every execution of the* `GI` *algorithm terminates.*

Proof. The proof is similar to the one in [11, 12] for the termination theorem; just replace the order $\sqsubseteq_{CP}$ with $\leq_S$ and accordingly the set D with B, where B is a semiring closure of P.

Notice that this theorem is similar to the one in Section 3.1 about termination; however there we force the set B to be finite in order to guarantee the termination of a local consistency algorithm; a hypothesis that is implied by ours.

If we have a fuzzy constraint problem, then we can take B as the set of all semiring values appearing in the initial problem. In fact, this set is closed with respect to min and max, which are the two semiring operations in this case. Moreover, it is well-found, since it is finite. Another example is constraint optimization over the reals: if the initial problem contains only natural numbers, then the set B can be the set of all naturals, which is is a well-founded subset of the reals and it is closed w.r.t. $+$ (min) and $\times$ (sum).

Therefore, by using Theorem 3.5.7 we can also prove that constraint propagation over fuzzy constraint problems always terminates, provided that each step of the algorithm uses a local consistency function, which is defined in terms of the two semiring operations only.

It is not, however, always easy to find a semiring closure of a given SCSP P, mainly because we should check that the order restricted to a tentative set B, closed and containing $Cl(P)$, is well-found. Nevertheless, there is a special case in which we do not have to find such a set, because we can prove that it always exists (which is what Theorem 3.5.7 requires). This special case occurs when the multiplicative operation of the semiring is idempotent. In fact, we can prove that in this case there always exists a finite (and thus well-found) semiring closure of any given problem over that semiring. This obviously is very convenient, since it provides us with an easy way to check whether Theorem 3.5.7 can be applied.

Theorem 3.5.8 (idempotency of $\times$ and termination). *Consider a constraint system CS, an SCSP P on it and a finite set of local consistency functions R defined via $+$ and $\times$. Assume also that $\times$ is idempotent. Then there exists a finite semiring closure of P, and thus every execution of the GI algorithm terminates.*

Proof. Consider the set $Cl(P)$ of all semiring elements appearing in P. If we combine any subset of them via the $+$ operation, we generate a set of elements, which contains $Cl(P)$, that we denote by $Cl+$. Notice that this set is finite, because the number of subsets of $Cl(P)$ is finite.

Let us now combine any subset of elements of $Cl+$ via the $\times$ operation: in this way we generate a larger set of elements, containing $Cl+$, which we denote by $Cl + \times$. Again, this set is finite. Therefore $Cl + \times$ is a finite semiring closure of P.

Consider again the fuzzy constraint example. Here $\times$ is min, thus it is idempotent. Therefore, by Theorem 3.5.8, GI over such problems always terminates. This is an alternative, and easier, way (to Theorem 3.5.7) to guarantee that soft constraint propagation over fuzzy constraints terminates. In fact, we do not have to find a semiring closure of the problem, but just check that the multiplicative operation is idempotent.

Considering all the above results, we can devise the following steps towards proving the termination of algorithm GI on a soft constraint problem P over a semiring S:

- If the local consistency functions are defined via the two operations of S, and the multiplicative operation of S is idempotent, then GI terminates (by Theorems 3.5.8).
- If instead $\times$ is not idempotent, but we still have local consistency functions defined via the two semiring operations, we can try to find a semiring closure of P. If we find it, then GI terminates (by Theorem 3.5.7).
- If we cannot find a semiring closure of P, or the local consistency functions are more general, then we can try to prove that the partial order of problems is well-found. If it is so, the GI terminates (by Theorem 3.5.6).

While Theorem 3.5.8 applies in a special case of the hypothesis of Theorem 3.5.7, it is interesting to investigate the relationship between the hypothesis of Theorem 3.5.6 and 3.5.7. What can be proven is that these two conditions, namely, the well-foundness of the partial order of problems and the existence of a semiring closure, are independent. In other words, there are cases in which one holds and not the other one, and vice versa. To prove this result, we need the following definition.

Definition 3.5.10. *Let $S = \langle A, +, \times, \mathbf{0}, \mathbf{1}\rangle$ be a semiring and B a subset of A. The set B is a* down-set *(or an* order ideal*) if, whenever $a \in B$, $a' \in A$ and $a' \leq_S a$, then $a' \in B$. Given any subset B of A, the downward closure of B is*

$$B \downarrow := \{d' \in A \,:\, \exists d\,(d \in B \textit{ and } d' \leq_S d)\}.$$

Observe that the class $\mathcal{F}$ of down-sets containing a subset B of A is not empty, since A itself is such a set. Moreover, it is easy to check that the downward closure of B is the smallest down-set of $\mathcal{F}$; hence the downward closure of a set is well defined.

Given a semiring $S = \langle A, +, \times, \mathbf{0}, \mathbf{1}\rangle$, the following result links the upward closure of a problem P with the downward closure of $Cl(P)$, thus allowing us to compare the conditions in Theorem 3.5.6 and 3.5.7.

Proposition 3.5.2. *Given a constraint system CS and a problem P defined on it, consider the set $B := \{def(t) \in A : \exists P' \in P \uparrow (c := \langle def, con\rangle \in P',\ t \in D^{|con|})\}$. Then $B = Cl(P) \downarrow$.*

Proof. It follows immediately from the definition of $\sqsupseteq_{CP}$, of $P \uparrow$ and $Cl(P) \downarrow$.

Now we can notice that, given a subset B of a semiring, if $<_S$ restricted to B is well founded, then so is $<_S$ restricted to $\hat{B} := B \cup \{\mathbf{0}, \mathbf{1}\}$. Furthermore, if B is finite, so is $\hat{B}$. In fact it is sufficient to check that the following identities hold because of the fact that $S = \langle A, +, \times, \mathbf{0}, \mathbf{1}\rangle$ is a c-semiring: if $a \in B$ then $a + \mathbf{0} = a \in B$; if $a \in B$ then $a \times \mathbf{1} = a \in B$; if $a \in B$ then $a + \mathbf{1} = \mathbf{1} \in \hat{B}$; if $a \in B$ then $a \times \mathbf{0} = \mathbf{0} \in \hat{B}$. Hence, in Theorem 3.5.7, we can replace the hypothesis "$Cl(P) \subseteq B$ and B a semiring closure of $Cl(P)$" with the condition that "Cl(P) is a subset of a well founded sub-c-semiring B of §$S = \langle A, +, \times, \mathbf{0}, \mathbf{1}\rangle$".

Moreover, $+$ is the least upper bound operation and, if $\times$ is idempotent, $\times$ is the greatest lower bound operation (see chapter 2 and [47]). Hence a sub-c-semiring is also a sub-lattice of $S = \langle A, +, \times, \mathbf{0}, \mathbf{1}\rangle$ and vice versa if $\times$ is idempotent. Thus, a subset B of a semiring can be a down-set and yet it may be not closed with respect to $\times$ and $+$. For instance, the set of negative real numbers augmented with $-\infty$ is a down-set in the lattice $\boldsymbol{R}^{\infty}$ of reals extended with $\{+\infty, -\infty\}$ and the usual linear ordering; however, it is not a sub-lattice itself. Vice versa, there are sub-lattices of $\boldsymbol{R}^{\infty}$ - hence sets that are closed with respect to the least upper bound and the greatest lower bound operations - that are not down-sets. For instance, the extended interval $[0, 1] \cup \{+\infty, -\infty\}$. Therefore, the two conditions that guarantee the termination of algorithm `GI` in Theorems 3.5.6 and 3.5.7 are independent.

3.6 Dynamic Programming for SCSPs

Dynamic programming [24, 25] can be used to solve a problem by solving some subproblems of it and then combining their solutions to obtain the solution of the whole problem (see for example [116] for its use in graphs and [151, 152] for its adoption in classical CSPs where is called *perfect relaxation*). In the SCSP framework, a suitably instantiated version of dynamic programming can be fruitfully used as well: at each step, a subset of constraints is chosen and solved, and its solution (which, we recall, is a constraint) replaces the whole subset of constraints.

Before defining the algorithm, we need to describe in a tree-like way the SCSP to be solved, so that the algorithm can then follow the tree structure in a bottom-up way.

Definition 3.6.1 (parsing tree). *Given an SCSP $P = \langle C, con\rangle$, a parsing tree of P is a sequence of local consistency rules $S = r_1; \ldots; r_n$, where $r_i = (l_i \leftarrow L_i)$. Let*

- *$L = \{l_i, i = 1, \ldots, n\} \cup \bigcup_{i=1,\ldots,n} L_i$ (that is, L is the set of all locations occurring in all rules in S);*

– *$l_i \prec l$ iff $l \in L_i$ (that is, rule r_i "generates" location l);*
– *$l_i \prec v$ iff $v \notin l_i$ and $v \in l'$ with $l' \in L_i$ (that is, rule r_i "generates" variable v);*
– *for any x location or variable, $prec(x) = \{l \mid l \prec x\}$.*

Then we require that

– (tree-like) *for any x variable or location, if $x = l_n$ or $x \in l_n$, then $prec(x) = \emptyset$, else $prec(x)$ is a singleton;*
– (cover) *$con = l_n$, and $\langle def, con' \rangle \in C$ implies $con' \in L$;*
– (bottom-up parsing) *$l_i \prec l_j$ implies $j < i$.*

In words, the above definition provides a sequence of rules which cover the whole problem and are connected among them as a tree. Moreover, the sequence gives a bottom-up visit of the tree. It is now easy to define a dynamic programming algorithm where, at each step, one rule is processed, according to the given sequence. Note also that the rules we use to explore the tree in a dynamic programming fashion have the same structure of the rule used to perform a step of relaxation of the problem. This means that dynamic programming can be seen as a special case of relaxation algorithm.

Definition 3.6.2 (dynamic programming algorithm). *Given an SCSP P, consider a parsing tree $S = r_1; \ldots; r_n$ of P. Then compute $[S](P)$.*

We will now prove that the value of location l_n when the algorithm terminates coincides with the solution of the given problem P.

Theorem 3.6.1 (solution). *Given an SCSP $P = \langle C, l \rangle$ and a parsing tree S for P, we have that $Sol(P) = [l]_{[S](P)}$.*

Proof. We will prove the statement of the theorem by induction on the length of the parsing tree. For the base case, consider a parsing tree with just one rule, that is, $l \leftarrow L$. It is easy to see that $[l]_{[r](P)}$ is the solution of P. In fact, applying this rule means exactly solving the whole problem. Assume now that the statement holds for all parsings with $n-1$ rules, and consider a parsing S_1 with n rules $r_1, \ldots, r_n$, where $r_i = l_i \leftarrow L_i$ for all $i = 1, \ldots, n$. Consider now the first rule of this parsing, that is, $r_1 = l_1 \leftarrow L_1$, and take another rule r_i such that $l_1 = l_i$ or $l_1 \in L_i$, and there is no other rule r_j with $j < i$ and such that $l_1 = l_j$ or $l_1 \in L_j$. That is, r_i is the first rule after r_1 which has l_1 either in its left hand side or in its right hand side. Note that such a rule must exist by definition of parsing tree. In fact, either $l_1 = l_n$, and in this case l_1 must appear at least in the left hand side on r_n, or $l_1 \neq l_n$, in which case it must be generated by one rule (r_i). We will now consider the two separate cases: that $l_1 = l_i$ and that $l_1 \in L_i$.

Assume that $l_1 \in L_i$. Consider then the sequence $S_2 = r_2, \ldots, r_{i-1}, r_1, r_i, \ldots, r_n$. It is easy to see that this is still a parsing for the given problem. Moreover, the constraint of type l_n resulting after applying the rules in S_2 coincides with that obtained after applying the rules in S_1. In fact, since l_1 can be generated by just one rule (by definition of parsing), all rules in $\{r_2, \dots, r_{i-1}\}$ cannot change the definition of l_1 and thus applying r_1 before or after them cannot change the resulting constraint. Consider now another sequence S_3 of $n-1$ rules $r_2, \dots, r_{i-1}, r'_i, r_{i+1}, \dots, r_n$, where $r'_i = l_i \leftarrow L_i \cup L_1$. It is easy to see that this sequence of rules is again a parsing tree for P, thus by induction hypothesis the constraint with type l_n after the application of S_2 is the solution of the problem. Now we have to show that such a constraint coincides with that obtained at location l_n after applying sequence S_2 (which we already know to coincide with that obtained at location l_n after sequence S_1). What makes S_2 and S_3 differ is the fact that rule r_1 of S_2 has been "merged" with rule r_i to give rule r'_i in S_3. More precisely, the application of rule r'_i combines all constraints specified by $L_1 \cup L_i$, while the application of first rule r_1 and then rule r_i combines first the constraint specified by L_1 and then combines the resulting constraint with those constraints specified by L_i. We will show that these two methods yield the same final constraint.

In the following of this proof, for ease of readability, we will write l instead of $[l]_P$. On one side (rule r'_i) we have the constraint $(\bigotimes(L_i \cup L_1 \cup \{l_i\})) \Downarrow_{l_i}$, and on the other side (rule r_1 and then r_i) we have the constraint $(\bigotimes((L_i - \{l_1\}) \cup \{l_i\} \cup \{(\bigotimes(L_1 \cup \{l_1\})) \Downarrow_{l_1}\})) \Downarrow_{l_i}$. Thus we have to prove that:

$$(\bigotimes(L_i \cup L_1 \cup \{l_i\})) \Downarrow_{l_i} = (\bigotimes((L_i - \{l_1\}) \cup \{l_i\} \cup \{(\bigotimes L_1 \cup \{l_1\}) \Downarrow_{l_1}\})) \Downarrow_{l_i} .$$

We will start from the left hand side of the formula and try to reach the right hand side. We have:
$(\bigotimes(L_i \cup L_1 \cup \{l_i\})) \Downarrow_{l_i} =$
{by separating $(L_i \cup \{l_i\}) - \{l_1\}$ and $L_1 \cup \{l_1\}$ which are disjoint}
$((\bigotimes((L_i \cup \{l_i\}) - \{l_1\})) \bigotimes(\bigotimes(L_1 \cup \{l_1\}))) \Downarrow_{l_i} =$
{by Theorem 2.2.1, and letting $V_i = \bigcup(L_i \cup \{l_i\}$, that is the set of all variables involved in rule r_i }
$((\bigotimes((L_i \cup \{l_i\}) - \{l_1\})) \bigotimes(\bigotimes(L_1 \cup \{l_1\}))) \Downarrow_{V_i} \Downarrow_{l_i} =$
{by Theorem 2.2.2, since the variables not in V_i are not involved in $L_i \cup \{l_i\} - \{l_1\}$}
$((\bigotimes((L_i \cup \{l_i\}) - \{l_1\})) \bigotimes(\bigotimes(L_1 \cup \{l_1\}) \Downarrow_{V_i})) \Downarrow_{l_i} =$
{by Proposition 2.2.1, since $V_i \cap V_1 = l_1$, where $V_1 = (\bigcup L_1) \cup l_1$}
$((\bigotimes((L_i \cup \{l_i\}) - \{l_1\})) \bigotimes(\bigotimes(L_1 \cup \{l_1\})) \Downarrow_{l_1}) \Downarrow_{l_i}$. which coincides with the right hand side of the formula.

Let us now consider the second case, in which $l_1 = l_i$. Then consider the sequence $S_2 = r_2, \dots, r_{i-1}, r_1, r_i, \dots, r_n$. With a reasoning similar to above, it is easy to see that S_2 is still a parsing tree for P and that the constraint of type l_n obtained via S_2 is the same as the one obtained via S_1, since by assumption location l_1 does not appear in rules $r_2, \dots, r_{i-1}$. Consider now $S_3 = r_2, \dots, r_{i-1}, r'_i, r_{i+1}, \dots, r_n$ where $r'_i = l_1 \leftarrow L_i \cup L_1$. Again, S_3 is a parsing tree for P. Moreover, it has $n-1$ rules, thus the constraint of type l_n after the application of S_3 coincides with the solution of P. Now we will show that this is the same as that obtained after the application of S_2. More precisely, we have to show that

$$(\bigotimes(L_1 \cup L_i \cup \{l_1\})) \Downarrow_{l_1} = (\bigotimes(L_i \cup (\bigotimes(L_1 \cup \{l_1\})) \Downarrow_{l_1}) \Downarrow_{l_1} .$$

This can be easily proven by using reasoning similar to that of the first case.

Note that this dynamic programming algorithm can be applied to any instance of the SCSP framework, even when $\times$ is not idempotent (and thus the local consistency algorithms cannot safely be applied).

Obviously any SCSP has a parsing tree[6], but not all classes of SCSPs have *convenient* parsing trees for each SCSP of the class (see [142], where it is shown that the class of rectangular lattices does not have such a property). Here, by convenient we mean that the size of each rule is smaller than some bound N, which is fixed and the same for the whole class of considered SCSPs. In such a case, each step of the algorithm solves an SCSP with bounded size. Thus, the complexity of this step may be exponential in the size of such an SCSP, but constant w.r.t. the size of the overall problem. Therefore, the complexity of the algorithm is linear in the number of rules of the parsing tree, which is linear in the number of variables of the problem. Thus, the overall algorithm is linear in the number of variables of the problem. When applied to standard CSPs, this algorithm reduces to the *perfect relaxation* algorithm of [151, 152].

Definition 3.6.3 (N-bounded parsing tree). *Given an SCSP $P = \langle C, con \rangle$ and an integer N, consider a parsing tree $S = \{l_1 \leftarrow L_1; \ldots; l_n \leftarrow L_n\}$ for P. Then S is N-bounded for P if*

1. *for all $i = 1, \ldots, n$, $\mid l_i \cup \bigcup_{l \in L_i} l \mid \leq N$ (that is, the number of variables of a rule is bounded by N);*
2. *for all $i = 1, \ldots, n$, there is a variable v such that $l_i \prec v$ (that is, each rule generates at least a variable);*
3. *$V_P = \bigcup_{\langle def, con' \rangle \in C} con' = \bigcup_{l \in L} l$ (that is, all the variables occurring in the rules are present in the constraint problem P).*

Theorem 3.6.2 (linear algorithm when convenient parsing). *Given an integer N, consider the class of all SCSPs which have an N-bounded parsing tree. Consider now any SCSP P of this class, and let n be the number of its variables. Then in the worst case P can be solved in time $O(n)$.*

Proof. Just apply the dynamic programming algorithm which follows an N-bounded parsing tree for P, which exists by assumption. Each step of the algorithm applies one of the rules. This is in the worst case $O(\mid D \mid^N \times 2^N)$, where D is the domain of each variable and N is an upper bound to the number of variables of the rules. In fact, the number of tuples of values for the variables of the rule is exponential in the number of variables, which by assumption is bounded by N, and for each tuple one has to check a number of constraints equal to the number of locations, which again can be exponential in the number of variables of the rule. Thus, the worst case time complexity of a rule application is constant, since both N and D are fixed. There are as many steps as rules in the parsing tree. Since each rule by assumption generates at least one variable, the number of rules is bounded by the number of variables n of the whole problem. Therefore, in the worst case the number of steps of the algorithm is $O(n)$.

[6] Just take the parsing tree with just one rule.

Conditions *2* and *3* in Definition 3.6.3 are not restrictive for Theorem 3.6.2. In fact, if we have a parsing for some problem, which satisfies condition *1* but where some rule, say $r = l \leftarrow L$, does not eliminate any variable, then it is always possible to obtain another parsing for the same problem that still satisfies condition *1* but where each rule eliminates at least a variable: just eliminate rule r and replace every occurrence of l in the other rules with L, and do similarly for all rules that do not eliminate variables. If instead there are rules containing variables that do not appear in the problem, then one can always obtain another parsing where such variables are not there: just remove all locations involving such "ghost" variables. By doing this, we still have the cover property of the parsing, since a location involving one or more ghost variables cannot correspond to any constraint of the given problem.

Theorem 3.6.2 states that in some cases there is an algorithm which is $O(n)$. One could wonder whether n is a good measure of the size of the given SCSP, and consider instead the number m of its constraints as more significant. In fact, in general the number of constraints may be much larger than the number of variables (for example, in binary CSPs, m is $O(n^2)$). For the classes of SCSPs that have an N-bounded parsing tree, however, it is possible to show that m is $O(n)$. In fact, consider an N-bounded parsing tree for a problem P of the class. Then, each rule may have at most N variables, thus it will contain at most 2^N locations, and there are at most n rules. Thus the total number of locations in the parsing is $n \times 2^N$. Also, since a parsing tree for P covers P, the number of locations is either the same or greater than the number of constraints in P. Thus $m \leq (n \times 2^N)$. Therefore we have that m is $O(n)$.

In order to find a parsing tree for a given SCSP, one could adapt the studies on the *secondary problem* in dynamic programming (see for example [28]). In some cases, however, the problem is generated in a way that the parsing tree is already explicit, like in the case of constraint logic programming (CLP) languages [126]. In fact, during the execution of a CLP program, the use of the clauses of the program builds a constraint problem with a parsing tree where each node of the tree directly corresponds to one of the used clauses.

A characterization of what dynamic programming is and when it can be applied, which is similar to the one given in this section, can be found in [178, 179]. There, *valuation-based systems* are defined as systems based on variables, valuations, and two operations, called *combination* and *marginalization.* These two operations are very related, respectively, to our notions of *combination* and *projection.* In valuation-based systems, three axioms are required for the correct application of a dynamic programming algorithm. These three axioms are indeed satisfied by our framework as well. In fact, they correspond, respectively, to 1) commutativity and associativity of $\otimes$ (see comment after Definition 2.2.5), 2) Theorem 2.2.1, and 3) Theorem 2.2.2. Note that the proof of Theorem 3.6.1 relies just on these three properties. In fact, our dynamic programming algorithm, defined in Definition 3.6.2, can be seen as an extension of the *fusion algorithm* defined in [178,179], where the extension consists in the fact that more than one variable at a time can be eliminated. In fact, in our algorithm all variables in the right-hand side of a rule are considered at a time.

Moreover, a related algorithm that solves a constraint problem with a tree-like shape in a bottom-up way, as in Definition 3.6.2, has been described in [84] for optimization problems and in [83] for belief maintenance. In those papers, the idea is to consider either the constraint graph, if it is acyclic, or the dual graph of a constraint problem (where nodes are constraints and arcs are associated with variables shared among constraints), and to use techniques like cycle-cutset [85] or tree-clustering [86] to provide such a dual graph with a tree-like shape. In [84], however, constraints are combined via the usual *and* operator, and the value associated to each tuple is computed in a completely independent way, via a given utility function. Thus, apart from the tree-structure, our approach and that in [84] are very different, since we also generalize the way constraints are combined, via a general notion of combination and projection, of which *and* and *or* are just instances.

Our notion of parsing tree is more general than that of *hinge-trees* presented in [119]. In fact, we do not assume anything about the structure of each node of the tree, which may be a generic graph. On the contrary, in [119] they consider only nodes that cannot be further decomposed into hinge-trees.

3.7 Conclusions

In this chapter we have generalized the classical solution and preprocessing techniques used for crisp CSPs so they could also be fruitfully used in SCSPs. In particular, we have shown that sufficient conditions for the effective applicability of local consistency and of dynamic programming algorithms could be checked just looking at the semiring properties. We have reviewed in this sense the different non-crisp frameworks and we have given a detailed description of the applicability of these techniques. Sufficient conditions for the termination of these (and more general) techniques have been given by just looking at the partial order structure represented by the constraint system or by the semiring.

Moreover, we have described some methodologies that could be applied to speed up the search for solutions (by applying labeling and/or partial local consistency) and some topological properties of the constraint graph that have to be checked to perform special cuts in the search tree.

The techniques of local consistency (or partial local consistency) will also be used in the next chapter where SCSPs' abstraction will be introduced. The idea is that if we are dealing with a framework where the local consistency steps cannot be safely applied (like in the WCSPs) we can, nonetheless, reduce the search space. In fact, we can abstract the problem, perform safe local consistency in the abstract framework, and then take back the collected information over the concrete problem.

Moreover, these same techniques will be the object of the studies presented in Chapter 5. There, in fact, a small language will be introduced and each of the steps used in the local consistency techniques will be seen as a function plus an assignment operation that will change the semiring value associated with the problem.

4. SCSP Abstraction

(Abstraction) The white cross, 1922
Wassily Kandinsky

Overview

We propose an abstraction scheme for soft constraint problems and we study its main properties. Processing the abstracted version of a soft constraint problem can help us in many ways: for example, to find good approximations of the optimal solutions, or also to provide us with information that can make the subsequent search for the best solution easier. Moreover, we show how the abstraction framework can be used to import constraint propagation algorithms from the abstract scenario to the concrete one. This may be useful when we don't have any (or any efficient) propagation algorithm in the concrete setting.

Although it is obvious that SCSPs are much more expressive than classical CSPs, they are also more difficult to process and to solve. Therefore, sometimes it may be too costly to find all, or even only one, optimal solution. Also, although classical propagation techniques, like arc-consistency [138], can be extended to SCSPs (see Chapter 3), even such techniques can be too costly to be used, depending on the size and structure of the partial order associated to the SCSP.

For these reasons, it may be reasonable to work on a simplified version of the given problem, trying, however, to not lose too much information. We propose to define this simplified version by means of the notion of abstraction, which takes an SCSP and returns a new one that is simpler to solve. Here, as in many other works on abstraction [191,192], "simpler" may mean many things, like the fact that a certain solution algorithm finds a solution, or an optimal solution, in a fewer number of steps, or also that the abstracted problem can be processed by machinery that is not available in the concrete context.

There are many formal proposals to describe the process of abstracting a notion, be it a formula, or a problem [192], or even a classical [106] or a soft CSP [174]. Among these, we chose to use one based on Galois insertions [77], mainly to refer to a well-know theory, with many results and properties that can be useful for our purposes. This made our approach compatible with the general theory of abstraction in [192]. Then, we adapted it to work on soft constraints: given an SCSP (the *concrete* one), we get an abstract SCSP by just changing the associated semiring, and relating the two structures (the concrete and the abstract one) via a *Galois insertion*. Note that this way of abstracting constraint problems does not change the structure of the problem (the set of variables remains the same, as well as the set of constraints), but just the semiring values to be associated to the tuples of values for the variables in each constraint.

Once we get the abstracted version of a given problem, we propose to

1. process the abstracted version: this may mean either solving it completely, or also applying some incomplete solver, which may derive some useful information from the abstract problem;
2. bring back to the original problem some (or possibly all) of the information gained in the abstract context;
3. continue the solution process on the transformed problem, which is a concrete problem equivalent to the given one.

All this process has the main aim of finding an optimal solution, or an approximation of it, for the original SCSP, within the resource bounds we have. The hope is that, by following the above three steps, we get to the final goal faster than just solving the original problem.

A deep study of the relationship between the concrete SCSP and the corresponding abstract one allows us to prove some results that can help in deriving useful information from the abstract problem and then take some of the derived information back to the concrete problem. In particular, we can prove the following:

- If the abstraction satisfies a certain property, all optimal solutions of the concrete SCSP are also optimal in the corresponding abstract SCSP (see Theorem 4.3.2). Thus, in order to find an optimal solution of the concrete problem, we could find all the optimal solutions of the abstract problem, and then just check their optimality on the concrete SCSP.
- Given any optimal solution of the abstract problem, we can find upper and lower bounds for an optimal solution for the concrete problem (see Theorem 4.3.3). If we are satisfied with these bounds, we could just take the optimal solution of the abstract problem as a reasonable approximation of an optimal solution for the concrete problem.
- If we apply some constraint propagation technique over the abstract problem, say P, obtaining a new abstract problem, say P', some of the information in P' can be inserted into P, obtaining a new concrete problem, which is closer to its solution and thus easier to solve (see Theorem 4.3.4 and 4.3.6). This, however, can be done only if the semiring operation that describes how to combine constraints on the concrete side is idempotent (see Theorem 4.3.4).
- If instead this operation is not idempotent, we can still bring back some information from the abstract side. In particular, we can bring back the inconsistencies (that is, tuples associated with the worst element of the semiring), since we are sure that these same tuples are also inconsistent also in the concrete SCSP (see Theorem 4.3.6).

In both the last two cases, the new concrete problem is easier to solve, in the sense, for example, that a branch-and-bound algorithm would explore a smaller (or equal) search tree before finding an optimal solution.

The results of this chapter, already appeared in [33,35,36].

4.1 Abstraction

Abstract interpretation [30,77,78] is a theory developed to reason about the relation between two different semantics (the *concrete* and the *abstract* semantics). The idea of approximating program properties by evaluating a program on a simpler domain of descriptions of "concrete" program states goes back to the early' 70's. The inspiration was that of approximating properties from the exact (concrete) semantics into an approximate (abstract) semantics, that explicitly exhibits a structure (e.g., ordering) which is somehow present in the richer concrete structure associated to program execution.

The guiding idea is to relate the concrete and the abstract interpretation of the calculus by a pair of functions, the *abstraction* function α and the *concretization* function γ, which form a Galois connection.

Let $(\mathcal{C}, \sqsubseteq)$ (concrete domain) be the domain of the concrete semantics, while $(\mathcal{A}, \leq)$ (abstract domain) be the domain of the abstract semantics. The partial order relations reflect an approximation relation. Since in approximation theory a partial order specifies the precision degree of any element in a poset, it is obvious to assume that if α is a mapping associating an abstract object in $(\mathcal{A}, \leq)$ for any concrete element in $(\mathcal{C}, \sqsubseteq)$, then the following holds: if $\alpha(x) \leq y$, then y is also a correct, although less precise, abstract approximation of x. The same argument holds if $x \sqsubseteq \gamma(y)$. Then y is also a correct approximation of x, although x provides more accurate information than $\gamma(y)$. This gives rise to the following formal definition.

Definition 4.1.1 (Galois insertion). *Let* $(\mathcal{C}, \sqsubseteq)$ *and* $(\mathcal{A}, \leq)$ *be two posets (the concrete and the abstract domain). A Galois connection* $\langle \alpha, \gamma \rangle : (C, \sqsubseteq) \rightleftharpoons (A, \leq)$ *is a pair of maps* $\alpha : \mathcal{C} \to \mathcal{A}$ *and* $\gamma : \mathcal{A} \to \mathcal{C}$ *such that*

1. α *and* γ *are monotonic,*
2. *for each* $x \in \mathcal{C}, x \sqsubseteq \gamma(\alpha(x))$ *and*
3. *for each* $y \in \mathcal{A}, \alpha(\gamma(y)) \leq y$.

Moreover, a Galois insertion (of $\mathcal{A}$ *in* $\mathcal{C}$*)* $\langle \alpha, \gamma \rangle : (\mathcal{C}, \sqsubseteq) \rightleftharpoons (\mathcal{A}, \leq)$ *is a Galois connection where* $\gamma \cdot \alpha = Id_{\mathcal{A}}$.

Property 2 is called *extensivity* of $\alpha \cdot \gamma$. The map α (γ) is called the *lower* (*upper*) *adjoint* or *abstraction* (*concretization*) in the context of abstract interpretation.

The following basic properties are satisfied by any Galois insertion:

1. γ is injective and α is surjective.
2. $\alpha \cdot \gamma$ is an upper closure operator in $(\mathcal{C}, \sqsubseteq)$.
3. α is additive and γ is co-additive.
4. Upper and lower adjoints uniquely determine each other. Namely,

$$\gamma = \lambda y. \bigsqcup_{\mathcal{C}} \{x \in \mathcal{C} \mid \alpha(x) \sqsubseteq y\}, \alpha = \lambda x. \bigsqcap_{\mathcal{A}} \{y \in \mathcal{A} \mid x \leq \gamma(y)$$

5. α is an isomorphism from $(\gamma\alpha)(\mathcal{C})$ to $\mathcal{A}$, having γ as its inverse.

An example of a Galois insertion can be seen in Figure 4.1. Here, the concrete lattice is $\langle [0, 1], \leq \rangle$, and the abstract one is $\langle \{0, 1\}, \leq \rangle$. Function α maps all real numbers in $[0, 0.5]$ into 0, and all other integers (in $(0.5, 1]$) into 1. Function γ maps 0 into 0.5 and 1 into 1.

One property that will be useful later relates to a precise relationship between the ordering in the concrete lattice and that in the abstract one.

Theorem 4.1.1 (total ordering). *Consider a Galois insertion from* $(\mathcal{C}, \sqsubseteq)$ *to* $(\mathcal{A}, \leq)$. *Then, if* $\sqsubseteq$ *is a total order, also* $\leq$ *is so.*

Proof. It easily follows from the monotonicity of α (that is, $x \sqsubseteq y$ implies $\alpha(x) \leq \alpha(y)$, and from its surjectivity (that is, there is no element in $\mathcal{A}$ which is not the image of some element in $\mathcal{C}$ via α).

Usually, both the concrete and the abstract lattice have some operators that are used to define the corresponding semantics. Most of the times it is useful, and required, that the abstract operators show a certain relationship with the corresponding concrete ones. This relationship is called *local correctness*.

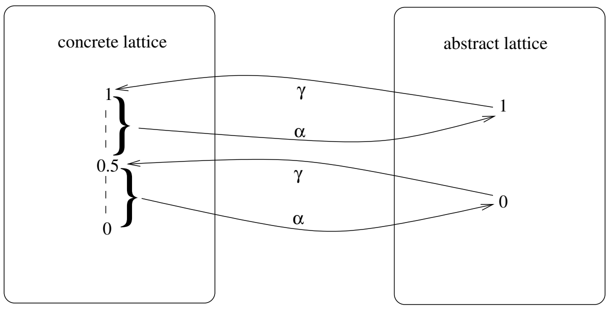


Fig. 4.1. A Galois insertion

Definition 4.1.2 (local correctness). *Let $f : \mathcal{C}^n \to \mathcal{C}$ be an operator over the concrete lattice, and assume that $\tilde{f}$ is its abstract counterpart. Then $\tilde{f}$ is locally correct w.r.t. f if $\forall x_1, \ldots, x_n \in \mathcal{C}. f(x_1, \ldots, x_n) \sqsubseteq \gamma(\tilde{f}(\alpha(x_1), \ldots, \alpha(x_n)))$.*

4.2 Abstracting Soft CSPs

Given the notions of soft constraints and abstraction, recalled in the previous sections, we now want to show how to abstract soft constraint problems. The main idea is very simple: we just want to pass, via the abstraction, from an SCSP P over a certain semiring S to another SCSP $\tilde{P}$ over the semiring $\tilde{S}$, where the lattices associated to $\tilde{S}$ and S are related by a Galois insertion as shown above.

Definition 4.2.1 (abstracting SCSPs). *Consider the* concrete *SCSP $P = \langle C, con \rangle$ over semiring S, where*

- $S = \langle A, +, \times, \mathbf{0}, \mathbf{1} \rangle$ *and*
- $C = \{c_0, \ldots, c_n\}$ *with* $c_i = \langle con_i, def_i \rangle$ *and* $def_i : D^{|con_i|} \to A$;

we define the abstract *SCSP $\tilde{P} = \langle \tilde{C}, con \rangle$ over the semiring $\tilde{S}$, where*

- $\tilde{S} = \langle \tilde{A}, \tilde{+}, \tilde{\times}, \tilde{\mathbf{0}}, \tilde{1} \rangle$;
- $\tilde{C} = \{\tilde{c_0}, \ldots, \tilde{c_n}\}$ *with* $\tilde{c_i} = \langle con_i, \tilde{def_i} \rangle$ *and* $\tilde{def_i} : D^{|con_i|} \to \tilde{A}$;
- *if $L = \langle A, \leq \rangle$ is the lattice associated to S and $\tilde{L} = \langle \tilde{A}, \tilde{\leq} \rangle$ the lattice associated to $\tilde{S}$, then there is a Galois insertion $\langle \alpha, \gamma \rangle$ such that $\alpha : L \to \tilde{L}$;*
- *$\tilde{\times}$ is locally correct with respect to $\times$.*

Notice that the kind of abstraction we consider in this chapter does not change the structure of the SCSP. That is, C and $\tilde{C}$ have the same number of constraints, and c_i and $\tilde{c_i}$ involve the same variables. The only thing that is changed by abstracting an SCSP is the semiring. Thus, P and $\tilde{P}$ have the same graph topology (variables and constraints), but different constraint definitions, since if a certain tuple of domain values in a constraint of P has semiring value a, then the same tuple in the same constraint of $\tilde{P}$ has semiring value $\alpha(a)$. Notice also that α and γ can be defined in several different ways, but all of them have to satisfy the properties of the Galois insertion, from which it derives, among others, that $\alpha(\mathbf{0}) = \tilde{\mathbf{0}}$ and $\alpha(\mathbf{1}) = \tilde{\mathbf{1}}$.

Example 4.2.1. As an example, consider any SCSP over the semiring for optimization

$$S_{WCSP} = \langle \Re^- \cup \{-\infty\}, \max, +, -\infty, 0\rangle$$

(where costs are represented by negative reals) and suppose we want to abstract it onto the semiring for fuzzy reasoning

$$S_{FCSP} = \langle [0,1], max, min, \mathbf{0}, \mathbf{1}\rangle.$$

In other words, instead of computing the maximum of the sum of all costs, we just want to compute the maximum of the minimum of all costs, and we want to normalize the costs over [0..1]. Notice that the abstract problem is in the FCSP class and it has an idempotent $\times$ operator (which is the min). This means that in the abstract framework we can perform local consistency over the problem in order to find inconsistencies. As noted above, the mapping $\alpha : \langle \Re^-, \leq_{WCSP} \rangle \rightarrow \langle [0,1], \leq_{FCSP}\rangle$ can be defined in different ways. For example one can decide to map all the reals below some fixed real x onto 0 and then to map the reals in $[x, 0]$ into the reals in $[0, 1]$ by using a normalization function, for example $f(r) = \frac{x-r}{x}$.

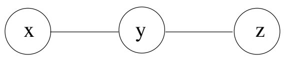
Example 4.2.2. Another example is the abstraction from the fuzzy semiring to the classical one:

$$S_{CSP} = \langle \{0,1\}, \vee, \wedge, 0, 1\rangle.$$

Here function α maps each element of $[0, 1]$ into either 0 or 1. For example, one could map all the elements in $[0, x]$ onto 0, and all those in $(x, 1]$ onto 1, for some fixed x. Figure 4.1 represents this example with $x = 0.5$.

In Figure 4.2 we show an example of fuzzy CSP and its solutions.

We have defined Galois insertions between two lattices $\langle L, \leq_S\rangle$ and $\langle \tilde{L}, \tilde{\leq}_{\tilde{S}}\rangle$ of semiring values. For convenience, however, in the following we will often use



a a ... 0	a a ... 0.3
a b ... 1	a b ... 0.1
b a ... 0.5	b a ... 1
b b ... 0.7	b b ... 1

Solutions:

a a a ... min(0,0.3) = 0
a a b ... min(0,0.1) = 0
a b a ... min(1,1) = 1
a b b ... min(1,1) = 1
b a a ... min(0.5,0.3) = 0.3
b a b ... min(0.5,0.1) = 0.1
b b a ... min(0.7,1) = 0.7
b b b ... min(0.7,1) = 0.7

Best solutions:
a b a ... 1
a b b ... 1

Fig. 4.2. A fuzzy CSP and its solutions

Galois insertions between lattices of problems $\langle PL, \sqsubseteq_S \rangle$ and $\langle \tilde{PL}, \tilde{\sqsubseteq}_{\tilde{S}} \rangle$ where PL contains problems over the concrete semiring and $\tilde{PL}$ over the abstract semiring. This does not change the meaning of our abstraction, we are just upgrading the Galois insertion from semiring values to problems. Thus, when we will say that $\tilde{P} = \alpha(P)$, it will mean that $\tilde{P}$ is obtained by P via the application of α to all the semiring values appearing in P.

An important property of our notion of abstraction is that the composition of two abstractions is still an abstraction. This allows to build a complex abstraction by defining several simpler abstractions to be composed.

Theorem 4.2.1 (abstraction composition). *Consider an abstraction from the lattice corresponding to a semiring S_1 to that corresponding to a semiring S_2, denoted by the pair $\langle \alpha_1, \gamma_1 \rangle$. Consider now another abstraction from the lattice corresponding to the semiring S_2 to that corresponding to a semiring S_3, denoted by the pair $\langle \alpha_2, \gamma_2 \rangle$. Then the pair $\langle \alpha_1 \cdot \alpha_2, \gamma_2 \cdot \gamma_1 \rangle$ is an abstraction as well.*

Proof. We first have to prove that $\langle \alpha, \gamma \rangle = \langle \alpha_1 \cdot \alpha_2, \gamma_2 \cdot \gamma_1 \rangle$ satisfies the four properties of a Galois insertion:

– since the composition of monotone functions is again a monotone function, we have that both α and γ are monotone functions;
– given a value $x \in S_1$, from the first abstraction we have that $x \sqsubseteq_1 \gamma_1(\alpha_1(x))$. Moreover, for any element $y \in S_2$, we have that $y \sqsubseteq_2 \gamma_2(\alpha_2(y))$. This holds also for $y = \alpha_1(x)$, thus by monotonicity of γ_1 we have $x \sqsubseteq_1 \gamma_1(\gamma_2(\alpha_2(\alpha_1(x))))$.
– a similar proof can be used for the third property;
– the composition of two identities is still an identity .

To prove that $\times_3$ is locally correct w.r.t. $\times_1$, it is enough to consider the local correctness of $\times_2$ w.r.t. $\times_1$ and of $\times_3$ w.r.t. $\times_2$, and the monotonicity of γ_1.

4.3 Properties and Advantages of the Abstraction

In this section we will define and prove several properties that hold on abstractions of soft constraint problems. The main goal here is to point out some of the advantages that one can have in passing through the abstracted version of a soft constraint problem instead of directly solving the concrete version.

4.3.1 Relating a Soft Constraint Problem and Its Abstract Version

Let us consider the scheme depicted in Figure 4.3. Here, and in the following pictures, the left box contains the lattice of concrete problems, and the right one the lattice of abstract problems. The partial order in each of these lattices is shown via dashed lines. Connections between the two lattices, via the abstraction and concretization functions, is shown via directed arrows. In the following, we will call $S = \langle A, +, \times, \mathbf{0}, \mathbf{1} \rangle$ the concrete semiring and $\tilde{S} = \langle \tilde{A}, \tilde{+}, \tilde{\times}, \tilde{\mathbf{0}}, \tilde{\mathbf{1}} \rangle$ the abstract one. Thus, we will always consider a Galois insertion $\langle \alpha, \gamma \rangle : \langle A, \leq_S \rangle \rightleftharpoons \langle \tilde{A}, \leq_{\tilde{S}} \rangle$.

In Figure 4.3, P is the starting SCSP. Then with the mapping α we get $\tilde{P} = \alpha(P)$, which is an abstraction of P. By applying the mapping γ to $\tilde{P}$, we get the problem $\gamma(\alpha(P))$. Let us first notice that these two problems (P and $\gamma(\alpha(P))$) are related by a precise property, as stated by the following theorem.

Theorem 4.3.1. *Given an SCSP P over S, we have that $P \sqsubseteq_S \gamma(\alpha(P))$.*

Proof. Immediately follows from the properties of a Galois insertion, in particular from the fact that $x \leq_S \gamma(\alpha(x))$ for any x in the concrete lattice. In fact, $P \sqsubseteq_S \gamma(\alpha(P))$ means that, for each tuple in each constraint of P, the semiring value associated to such a tuple in P is smaller (w.r.t. $\leq_S$) than the corresponding value associated to the same tuple in $\gamma(\alpha(P))$.

Notice that this implies that, if a tuple in $\gamma(\alpha(P))$ has semiring value $\mathbf{0}$, then it must have value $\mathbf{0}$ also in P. This holds also for the solutions, whose semiring value is obtained by combining the semiring values of several tuples.

Corollary 4.3.1. *Given an SCSP P over S, we have that $Sol(P) \sqsubseteq_S Sol(\gamma(\alpha(P))$.*

Proof. We recall that $Sol(P)$ is just a constraint, which is obtained as $\bigotimes(C) \Downarrow_{con}$. Thus the statement of this corollary comes from the monotonicity of $\times$ and $+$.

Therefore, by passing from P to $\gamma(\alpha(P))$, no new inconsistencies are introduced: if a solution of $\gamma(\alpha(P))$ has value $\mathbf{0}$, then this was true also in P. It is possible, however, that some inconsistencies are forgotten (that is, they appear to be consistent after the abstraction process).

Example 4.3.1. Consider the abstraction from the fuzzy to the classical semiring, as described in Figure 4.1. Then, if we call P the fuzzy problem in Figure 4.2,

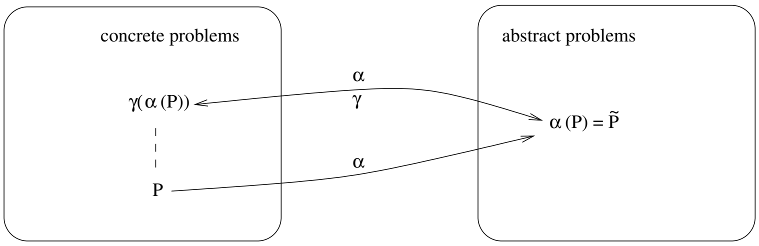


Fig. 4.3. The concrete and the abstract problem

Figure 4.4 shows the concrete problem P, the abstract problem $\alpha(P)$, and its concretization $\gamma(\alpha(P))$. It is easy too see that, for each tuple in each constraint, the associated semiring value in P is lower than or equal to that in $\gamma(\alpha(P))$.

If the abstraction preserves the semiring ordering (that is, applying the abstraction function and then combining gives elements that are in the same ordering as the elements obtained by combining only), then there is also an interesting relationship between the set of optimal solutions of P and that of $\alpha(P)$. In fact, if a certain tuple is optimal in P, then this same tuple is also optimal in $\alpha(P)$. Let us first investigate the meaning of the *order-preserving* property.

Definition 4.3.1 (order-preserving abstraction). *Consider two sets I_1 and I_2 of concrete elements. Then an abstraction is said to be order-preserving if*

$$\tilde{\prod_{x \in I_1}} \alpha(x) \leq_{\tilde{S}} \tilde{\prod_{x \in I_2}} \alpha(x) \Rightarrow \prod_{x \in I_1} x \leq_S \prod_{x \in I_2} x$$

where the products refer to the multiplicative operations of the concrete ($\prod$) and the abstract ($\tilde{\prod}$) semirings.

In words, this notion of order-preservation means that if we first abstract and then combine, or we combine only, we get the same ordering (but on different semirings) among the resulting elements.

Example 4.3.2. An abstraction that is not order-preserving can be seen in Figure 4.5. Here, the concrete and the abstract sets, as well as the additive operations of the two semirings, can be seen from the picture. For the multiplicative operations, we assume they coincide with the glb of the two semirings.

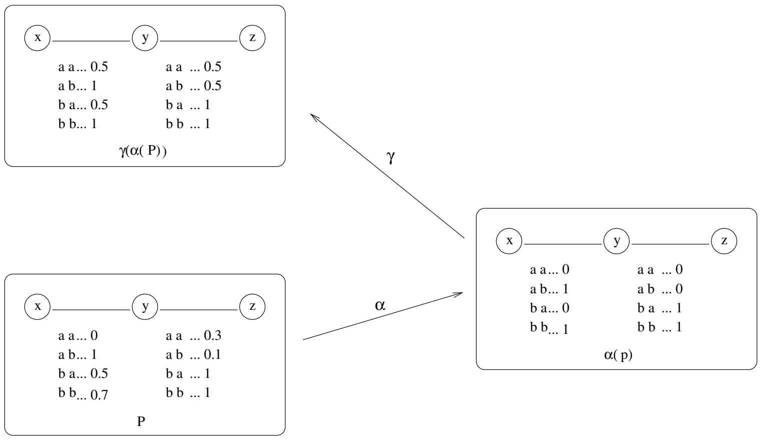


Fig. 4.4. An example of the abstraction fuzzy-classical

In this case, the concrete ordering is partial, while the abstract ordering is total. Functions α and β are depicted in the figure by arrows going from the concrete semiring to the abstract one and vice versa. Assume that the concrete problem has no solution with value 1. Then all solutions with value a or b are optimal. Suppose a solution with value b is obtained by computing $b = 1 \times b$, while we have $a = 1 \times a$. Then the abstract counterparts will have values $\alpha(1) \times' \alpha(b) = 1 \times' 0 = 0$ and $\alpha(1) \times' \alpha(a) = 1 \times' 1 = 1$. Therefore, the solution with value a, which is optimal in the concrete problem, is not optimal anymore in the abstract problem.

Example 4.3.3. The abstraction in Figure 4.1 is order-preserving. In fact, consider two abstract values that are ordered, that is $0 \leq' 1$. Then $1 = 1 \times' 1 = \alpha(x) \times' \alpha(y)$, where both x and y must be greater than 0.5. Thus their concrete combination (which is the min), say v, is always greater than 0.5. On the other hand, 0 can be obtained by combining either two 0's (therefore the images of two elements smaller than or equal to 0.5, whose minimum is smaller than 0.5 and thus smaller than v), or by combining a 0 and a 1, which are images of a value greater than 0.5 and one smaller than 0.5. Also in this case, their combination (the min) is smaller than 0.5 and thus smaller than v. Thus the order-preserving property holds.

Example 4.3.4. Another abstraction that is not order-preserving is the one that passes from the semiring $\langle N \cup \{+\infty\}, min, sum, 0, +\infty \rangle$, where we minimize the sum of values over the naturals, to the semiring $\langle N \cup \{+\infty\}, min, max, 0, +\infty \rangle$, where we minimize the maximum of values over the naturals. In words, this abstraction maintains the same domain of the semiring, and the same additive operation (min), but it changes the multiplicative operation (which passes from sum to max). Notice that the semiring orderings are the opposite as those usually used over the naturals: if i is smaller than j then $j \leq_S i$, thus, the best element is 0 and the worst is $+\infty$. The abstraction function α is just the identity, and also the concretization function γ.

In this case, consider in the abstract semiring two values and the way they are obtained by combining other two values of the abstract semiring: for example, $8 = max(7, 8)$ and $9 = max(1, 9)$. In the abstract ordering, 8 is higher than 9.

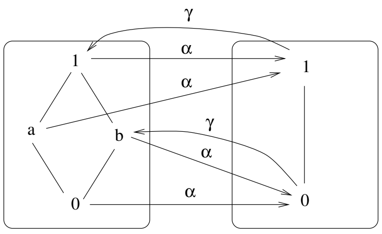


Fig. 4.5. An abstraction which is not order-preserving

Then, let us see how the images of the combined values (the same values, since α is the identity) relate to each other: we have $sum(7,8) = 15$ and $sum(1,9) = 10$, and 15 is lower than 10 in the concrete ordering. Thus the order-preserving property does not hold.

Notice that, if we reduce the sets I_1 and I_2 to singletons, say x and y, then the order-preserving property says that $\alpha(x) \leq_{\tilde{S}} \alpha(y)$ implies that $x \leq_S y$. This means that if two abstract objects are ordered, then their concrete counterparts will be ordered as well, and in the same way. Of course they could never be ordered in the opposite sense, otherwise α would not be monotonic; but they could be incomparable. Therefore, if we choose an abstraction where incomparable objects are mapped by α onto ordered objects, then we do not have the order-preserving property. A consequence of this is that if the abstract semiring is totally ordered, and we want an order-preserving abstraction, then the concrete semiring will be totally ordered as well.

On the other hand, if two abstract objects are not ordered, then the corresponding concrete objects can be ordered in any sense, or they can also be not comparable. Notice that this restriction of the order-preserving property to singleton sets always holds when the concrete ordering is total. In fact, in this case, if two abstract elements are ordered in a certain way, then it is impossible that the corresponding concrete elements are ordered in the opposite way, because, as we said above, of the monotonicity of the α function.

Theorem 4.3.2. *Consider an abstraction that is order-preserving. Given an SCSP P over S, we have that* $Opt(P) \subseteq Opt(\alpha(P))$.

Proof. Let us take a tuple t that is optimal in the concrete semiring S, with value v. Then v has been obtained by multiplying the values of some subtuples. Suppose, without loss of generality, that the number of such subtuples is two (that is, we have two constraints): $v = v_1 \times v_2$. Let us then take the value of this tuple in the abstract problem, that is, the abstract combination of the abstractions of v_1 and v_2: this is $v' = \alpha(v_1) \times' \alpha(v_2)$. We have to show that if v is optimal, then also v' is optimal.

Suppose then that v' is not optimal, that is, there exists another tuple t'' with value v'' such that $v' \leq_{S'} v''$. Assume $v'' = v_1'' \times' v_2''$. Now let us see the value of tuple t'' in P. If we set $v_i'' = \alpha(\bar{v}_i)$, then we have that this value is $\bar{v} = \bar{v}_1 \times \bar{v}_2$. Let us now compare v with $\bar{v}$. Since $v' \leq_{\tilde{S}} v''$, by order-preservation we get that $v \leq_S \bar{v}$. But this means that v is not optimal, which was our initial assumption. Therefore v' has to be optimal.

Therefore, in case of order-preservation, the set of optimal solutions of the abstract problem contains all the optimal solutions of the concrete problem. In other words, it is not allowed that an optimal solution in the concrete domain becomes non-optimal in the abstract domain. Some non-optimal solutions, however, could become optimal by becoming incomparable with the optimal solutions.

Example 4.3.5. Consider again the previous example. The optimal solutions in P are the tuples $\langle a, b, a\rangle$ and $\langle a, b, b\rangle$. It is easy to see that these tuples are also optimal in $\alpha(P)$. In fact, this is a classical constraint problem where the solutions are tuples $\langle a, b, a\rangle$, $\langle a, b, b\rangle$, $\langle b, b, a\rangle$, and $\langle b, b, b\rangle$.

Thus, if we want to find an optimal solution of the concrete problem, we could find all the optimal solutions of the abstract problem, and then use them on the concrete side to find an optimal solution for the concrete problem. Assuming that working on the abstract side is easier than on the concrete side, this method could help us find an optimal solution of the concrete problem by looking at just a subset of tuples in the concrete problem.

Another important property, which holds for any abstraction, concerns computing bounds that approximate an optimal solution of a concrete problem. In fact, any optimal solution, say t, of the abstract problem, say with value $\tilde{v}$, can be used to obtain both an upper and a lower bound of an optimum in P. In fact, we can prove that there is an optimal solution in P with value between $\gamma(\tilde{v})$ and the value of t in P. Thus, if we think that approximating the optimal value with a value within these two bounds is satisfactory, we can take t as an approximation of an optimal solution of P.

Theorem 4.3.3. *Given an SCSP P over S, consider an optimal solution of $\alpha(P)$, say t, with semiring value $\tilde{v}$ in $\alpha(P)$ and v in P. Then there exists an optimal solution $\bar{t}$ of P, say with value $\bar{v}$, such that $v \leq \bar{v} \leq \gamma(\tilde{v})$.*

Proof. By local correctness of the multiplicative operation of the abstract semiring, we have that $v \leq_S \gamma(\tilde{v})$. Since v is the value of t in P, either t itself is optimal in P, or there is another tuple that has value better than v, say $\bar{v}$. We will now show that $\bar{v}$ cannot be greater than $\gamma(\tilde{v})$.

In fact, assume by absurd that $\gamma(\tilde{v}) \leq_S \bar{v}$. Then, by local correctness of the multiplicative operation of the abstract semiring, we have that $\alpha(\bar{v})$ is smaller than the value of $\bar{t}$ in $\alpha(P)$. Also, by monotonicity of α, by $\gamma(\tilde{v}) \leq_S \bar{v}$ we get that $\tilde{v} \leq_{\tilde{S}} \alpha(\bar{v})$. Therefore, by transitivity we obtain that $\tilde{v}$ is smaller than the value of $\bar{t}$ in $\alpha(P)$, which is not possible because we assumed that $\tilde{v}$ was optimal.

Therefore, there must be an optimal value between v and $\gamma(\tilde{v})$.

Thus, given a tuple t with optimal value $\tilde{v}$ in the abstract problem, instead of spending time to compute an exact optimum of P, we can do the following:

- compute $\gamma(\tilde{v})$, thus obtaining an upper bound of an optimum of P;
- compute the value of t in P, which is a lower bound of the same optimum of P;
- If we think that such bounds are close enough, we can take t as a reasonable approximation (to be precise, a lower bound) of an optimum of P.

Notice that this theorem does not need the order-preserving property in the abstraction, thus any abstraction can exploit its result.

Example 4.3.6. Consider again the previous example. Now take any optimal solution of $\alpha(P)$, for example tuple $\langle b, b, b\rangle$. Then the above result states that there exists an optimal solution of P with semiring value v between the value of this tuple in P, which is 0.7, and $\gamma(1) = 1$. In fact, there are optimal solutions with value 1 in P.

4.3.2 Working on the Abstract Problem

Consider now what we can do on the abstract problem, $\alpha(P)$. One possibility is to apply an abstract function $\tilde{f}$, which can be, for example, a local consistency algorithm or also a solution algorithm. In the following, we will consider functions $\tilde{f}$, which are always intensive, that is, which bring the given problem closer to the bottom of the lattice. In fact, our goal is to solve an SCSP, thus going higher in the lattice does not help in this task, since solving means combining constraints and thus getting lower in the lattice. Also, functions $\tilde{f}$ will always be locally correct with respect to any function f_{sol} that solves the concrete problem. We will call such a property *solution-correctness.* More precisely, given a problem P with constraint set C, $f_{sol}(P)$ is a new problem P' with the same topology as P whose tuples have semiring values possibly lower. Let us call C' the set of constraints of P'. Then, for any constraint $c' \in C'$, $c' = (\bigotimes C) \Downarrow_{var(c')}$. In other words, f_{sol} combines all constraints of P and then projects the resulting global constraint over each of the original constraints.

Definition 4.3.2. *Given an SCSP P over S, consider a function $\tilde{f}$ on $\alpha(P)$. Then $\tilde{f}$ is solution-correct if, given any f_{sol} which solves P, $\tilde{f}$ is locally correct w.r.t. f_{sol}.*

We will also need the notion of safeness of a function, which just means that it maintains all the solutions.

Definition 4.3.3. *Given an SCSP P and a function $f : PL \to PL$, f is safe if $Sol(P) = Sol(f(P))$.*

It is easy to see that any local consistency algorithm, as defined in [47], can be seen as a safe, intensive, and solution-correct function.

From $\tilde{f}(\alpha(P))$, applying the concretization function γ, we get $\gamma(\tilde{f}(\alpha(P)))$, which therefore is again over the concrete semiring (the same as P). The following property says that, under certain conditions, P and $P \otimes \gamma(\tilde{f}(\alpha(P)))$ are equivalent. Figure 4.6 describes such a situation. In this figure, we can see that several partial order lines have been drawn:

- on the abstract side, function $\tilde{f}$ takes any element closer to the bottom, because of its intensiveness;
- on the concrete side, we have that
  - $P \otimes \gamma(\tilde{f}(\alpha(P)))$ is smaller than both P and $\gamma(\tilde{f}(\alpha(P)))$ because of the properties of $\otimes$;
  - $\gamma(\tilde{f}(\alpha(P)))$ is smaller than $\gamma(\alpha(P))$ because of the monotonicity of γ;

- $\gamma(\tilde{f}(\alpha(P)))$ is higher than $f_{sol}(P)$ because of the solution-correctness of $\tilde{f}$;
- $f_{sol}(P)$ is smaller than P because of the way $f_{sol}(P)$ is constructed;
- if $\times$ is idempotent, then it coincides with the glb, thus we have that $P \otimes \gamma(\tilde{f}(\alpha(P)))$ is higher than $f_{sol}(P)$, because by definition the glb is the highest among all the lower bounds of P and $\gamma(\tilde{f}(\alpha(P)))$.

Theorem 4.3.4. *Given an SCSP P over S, consider a function $\tilde{f}$ on $\alpha(P)$ which is safe, solution-correct, and intensive. Then, if $\times$ is idempotent, $Sol(P) = Sol(P \otimes \gamma(\tilde{f}(\alpha(P))))$.*

Proof. Take any tuple t with value v in P, which is obtained by combining the values of some subtuples, say two: $v = v_1 \times v_2$. Let us now consider the abstract versions of v_1 and v_2: $\alpha(v_1)$ and $\alpha(v_2)$. Function $\tilde{f}$ changes these values by lowering them, thus we get $\tilde{f}(\alpha(v_1)) = v_1'$ and $\tilde{f}(\alpha(v_2)) = v_2'$.

Since $\tilde{f}$ is safe, we have that $v_1' \times' v_2' = \alpha(v_1) \times' \alpha(v_2) = v'$. Moreover, $\tilde{f}$ is solution-correct, thus $v \leq_S \gamma(v')$. By monotonicity of γ, we have that $\gamma(v') \leq_S \gamma(v_i')$ for $i = 1, 2$. This implies that $\gamma(v') \leq_S (\gamma(v_1') \times \gamma(v_2'))$, since $\times$ is idempotent by assumption and thus it coincides with the glb. Thus we have that $v \leq_S (\gamma(v_1') \times \gamma(v_2'))$.

To prove that P and $P \otimes \gamma(\tilde{f}(\alpha(P)))$ give the same value to each tuple, we now have to prove that $v = (v_1 \times \gamma(v_1')) \times (v_2 \times \gamma(v_2'))$. By commutativity of $\times$, we can write this as $(v_1 \times v_2) \times (\gamma(v_1') \times \gamma(v_2'))$. Now, $v_1 \times v_2 = v$ by assumption, and we have shown that $v \leq_S \gamma(v_1') \times \gamma(v_2')$. Therefore $v \times (\gamma(v_1') \times \gamma(v_2')) = v$.

This theorem does not say anything about the power of $\tilde{f}$, which could make many modifications to $\alpha(P)$, or it could also not modify anything. In this last case, $\gamma(\tilde{f}(\alpha(P))) = \gamma(\alpha(P)) \sqsupseteq P$ (see Figure 4.7), so $P \otimes \gamma(\tilde{f}(\alpha(P))) = P$, which

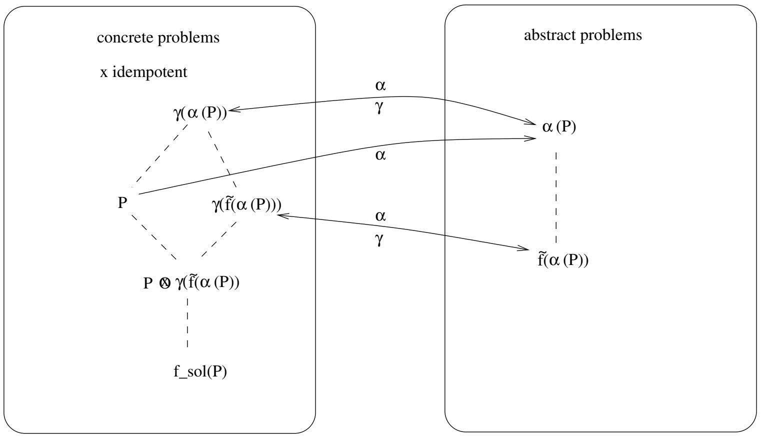


Fig. 4.6. The general abstraction scheme, with $\times$ idempotent

means that we have not gained anything in abstracting P. We can, however, always use the relationship between P and $\alpha(P)$ (see Theorem 4.3.2 and 4.3.3) to find an approximation of the optimal solutions and of the inconsistencies of P.

If instead $\tilde{f}$ modifies all semiring elements in $\alpha(P)$, then if the order of the concrete semiring is total, we have that $P \otimes \gamma(\tilde{f}(\alpha(P))) = \gamma(\tilde{f}(\alpha(P)))$ (see Figure 4.8), and thus we can work on $\gamma(\tilde{f}(\alpha(P)))$ to find the solutions of P. In fact, $\gamma(\tilde{f}(\alpha(P)))$ is lower than P and thus closer to the solution.

Theorem 4.3.5. *Given an SCSP P over S, consider a function $\tilde{f}$ on $\alpha(P)$ which is safe, solution-correct, and intensive. Then, if $\times$ is idempotent, $\tilde{f}$ modifies every semiring element in $\alpha(P)$, and the order of the concrete semiring is total, we have that $P \sqsupseteq_S \gamma(\tilde{f}(\alpha(P))) \sqsupseteq_S f_{sol}(P)$.*

Proof. Consider any tuple t in any constraint of P, and let us call v its semiring value in P and v_{sol} its value in $f_{sol}(P)$. Obviously, we have that $v_{sol} \leq_S v$. Now take $v' = \gamma(\alpha(v))$. By monotonicity of α, we cannot have $v <_S v'$. Also, by solution-correctness of $\tilde{f}$, we cannot have $v' <_S v_{sol}$. Thus, we must have $v_{sol} \leq_S v' \leq_s v$, which proves the statement of the theorem.

Example 4.3.7. Figure 4.9 uses the abstraction in Figure 4.1 and shows a concrete problem and the result of the construction of Figure 4.6 over it.

Notice that we need the idempotency of the $\times$ operator for Theorem 4.3.4 and 4.3.5. If instead $\times$ is not idempotent, then we can prove something weaker. Figure 4.10 shows this situation. With respect to Figure 4.6, we can see that the possible non-idempotency of $\times$ changes the partial order relationship on the concrete side. In particular, we do not have the problem $P \otimes \gamma(\tilde{f}(\alpha(P)))$ any more, nor the problem $f_{sol}(P)$, since these problems would not have the same solutions as P and thus are not interesting to us. We have instead a new problem P', which is constructed in such a way as to "insert" the inconsistencies of $\gamma(\tilde{f}(\alpha(P)))$ into P. P' is obviously lower than P in the concrete partial order, since it is the same as P with the exception of some more **0**'s, but the most important point is that it has the same solutions as P.

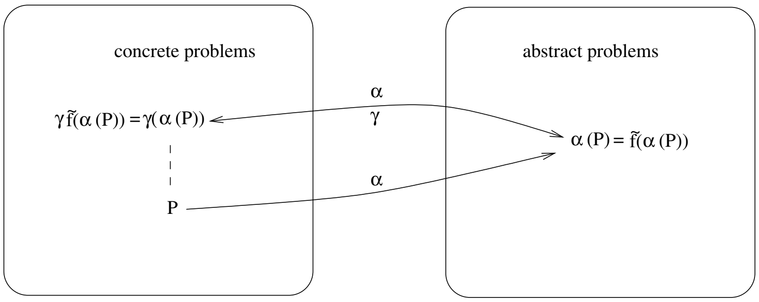


Fig. 4.7. The scheme when $\tilde{f}$ does not modify anything

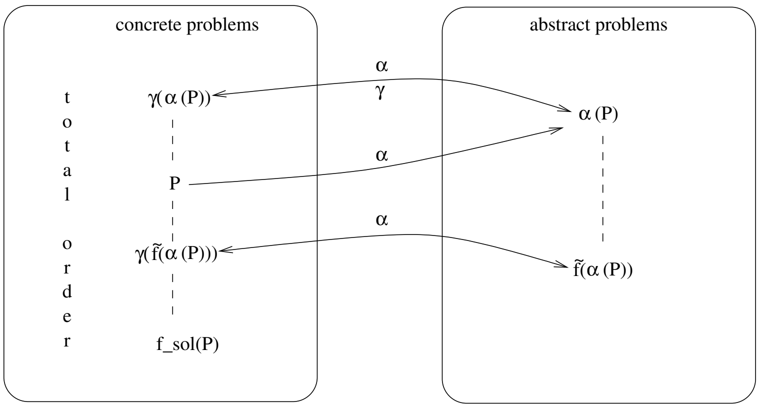


Fig. 4.8. The scheme when the concrete semiring has a total order

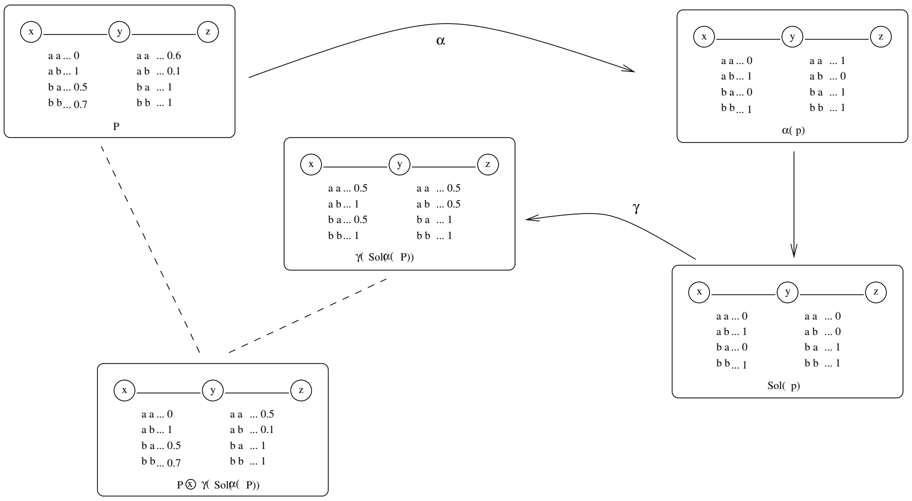


Fig. 4.9. An example with $\times$ idempotent

Theorem 4.3.6. *Given an SCSP P over S, consider a function $\tilde{f}$ on $\alpha(P)$ which is safe, solution-correct and intensive. Then, if $\times$ is not idempotent, consider P' to be the SCSP which is the same as P except for those tuples which have semiring value* **0** *in $\gamma(\alpha(\tilde{f}(P)))$: these tuples are given value* **0** *also in P'. Then we have that $Sol(P) = Sol(P')$.*

Proof. Take any tuple t with value v in P, which is obtained by combining the values of some subtuples, say two: $v = v_1 \times v_2$. Let us now consider the abstract versions of v_1 and v_2: $\alpha(v_1)$ and $\alpha(v_2)$. Function $\tilde{f}$ changes these values by lowering them, thus we get $\tilde{f}(\alpha(v_1)) = v_1'$ and $\tilde{f}(\alpha(v_2)) = v_2'$.

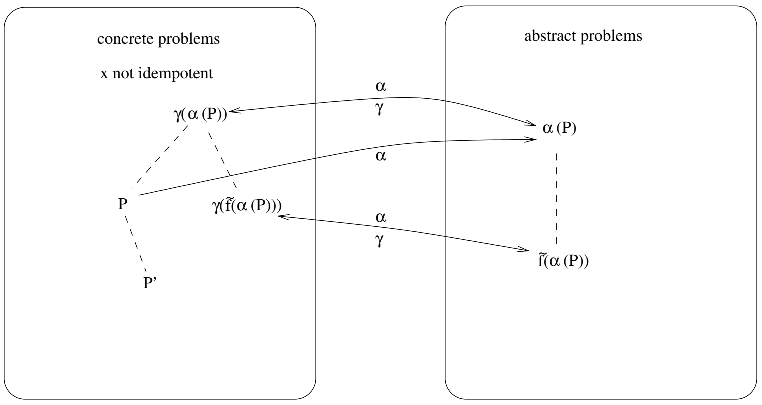


Fig. 4.10. The scheme when $\times$ is not idempotent

Since $\tilde{f}$ is safe, we have that $v_1' \times' v_2' = \alpha(v_1) \times' \alpha(v_2) = v'$. Moreover, $\tilde{f}$ is solution-correct, thus $v \leq_S \gamma(v')$. By monotonicity of γ, we have that $\gamma(v') \leq_S \gamma(v_i')$ for $i = 1, 2$. Thus we have that $v \leq_S \gamma(v_i')$ for $i = 1, 2$.

Now suppose that $\gamma(v_1') = \mathbf{0}$. This implies that also $v = \mathbf{0}$. Therefore, if we set $v_1 = \mathbf{0}$, again the combination of v_1 and v_2 will result in v, which is $\mathbf{0}$.

Example 4.3.8. Consider the abstraction from the semiring $S = \langle Z^- \cup \{-\infty\}$, max, $+$, $-\infty$, $0\rangle$ to the semiring $S' = \langle Z^- \cup \{-\infty\}, max, min, -\infty, 0\rangle$, where α and γ are the identity. This means that we perform the abstraction just to change the multiplicative operation, which is min instead of $+$. Then Figure 4.11 shows a concrete problem over S, and the construction shown in Figure 4.10 over it.

Summarizing, the above theorems can give us several hints on how to use the abstraction scheme to make the solution of P easier: If $\times$ is idempotent, then we can replace P with $P \otimes \gamma(\alpha(\tilde{f}(P)))$, and get the same solutions (by Theorem 4.3.4). If instead $\times$ is not idempotent, we can replace P with P' (by Theorem 4.3.6). In any case, the point in passing from P to $P \otimes \gamma(\alpha(\tilde{f}(P)))$ (or P') is that the new problem should be easier to solve than P, since the semiring values of its tuples are more explicit, that is, closer to the values of these tuples in a completely solved problem.

More precisely, consider a branch-and-bound algorithm to find an optimal solution of P. Then, once a solution is found, its value will be used to cut away some branches, where the semiring value is worse than the value of the solution already found. Now, if the values of the tuples are worse in the new problem than in P, each branch will have a worse value and thus we might cut away more branches. For example, consider the fuzzy semiring (that is, we want to maximize the minimum of the values of the subtuples): if the solution already found has value 0.6, then each partial solution of P with value smaller than or equal to 0.6 can be discarded (together with all its corresponding subtree in

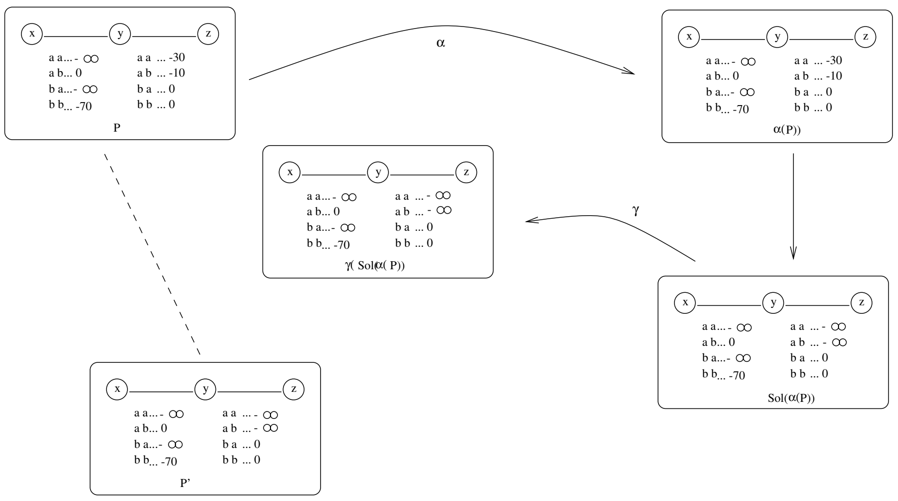


Fig. 4.11. An example with × not idempotent

the search tree), but all partial solutions with value greater than 0.6 must be considered; if instead we work in the new problem, the same partial solution with value greater than 0.6 may now have a smaller value, possibly also smaller than 0.6, and thus can be disregarded. Therefore, the search tree of the new problem is smaller than that of P.

Another point to notice is that, if using a greedy algorithm to find the initial solution (to use later as a lower bound), this initial phase in the new problem will lead to a better estimate, since the values of the tuples are worse in the new problem and thus close to the optimum. In the extreme case in which the change from P to the new problem brings the semiring values of the tuples to coincide with the value of their combination, it is possible to see that the initial solution is already the optimal one.

Notice also that, if $\times$ is not idempotent, a tuple of P' has either the same value as in P, or $\mathbf{0}$. Thus, the initial estimate in P' is the same as that of P (since the initial solution must be a solution), but the search tree of P' is again smaller than that of P, since there may be partial solutions, which in P have value different from $\mathbf{0}$ and in P' have value $\mathbf{0}$, and thus the global inconsistency may be recognized earlier.

The same reasoning used for Theorem 4.3.2 on $\alpha(P)$ can also be applied to $\tilde{f}(\alpha(P))$. In fact, since $\tilde{f}$ is safe, the solutions of $\tilde{f}(\alpha(P))$ have the same values as those of $\alpha(P)$. Thus, also the optimal solution sets coincide. Therefore, we have that $Opt(\tilde{f}(\alpha(P)))$ contains all the optimal solutions of P if the abstraction is order-preserving. This means that, in order to find an optimal solution of P, we can find all optimal solutions of $\tilde{f}(\alpha(P))$ and then use such a set to prune the search for an optimal solution of P.

Theorem 4.3.7. *Given an SCSP P over S, consider a function $\tilde{f}$ on P which is safe, solution-correct and intensive, and let us assume the abstraction is order-preserving. Then we have that $Opt(P) \subseteq Opt(\tilde{f}(\alpha(P)))$.*

Proof. Easy follows from Theorem 4.3.2 and from the safeness of $\tilde{f}$.

Theorem 4.3.3 can be adapted to $\tilde{f}(\alpha(P))$ as well, thus allowing us to use an optimal solution of $\tilde{f}(\alpha(P))$ to find both a lower and an upper bound of an optimal solution of P.

4.4 Some Abstraction Mappings

In this section we will list some semirings and several abstractions between them, in order to provide the reader with a scenario of possible abstractions that he/she can use, starting from one of the semirings considered here. Some of these semirings and/or abstractions have been already described in the previous sections of the chapter; however, here we will re-define them to make this section self-contained. Of course many other semirings could be defined, but here we focus on the ones for which either it has been defined, or it is easy to imagine, a system of constraint solving. The semirings we will consider are the following ones:

- the classical one, which describes classical CSPs via the use of logical *and* and logical *or*:

$$S_{CSP} = \langle \{T, F\}, \vee, \wedge, F, T \rangle$$

- the fuzzy semiring, where the goal is to maximize the minimum of some values over $[0, 1]$:

$$S_{fuzzy} = \langle [0,1],\ \max,\ \min, \mathbf{0}, \mathbf{1} \rangle$$

- the extension of the fuzzy semiring over the naturals, where the goal is to maximize the minimum of some values of the naturals:

$$S_{fuzzyN} = \langle N \cup \{+\infty\},\ \max,\ \min, 0, +\infty \rangle$$

- the extension of the fuzzy semiring over the positive reals:

$$S_{fuzzyR} = \langle R^+ \cup \{+\infty\},\ \max,\ \min, 0, +\infty \rangle$$

- the optimization semiring over the naturals, where we want to maximize the sum of costs (which are negative integers):

$$S_{optN} = \langle Z^- \cup \{-\infty\}, \max, +, -\infty, 0 \rangle$$

- the optimization semiring over the negative reals:

$$S_{optR} = \langle R^- \cup \{-\infty\}, \max, +, -\infty, 0 \rangle$$

– the probabilistic semiring, where we want to maximize a certain probability which is obtained by multiplying several individual probabilities. The idea here is that each tuple in each constraint has associated with it the probability of being allowed in the real problems we are modeling, and different tuples in different constraints have independent probabilities (so that their combined probability is just the multiplication of their individual probabilities) [96]. The semiring is:

$$S_{prob} = \langle [0,1],\ \max, \times, \mathbf{0}, \mathbf{1} \rangle$$

– the subset semiring, where the elements are all the subsets of a certain set, the operations are set intersection and set union, the smallest element is the empty set, and the largest element is the whole given set:

$$S_{sub} = \langle \mathcal{P}(A), \bigcup, \bigcap, \emptyset, A \rangle.$$

We will now define several abstractions between pairs of these semirings. The result is drawn in Figure 4.12, where the dashed lines denote the abstractions that have not been defined but can be obtained by abstraction composition.

In reality, each line in the figure represents a whole family of abstractions, since each $\langle \alpha, \gamma \rangle$ pair makes a specific choice which identifies a member of the family. Moreover, by defining one of this families of abstractions we do not want to say that there do not exist other abstractions between the two semirings.

It is easy to see that some abstractions focus on the domain, by passing to a given domain to a smaller one, others change the semiring operations, and others change both:

1. from fuzzy to classical CSPs: this abstraction changes both the domain and the operations. The abstraction function is defined by choosing a threshold within the interval $[0,1]$, say x, and mapping all elements in $[0,x]$ to F and all elements in $(x,1]$ to T. Consequently, the concretization function maps T to 1 and F to x. See Figure 4.1 as an example of such an abstraction. We recall that all the abstractions in this family are order-preserving, so Theorem 4.3.2 can be used.
2. from fuzzy over the positive reals to fuzzy CSPs: this abstraction changes only the domain, by mapping the whole set of positive reals into the $[0,1]$ interval. This means that the abstraction function has to set a threshold, say x, and map all reals above x into 1, and any other real, say r, into $\frac{r}{x}$. Then, the concretization function will map 1 into $+\infty$, and each element of $[0,1)$, say y, into $y \times x$. It is easy to prove that all the members of this family of abstractions are order-preserving.
3. from probabilistic to fuzzy CSPs: this abstraction changes only the multiplicative operation of the semiring, that is, the way constraints are combined. In fact, instead of multiplying a set of semiring elements, in the abstracted version we choose the minimum value among them. Since the domain remains the same, both the abstraction and the concretization functions are

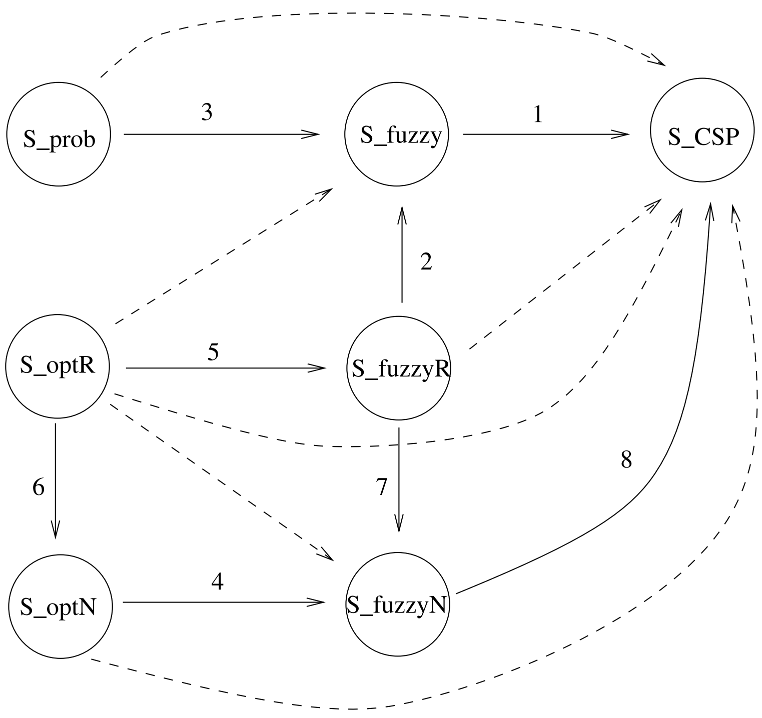


Fig. 4.12. Several semirings and abstractions between them

the identity (otherwise they would not have the properties required by a Galois insertion, like monotonicity). Thus this family of abstractions contains just one member.
It is easy to see that this abstraction is not order-preserving. In fact, consider for example the elements 0.6 and 0.5, obtained in the abstract domain by $0.6 = min(0.7, 0.6)$ and $0.5 = min(0.9, 0.5)$. These same combinations in the concrete domain would be $0.7 \times 0.6 = 0.42$ and $0.9 \times 0.5 = 0.45$, thus resulting in two elements which are in the opposite order with respect to 0.5 and 0.6.

4. from optimization-N to fuzzy-N CSPs: here the domain remains the same (the negative integers) and only the multiplicative operation is modified. Instead of summing the values, we want to take their minimum. As noted in a previous example, these abstractions are not order-preserving.
5. from optimization-R to fuzzy-R CSPs: similar to the previous one but on the negative reals.
6. from optimization-R to optimization-N CSPs: here we have to map the negative reals into the negative integers. The operations remain the same. A possible example of abstraction is the one where $\alpha(x) = \lceil x \rceil$ and $\gamma(x) = x$. It is not order-preserving.

7. from fuzzy-R to fuzzy-N CSPs: again, we have to map the positive reals into the naturals, while maintaining the same operations. The abstraction could be the same as before, but in this case it is order-preserving (because of the use of min instead of sum).
8. from fuzzy-N to classical CSPs: this is similar to the abstraction from fuzzy CSPs to classical ones. The abstraction function has to set a threshold, say x, and map each natural in $[0, x]$ into F, and each natural above x into T. The concretization function maps T into $+\infty$ and F into x. All such abstractions are order-preserving.
9. from subset CSPs to any of the other semirings: if we want to abstract to a semiring with domain A, we start from the semiring with domain $\mathcal{P}(A)$. The abstraction mapping takes a set of elements of A and has to choose one of them by using a given function, for example min or max. The concretization function will then map an element of A into the union of all the corresponding sets in $\mathcal{P}(A)$. For reasons similar to those used in Example 3, some abstractions of this family may be not order-preserving.

4.5 Abstraction vs. Local Consistency

It is now interesting to consider the relationship between our abstraction framework and the concept of local consistency.

In fact, it is possible to show that, given an abstraction $\langle \alpha, \gamma \rangle$ between semirings S and $\bar{S}$ and any propagation rule r in $\bar{S}$, the function $\gamma(r(\alpha(P))) \otimes P$ is a propagation rule for problem P over S. This can be convenient when S does not have any, or any efficient, propagation algorithms. In fact, in such cases, we can resort to the propagation algorithms of $\bar{S}$ to perform propagation also over S.

Notice, however, that, when S has a non-idempotent multiplicative operator, function $\gamma(r(\alpha(P))) \otimes P$ could change the solution of P. To avoid this problem, we just have to follow the same reasoning as in the previous section, that is, to replace such a function with a function that just inserts into P the inconsistencies of $\gamma(r(\alpha(P))$. We will denote such a function by using a different combination operator: $\otimes_0$. Thus, the function to be used in these cases is $\gamma(r(\alpha(P)) \otimes_0 P$. Notice that $\otimes_0$ is a non-commutative operator, since it inserts into the right operand the zeros of the left operand.

This results, however, hold only when the abstraction is order-preserving. We recall that this means that applying the abstraction function and then combining gives elements that are in the same ordering as the elements obtained by combining only. In particular, if two abstract elements $\alpha(x)$ and $\alpha(y)$ are ordered, then also x and y are ordered as well, and in the same direction.

Theorem 4.5.1. *Given an order-preserving abstraction $\langle \alpha, \gamma \rangle$ between semiring S and $\bar{S}$, assume that S has an idempotent multiplicative operation and consider any propagation rule r in $\bar{S}$ and any problem P over S. Then the function $f(P) = \gamma(r(\alpha(P))) \otimes P$ is a propagation rule for P.*

Proof. By definition, a propagation rule is an intensive, monotone, and idempotent function that takes a problem and returns an equivalent problem over the same semiring. Since $\otimes$ is intensive, also f is so. Moreover, by monotonicity of γ, r, α, and $\otimes$, also f is monotone.

For proving idempotency of f, we need the order-preserving property of the abstraction. In fact, consider what happens when applying function f to P: some tuple values in $\alpha(P)$, say $\bar{v} = \alpha(v)$, will not be changed by r, while others will receive a lower value, say $\bar{v}' = r(\bar{v})$. By order-preservation, the new tuples values in the concrete semiring (that is, $\gamma(r(\alpha(v))) \times v$), are equal or lower than the original values. Let us now apply function f again. Function α will bring these new concrete values to either $\bar{v}$ (if we start from v) or $\bar{v}'$ (if we start from $\gamma(r(\alpha(v)))$). In any case, r will bring such values to $\bar{v}'$, for Lemma 4.5.1 (see below). Thus f is idempotent. Finally, f returns an equivalent problem by the theorem depicted in Figure 4.6.

Lemma 4.5.1. *Consider any SCSP P over S and any propagation rule r for P, with $r(P) = P'$. Then, taken any SCSP P'' such that $P' \leq_S P'' \leq_S P$, we have $r(P'') = P'$.*

Proof. Any rule r solves a subproblem $\langle C, con\rangle$ and changes the values of the tuples connecting the variables in *con*. Thus, the result of applying r is a new constraint over *con*: $(C \otimes C_{con}) \Downarrow_{con}$, where C_{con} is the original constraint connecting the variables in *con*. This can also be written as $C_{con} \otimes C \Downarrow_{con}$. Let us now take any C''_{con} such that $(C_{con} \otimes C \Downarrow_{con}) \leq_S C''_{con} \leq_S C_{con}$. We can now multiply all these three constraints by $C \Downarrow_{con}$, obtaining: $(C_{con} \otimes C \Downarrow_{con} \otimes C \Downarrow_{con}) \leq_S (C''_{con} \otimes C \Downarrow_{con}) \leq_S (C_{con} \otimes C \Downarrow_{con})$. By idempotency of $\otimes$, we get: $(C_{con} \otimes C \Downarrow_{con}) \leq_S (C''_{con} \otimes C \Downarrow_{con}) \leq_S (C_{con} \otimes C \Downarrow_{con})$. Thus we have that $(C_{con} \otimes C \Downarrow_{con}) = (C''_{con} \otimes C \Downarrow_{con})$.

We can now prove a similar result for the case of a non-idempotent multiplicative operation in the concrete semiring. As noted above, however, we cannot combine the new problem with the old one, but we can just insert the zeroes of the new problem into the old one.

Theorem 4.5.2. *Given an order-preserving abstraction $\langle \alpha, \gamma\rangle$ between semiring S and $\bar{S}$, assume that S has a non-idempotent multiplicative operation and consider any propagation rule r in $\bar{S}$ and any problem P over S. Then the function $f(P) = \gamma(r(\alpha(P))) \otimes_0 P$ is a propagation rule for P, where $\otimes_0$ inserts the zeroes of its left operand into the right one.*

Proof. The proof of this theorem is similar to the previous one, and for the equivalence it refers also to the theorem depicted in Figure 4.10.

By taking several propagation rules in the abstract semiring, we can thus obtain an equal number of propagation rules over the concrete semiring. This set of rules can then be used to perform constraint propagation over a concrete problem. Notice, however, that, while an idempotent multiplicative operation in the concrete semiring allows us to use such rules until stability, with all the desired properties (equivalence, uniqueness, and termination), in the case of a non-idempotent multiplicative operation we can just apply the various propagation rules once each to insert several zeroes into the original problem (with the same properties as above).

4.6 Related Work

We will compare here our work to other abstraction proposals, more or less related to the concepts of constraints.

Abstracting valued CSPs.. The only other abstraction scheme for soft constraint problems we are aware of is the one in [169], where *valued CSPs* (see Section 2.3.7) are abstracted in order to produce good lower bounds for the optimal solutions. The concept of valued CSPs is similar to our notion of SCSPs. In fact, in valued CSPs, the goal is to minimize the value associated to a complete assignment. In valued CSPs, each constraint has one associated element, not one for each tuple of domain values of its variables; however, our notion of soft CSPs and that in valued CSPs are just different formalizations of the same idea, since one can pass from one formalization to the other one without changing the solutions, provided that the partial order is total (see Section 2.3.7). However, our abstraction scheme is different from the one in [169]. In fact, we are not only interested in finding good lower bounds for the optimum, but also in finding the exact optimal solutions in a shorter time. Moreover, we do not define *ad hoc* abstraction functions but we follow the classical abstraction scheme devised in [77], with Galois insertions to relate the concrete and the abstract domain, and locally correct functions on the abstract side. We think that this is important in that it allows to inherit many properties that have already been proven for the classical case. It is also worth noticing that our notion of an order-preserving abstraction is related to their concept of aggregation compatibility, although generalized to deal with partial orders.

Abstracting classical CSPs.. Other work related to abstracting constraint problems proposed the abstraction of the domains [106, 107, 175], or of the graph topology (for example to model a subgraph as a single variable or constraint) [94]. We did not focus on these kinds of abstractions for SCSPs in this chapter, but we believe that they could be embedded into our abstraction framework: we just need to define the abstraction function in such a way that not only we can change the semiring but also any other feature of the concrete problem. The only difference will be that we cannot define the concrete and abstract lattices of problems by simply extending the lattices of the two semirings.

A general theory of abstraction.. A general theory of abstraction has been proposed in [192]. The purpose of this work is to define a notion of abstraction that can be applied to many domains: from planning to problem solving, from theorem proving to decision procedures. Then, several properties of this notion are considered and studied. The abstraction notion proposed consists of just two formal systems Σ_1 and Σ_2 with languages L_1 and L_2 and an effective total function $f : L_1 \rightarrow L_2$, and it is written as $f : \Sigma_1 \Rightarrow \Sigma_2$. Much emphasis is placed in [192] onto the study of the properties that are preserved by passing from the concrete to the abstract system. In particular, one property that appears to be very desirable, and present in most abstraction frameworks, is that *what is a theorem in the concrete domain, remains a theorem in the abstract domain* (called the **TI** property, for *Theorem Increasing*).

It is easy to see that our definition of abstraction is an instance of this general notion. Then, to see whether our concept of abstraction has this property, we first must say what is a theorem in our context. A natural and simple notion of a theorem could be an SCSP which has at least one solution with a semiring value different from the 0 of the semiring. We can be more general than this, however, and say that a theorem for us is *an SCSP which has a solution with value greater than or equal to k, where* $k \geq 0$. Then we can prove our version of the TI property:

Theorem 4.6.1 (our TI property). *Given an SCSP P which has a solution with value $v \geq k$, then the SCSP $\alpha(P)$ has a solution with value $v' \geq \alpha(k)$.*

Proof. Take any tuple t in P with value $v > k$. Assume that $v = v_1 \times v_2$. By abstracting, we have $v' = \alpha(v_1) \times' \alpha(v_2)$. By solution correctness of $\times'$, we have that $v \leq_S \gamma(v')$. By monotonicity of α, we have that $\alpha(v) \leq_{S'} \alpha(\gamma(v')) = v'$. again by monotonicity of α, we have $\alpha(k) \leq_{S'} \alpha(v)$, thus by transitivity $\alpha(k) \leq_S v'$.

Notice that, if we consider the boolean semiring (where a solution has either value true or false), this statement reduces to saying that if we have a solution in the concrete problem, then we also have a solution in the abstract problem, which is exactly what the TI property says in [192]. Thus, our notion of abstraction, as defined in the previous sections, on the one hand can be cast within the general theory proposed in [192], while on the other hand it generalizes it to concrete and abstract domains, which are more complex than just the boolean semiring. This is predictable, because, while in [192] formulas can be either true (thus theorems) or false, here they may have any level of satisfaction, which can be described by the given semiring.

Notice also that, in our definition of abstraction of an SCSP, we have chosen to have a Galois insertion between the two lattices $\langle A, \leq \rangle$ (which corresponds to the concrete semiring S) and $\langle \tilde{A}, \tilde{\leq} \rangle$ (which corresponds to the abstract semiring $\tilde{S}$). This means that the ordering in the two lattices coincide with those of the semirings. We could have chosen differently: for example, that the ordering of the lattices in the abstraction be the opposite of those of those in the semirings. In that case, we would not have had property **TI**. We would, however, have the dual property (called **TD** in [192]), which states that abstract theorems remain theorems in the concrete domain. It has been shown that such a property can be useful in some application domains, such as databases.

4.7 Conclusions

In this chapter we have described how to apply *abstraction* to SCSPs. The main motivation to apply such a transformation is to obtain an easier representation of the problem, or (and many times) to obtain an SCSP whose semiring satisfied the needed properties for preprocessing algorithms applications.

5. Higher Order Semiring-Based Constraints

Overview

Semiring-based constraint problems (SCSPs), as described in Chapter 2, extend classical constraint problems (CSPs) by allowing preferences, costs, priorities, probabilities, and other soft features. They are based over the notion of a semiring, that is, a set plus two operations. In this chapter we introduce a uniform, abstract presentation of (soft) constraint satisfaction concepts and constructions. Moreover, the soft constraint environment is enriched with the operation of function abstraction and application that can suitably be used to give modularity and compositionality to the framework. Finally, a small language is defined and used to represent and give semantics to general local propagation and dynamic programming algorithms.

In this chapter the basic structures of the framework are represented as functions, and this gives modularity and compositionality. Many of the algorithms defined over a constraint system, like constraint propagation and dynamic programming techniques, can easily be expressed using expressions and commands of a suitable language.

The basic elements that we need to consider are the *domains* and the *semirings*; the whole structure of an SCSP will be represented as a function between these elements. As soon as we define constraints and SCSPs as specific functions, we can use abstraction, application and composition to model SCSP solution, dynamic programming, and local consistency preprocessing algorithms.

The representation of constraints as functions gives the possibility to represent a lot of useful features present in real problems. By performing a step-by-step application of the definitions of the constraints to the resolution function, we can model environments where the constraints are specified at run time by the user, or we can compute (partial) solutions for partially instantiated problems.

The chapter is organized as follows. First, in Section 5.1 we define the domain and semiring structure and we illustrate how constraints and SCSPs are represented as specific functions; in Section 5.3 we define a small language useful to represent algorithms over SCSPs; and in Section 5.4 we define the solution and the preprocessing algorithms as specific expressions and commands of the language. Finally in Section 5.5 we give some possible extension of the framework.

This chapter is based on some preliminary work presented in [49].

5.1 Domains and Semirings

All the SCSP's elements (constraints and problems) are defined here as functions over the set of domains D and semirings S. A domain can be a set D_0 (representing all the values that can be assigned to variables) or can be constructed using the cartesian product. Note that the domain of the variables may be different, so we have a collection $D_1, \dots, D_n$ of variable domains.

Definition 5.1.1 (domains). *The class of domains is denoted D and it is defined as*

$$D ::= D_1 \mid \cdots \mid D_n \mid D \times D$$

Intuitively, a value $d \in D$ represents a domain value for a variable in a given SCSP, while a value $\langle d_1, d_2 \rangle \in D_1 \times D_2$ and a value $\langle d_1, \dots, d_k \rangle \in D_1 \times \cdots \times D_k$ respectively represents a pair of domain values for a binary constraint and a k-tuple of domain values for a k-ary constraint.

The semiring structure will be represented as S; note that S represents all the primitive instances (for example the structures used for *fuzzy-CSPs*, *probabilistic-CSPs*, *weighted-CSPs*, etc.) and also the semiring obtained by using some special operators like the cartesian product (described in Section 2.3.8). In this case there is a collection of basic semiring $S_1, \dots, S_n$ (see Chapter 2) and their cartesian product.

Definition 5.1.2 (semiring). *The class of semirings is denoted by S and is defined as*

$$S ::= S_1 \mid \cdots \mid S_n \mid S \times S$$

By using the properties of $\times$, $+$ and of the cartesian product, we can write down some basic relationships between semiring elements; In Table 5.1 symbol $\star$ denotes any of the semiring operators $\times$ and $+$. The table shows how the semiring operations of the cartesian product is defined using the corresponding operations in the primitive semiring.

By using functions from D to S we can now represent (soft) constraints.

Definition 5.1.3 (constraints). *The set of constraints (over a semiring S) is denoted by C and it is defined as*

$$C ::= D \rightarrow S$$

$$\frac{a_1 \in S_1, a_2 \in S_2}{\langle a_1, a_2 \rangle \in S_1 \times S_2}$$

$$\frac{\langle a_1, a_2 \rangle, \langle b_1, b_2 \rangle \in S_1 \times S_2}{\langle a_1, a_2 \rangle \star_{S_1 \times S_2} \langle b_1, b_2 \rangle = \langle a_1 \star_{S_1} b_1, a_2 \star_{S_2} b_2 \rangle \in S_1 \times S_2}$$

Table 5.1. Relationships between semiring elements.

Note that the syntactic category D represents the set of domains of all the constraints of the problem. For unary constraints (that is, variable domain), there will be functions $D_i \rightarrow S$ for some i, and for the non unary constraint over $k \geq 1$ variables there will be a function $D_1 \times \cdots \times D_k \rightarrow S$.

We can now describe entire constraint problems:

Definition 5.1.4 (constraint problem). *Fixed a semiring S and the number n of constraints, a constraint problem P is defined as*

$$P ::= C^n \rightarrow C.$$

where C^n is obtained as the cartesian product of n constraints, that is $C^n = \underbrace{C \times \cdots \times C}_{n\ times}$

Example 5.1.1. Consider the fuzzy CSP represented in Figure 5.1. In this case the semiring for the problem is

$$S_{FCSP} = \langle [0,1], max, min, 0, 1 \rangle.$$

The domain is the same for all the variables that is $D = \{a, b\}$. The constraints c_x, c_y, c_z are functions $\{a, b\} \rightarrow S_{FSCP}$,and constraints c_1 and c_2 are functions $\{a, b\}^2 \rightarrow S_{FSCP}$.

The problem represented in Figure 5.1 is instead a function $c_x \times c_y \times c_z \times c_1 \times c_2 \rightarrow (D^2 \rightarrow S_{FSCSP})$.

5.2 Constraint Problems and Solutions

Since both constraints and constraint problems are defined as functions, we are now able to use function composition, abstraction and application to define in a general and easy way many solution or preprocessing algorithms. The definition of a constraint problem P can be split in two parts: the first one is called *constraint graph* and describes its *topology*, and the second one is called *solution* and represents its *value*. Note that problems with different topology can have

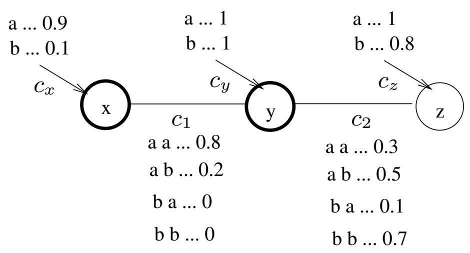


Fig. 5.1. A 3-variable fuzzy CSP

the same value, and that many times the same topology corresponds to different values.

To define the constraint graph in a concise way we need to introduce some special operators and properties:

Definition 5.2.1 (needed operators and properties).

- *The* restriction *operator ν will be useful to capture the variables of interest of a problem:*

$$\frac{f \in D \rightarrow S}{\nu f = \sum_{x \in D} f(x)}$$

We will write $\nu x.M$ instead of $\nu(\lambda x.M)$ and $\nu\langle x, y\rangle$ instead of $\nu x.\nu y$, with the meaning of restricting over pairs (or tuples) of domains instead of single domains.

- *some of its important properties*

$$\nu x.(t_1 \star t_2) = (\nu x.t_1) \star t_2 \textit{ if } x \notin FV(t_2) \qquad (\nu\text{-distributivity})$$
$$\nu x.(t_1 +_S t_2) = (\nu x.t_1) +_S (\nu x.t_2) \qquad (\nu\text{-associativity})$$

Let us define now the *constraint graph* and the *problem solution.*

We also remind (see Section 2.2) that a a constraint system is defined as a tuple $CS = \langle S, D, V\rangle$, where S is a c-semiring, D is a finite set, and V is an ordered set of variables. A constraint over CS is a pair $\langle def, con\rangle$, where

- $con \subseteq V$, it is called the *type* of the constraint;
- $def : D^k \rightarrow A$ (where k is the cardinality of con) is called the *value* of the constraint.

Definition 5.2.2 (Constraint graph). *Given an SCSP $P = \langle C, con\rangle$ over a constraint system $CS = \langle S, D, V\rangle$ with $C = \{c_1, \ldots, c_n\}$, $c_i = \langle def_i, con_i\rangle$, $def_i : D^{|con_i|} \rightarrow S$ and $V = \{x_1, \ldots, x_k\}$, a* constraint graph *is a function defined as follows:*

$$P = \lambda\langle c_1, \ldots, c_n\rangle.\lambda\langle con\rangle.\nu\langle V - con\rangle. \prod_{i=1\ldots n} c_i(con_i)$$

The meaning of this function is to take the constraints $\{c_1, \ldots, c_n\}$ over the variables in V and to perform their combination $\prod_{S i=1\ldots n} c_i(con_i)$. Since we are interested only to a subset of the variable $con \in V$, we need to use the ν operator to sum up over the variable in $V - con$. This is not completely correct because con and $V - con$ are sets of variables and not tuples. Anyway, since we fix a lexicographic order over the variables in V it is easy to pass from sets to tuples.

The *value* of an SCSP is obtained applying the constraint graph to the definition of its constraints. The result of this application represents the *problem solution.*

Definition 5.2.3 (Problem solution). *Given an SCSP $P = \langle C, con \rangle$ over a constraint system $CS = \langle S, D, V \rangle$ with $C = \{c_1, \ldots, c_n\}$, $c_i = \langle def_i, con_i \rangle$, $def_i : D^{|con_i|} \rightarrow S$, $V = \{x_1, \ldots, x_k\}$ and the corresponding constraint graph*

$$P = \lambda\langle c_1, \ldots, c_n\rangle.\lambda\langle con\rangle.\nu\langle V - con\rangle. \prod_{i=1\ldots n} c_i(con_i),$$

the function solution is defined as

$$Sol(P) = P(DEF_P) = \lambda\langle con\rangle.\nu\langle V - con\rangle. \prod_{i=1\ldots n} def_i(con_i)$$

where DEF_P represents the ordered list of all the definitions of the constraints in P.

Example 5.2.1. Consider again the fuzzy CSP represented in Figure 5.1. The constraint graph is represented by the function

$$\lambda\langle c_1, c_2, c_x, c_y, c_z\rangle.\lambda\langle x, y\rangle.\nu z.c_1(x, y) \times_S c_2(y, z) \times_S c_x(x) \times_S c_y(y) \times_S c_z(z).$$

The structure of this term gives an explicit way to represent an SCSP with 5 constraints and 3 variables, and shows how constraints c_1 and c_2 share one of the variables. Note that in this term, the variables x, y and z, and the constraints c_1,c_2,c_x,c_y,c_z are bound by λ-abstraction, so any term obtained from this one via α-renaming represents the same SCSP (i.e., α-renamed terms represent all the SCSPs with this shape). Note also that the $\times_S$ operator in the body of the function is not instantiated to the corresponding operator of the (fuzzy, in this case) semiring. This means that only the shape of the problem is represented and not its value. In other words, the semantical meaning of the $\times$ operator (that is the min) is not used.

To compute the solution of the problem, we first need to apply the definition of the constraints to the constraint graph, that is, to compute $P(DEF_P)$. In this way we obtain

$$\lambda\langle x, y\rangle.\nu z.def_1(x, y) \times_S def_2(y, z) \times_S def_x(x) \times_S def_y(y) \times_S def_z(z).$$

Now, if we want to compute the value of this SCSP we have to instantiate the variable and to compute the value of the solution function. As an example of solution, consider the pair $x = a$ and $y = b$. We obtain the function

$$\nu z.0.2 \times_S def_2(b, z) \times_S 0.9 \times_S 1 \times_S def_z(z) =$$

by instantiating the $\times_S$ operator to min

$$\nu z.min(0.2, def_2(b, z), 0.9, 1, def_z(z)).$$

By definition of ν we obtain

$$max_{z\in\{a,b\}}min(0.2, def_2(b, z), 0.9, 1, def_z(z)) =$$
$$max(min(0.2, def_2(b, a), 0.9, 1, def_z(a)), min(0.2, def_2(b, b), 0.9, 1, def_z(b))) =$$
$$max(min(0.2, 0.1, 0.9, 1, 1), min(0.2, 0.7, 0.9, 1, 0.8)) =$$
$$max(0.1, 0.2) = 0.2$$

The possibility to write the problem solution as a function, give as the possibility to compute partial solution for partially instantiated problem. This situation often arise in reality specially when the problem involve several data and the user knows only some of them. Another situation that can be represented by using partial instantiated SCSP is when the data of the problem are sent sequentially in several steps. In this case we can try to approximate the solution of the problem also before having all the problem data.

As an example let us consider same problem of Example 5.2.1 but by considering the case we do not know any information for constraint c_y.

Example 5.2.2. Consider again the fuzzy CSP represented in Figure 5.1, but suppose we do not know the detail of constraint c_y.

The solution of the problem is represented by the function

$$\lambda\langle x, y\rangle.\nu z.def_1(x,y) \times_S def_2(y,z) \times_S def_x(x) \times_S def_y(y) \times_S def_z(z).$$

Let suppose we want as before compute the value of the solution for the pair $x = a$ and $y = b$. We obtain the function

$$\nu z.0.2 \times_S def_2(b,z) \times_S 0.9 \times_S def_y(b) \times_S def_z(z) =$$

by instantiating the $\times_S$ operator to *min*

$$\nu z.min(0.2, def_2(b,z), 0.9, def_y(b), def_z(z)).$$

By definition of ν we obtain

$$\begin{aligned}&max_{z\in\{a,b\}} min(0.2, def_2(b,z), 0.9, def_y(b), def_z(z)) =\\ &max(min(0.2, def_2(b,a), 0.9, def_y(b), def_z(a)),\\ &min(0.2, def_2(b,b), 0.9, def_y(b), def_z(b))) =\\ &max(min(0.2, 0.1, 0.9, def_y(b), 1), min(0.2, 0.7, 0.9, def_y(b), 0.8))\end{aligned}$$

At this point we can try to make some consideration:

- $Sol(P)(a,b) = def_y(b)$ if $def_y(b) \leq_S 0.2$
- $Sol(P)(a,b) = 0.2$ else

So, if we statistically know that the value of $def_y(b)$ is always lower than 0.2 we can give a good approximation of the solution by declaring $sol(P)(a,b) = 0.2$.

5.3 A Small Language to Program with Soft Constraints

We will now define a small language that will be useful in representing and solving SCSPs, using local consistency and dynamic programming algorithms. To define such algorithms, we need to explicitly define the *typed locations*, which we remind are just a set of variables, that will be used to identify a subproblem of a given SCSP.

Definition 5.3.1 (Typed location). *A* typed location *l is a set of variables. Given a constraint system $\langle S, D, V \rangle$, the set of locations $\mathcal{L}$ is defined as $\mathcal{L} = \wp(V)$.*

Locations represent variables of type *constraint*; that is, each location can contain a constraint (i.e. a function $D \rightarrow S$). Let us now define the *(Constraint) Store.*

Definition 5.3.2 (Constraint Store). *A constraint store is a function σ such that $\sigma l : D^{|l|} \rightarrow S$, where we have $\sigma l \langle x_1, \ldots, x_{|l|} \rangle = \mathbf{1}$ for all except a finite number of locations.*

Given a constraint Problem $P = \langle C, con \rangle$, we can build the corresponding constraint store σ_C defined as follows:

$$\sigma_C l = \begin{cases} def_i \text{ if } c_i = \langle def_i, l \rangle \in C \\ \lambda \langle v_1, \ldots, v_{|l|} \rangle . \mathbf{1} \text{ otherwise.} \end{cases}$$

In the following we will write σ instead of σ_C if the corresponding problem $P = \langle C, con \rangle$ is easy to be recognized.

Let us now define the syntactic structure of our language: the *expressions Expr* and the *commands Com.*

Definition 5.3.3 (Expressions). *The expressions we can build with the language are:*

$$Expr ::= c \mid l \mid Sol(l) \mid Sol(l, L) \mid \phi(l, L)$$

The element c represents a constraint (that is, a basic piece of information); $Sol(l)$ represents the solution of a constraint problem whose variables of interest are the variables in l. $Sol(l, L)$ represents instead the solution of a subproblem (whose constraints are those connecting variables in $L \cup \{l\}$) and will be useful to define local consistency and dynamic programming algorithms. Finally, $\phi(l, L)$ is a generic monotone and extensive function that will be useful to represent generic soft local consistency algorithms.

Before giving a semantics to the expressions, let us introduce the syntactic category of commands.

Definition 5.3.4 (Commands). *The commands we can build with the language are defined by the following grammar:*

$$Com ::= l := Expr \mid Com; Com$$

Note that the assignment command "$l := Expr$" can be performed only if the type of the expression *Expr* is compatible with the location l.

The assignment command is useful to change the value of a location present in the constraint store, and the ";" operator is useful to perform several assignments sequentially.

To formally compute the *semantics* of the expressions and of the commands, we need to define two semantic functions: $\mathcal{E}$ for the expressions and $\mathcal{C}$ for the commands.

Function $\mathcal{E}$ evaluates an expression on a constraint store and gives as a result a constraint (that is, an element $c \in C$ that assigns to each tuple of elements in D a value in S).

Definition 5.3.5 (Function $\mathcal{E}$). *The semantic function*

$$\mathcal{E} : Expr \to \sigma \to C$$

is defined by structural recursion as follows:

$$\begin{aligned}
\mathcal{E}[\![\, Sol(l) \,]\!]\sigma &= \lambda\langle l\rangle.\nu\langle V - l\rangle. \prod\nolimits_{S}{}_{l' \in \mathcal{L}} \sigma l'\langle l'\rangle \\
\mathcal{E}[\![\, Sol(l, L) \,]\!]\sigma &= \lambda\langle l\rangle.\nu\langle L\rangle. \prod\nolimits_{S}{}_{l' \in (L \cup \{l\})} \sigma l'\langle l'\rangle \\
\mathcal{E}[\![\, c \,]\!]\sigma &= \hat{c} \\
\mathcal{E}[\![\, l \,]\!]\sigma &= \sigma l \\
\mathcal{E}[\![\, \phi(l, L) \,]\!]\sigma &= \hat{\phi}(l, L)
\end{aligned}$$

The element $\hat{c}$ and $\hat{\phi}(l, L)$ are basic elements representing respectively a constraint definition (that is, its def function) and a generic monotone and extensive function $\hat{\phi}(l, L) : (\prod_{l' \in (L \cup \{l\})} D^{|l'|} \to S) \to (D^{|l|} \to S)$. The function $Sol(l, L)$ is a specific $\hat{\phi}(l, L)$, and the function $sol(l)$ is obtained by performing $Sol(l, \mathcal{L})$ where $\mathcal{L}$ is the set of all the locations of the problem.

Since the language we are going to define uses assignments, we need to define the meaning of store modification:

Definition 5.3.6 (Store modifications). *Given a constraint store σ, and a constraint $c : D^{|l|} \to S$, the store modification operator is defined as follows:*

$$\sigma[c/l]l' = \begin{cases} c \text{ if } l' = l \\ \sigma l' \text{ otherwise.} \end{cases}$$

Let us now evaluate the commands using the function $\mathcal{C}$.

Definition 5.3.7 (The Function $\mathcal{C}$). *The semantic function*

$$\mathcal{C} : Com \to \sigma \to \sigma$$

is defined by structural recursion as follows:

$$\begin{aligned}
\mathcal{C}[\![\, l := e \,]\!]\sigma &= \sigma[\mathcal{E}[\![\, e \,]\!]\sigma/l] \\
\mathcal{C}[\![\, com_1; com_2 \,]\!]\sigma &= \mathcal{C}[\![\, com_2 \,]\!](\mathcal{C}[\![\, com_1 \,]\!]\sigma)
\end{aligned}$$

The assignment command $l := e$ change the constraint defined over the variable in l by using the constraint e. This type of modification of the store usually happens when a preprocessing algorithm is applied to the problem. In fact, in this case, the value of the constraints is changed (usually is lowered) by also maintaining the equivalence with the original problem.
The sequentialisation operator is instead needed since we want to put together several assignment operation and build several preprocessing (or solving) techniques.

5.4 Solving SCSPs

In this section we will show how the dynamic programming and local consistency algorithms can be represented as specific expressions of the language defined in Section 5.3.

5.4.1 Dynamic Programming Techniques

Using dynamic programming techniques, the problem is not solved in one step, but it is partitioned in subproblems which form a *parsing tree* S, and then it is solved by solving such subproblems in a bottom-up visit of S.

A subproblem in a parsing tree S is represented by a rule $r_i = (l_i \leftarrow L_i)$ where l_i represents the location of interest of the problem, and L_i represents all the variables and constraints involved in the subproblem (see Section 3.1). The parsing tree $S = r_1; \ldots; r_n$, as defined in [47], represents a correct way to decompose the global problem, and also gives a computation order of the solutions of the subproblems. In our syntax the solution of a subproblem defined by a rule $l \leftarrow L$ is defined as the term $Sol(l, L)$.

Theorem 5.4.1 (Dynamic programming solving). *Consider a problem $P = \langle C, con\rangle$ over the constraint system $CS = \langle S, D, V\rangle$, the corresponding constraint store σ and one of its parsing trees $S = r_1; \ldots; r_n$, where $r_i = (l_i \leftarrow L_i)$, we have:*

$$\mathcal{E}[\![\, Sol(con) \,]\!]\sigma = \mathcal{E}[\![\, l_n \,]\!](\mathcal{C}[\![\, l_1 := Sol(l_1, L_1); \ldots; l_n := Sol(l_n, L_n) \,]\!]\sigma)$$

Proof. The proof follows the lines of the Theorem 3.6.1, where the store σ takes the place of P and the semantics of subproblem solution $\mathcal{C}[\![\, l := sol(l, L) \,]\!]\sigma$ takes the place of the rule application $[l \leftarrow L]P$.

Example 5.4.1. Let us review some steps of the dynamic programming technique applied to the running example of the chapter represented in Figure 5.1.

Suppose to have the strategy $S = r_1, r_2$ with $r_1 = \{y\} \leftarrow \{\langle y, z\rangle, z\}$ and $r_2 = \{\langle x, y\rangle\} \leftarrow \{x, y\}$. So we need to show that

$$\mathcal{E}[\![\, Sol(\langle x, y\rangle) \,]\!]\sigma = \mathcal{E}[\![\, Sol(\langle x, y\rangle, \{x, y\}) \,]\!](\mathcal{C}[\![\, \{y\} := Sol(\{y\}, \{\langle y, z\rangle, z\}) \,]\!]\sigma).$$

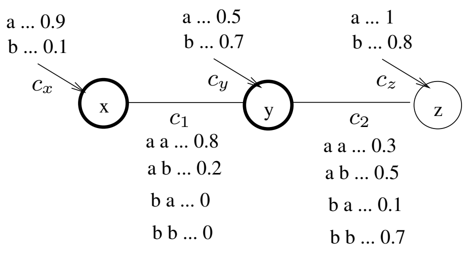


Fig. 5.2. The fuzzy CSP after the application of rule r_1

By solving the subproblem represented by the first rule, we obtain the constraint problem represented in Figure 5.2. Let us call σ'_C the constraint store represented in the figure.

By definition of the ";" operator, we now need to solve the problem

$$\mathcal{E}[\![\, Sol(\langle x, y\rangle, \{x, y\}) \,]\!]\sigma'_C.$$

It is now easy to check that the solution computed in this way is equivalent to the solution computed by giving semantics to $\mathcal{E}[\![\, Sol(\langle x, y\rangle) \,]\!]\sigma$.

5.4.2 Local Consistency Techniques

In this section we show how several soft local consistency algorithms can be captured and described in a uniform way in the framework described in this chapter. In section 3.5 we already described in a general way constraint propagation algorithms using monotone functions (extending in this way the techniques described in Section 3.1 with the results of Apt in [10,11,12]). Here we first represent classical soft local consistency rules as commands in our language, and then we show how the language is also able to capture more extended versions of the local propagation algorithms as depicted in Section 3.4.

For each local consistency rule that we have in our strategy, we define a suitable assignment that modifies a piece of the constraint store related to the consistency rule itself. We can claim that the semantics of $Sol(con)$ is the same if we compute it in the constraint store σ or in a constraint store $\mathcal{C}[\![\, Sol(l, L) \,]\!]\sigma$ for any l, L, and σ.

Theorem 5.4.2 (Equivalence for $Sol(l, L)$**).** *Consider a problem* $P = \langle C, con\rangle$ *and a local consistency rule* $[l \leftarrow L]$. *Then we have that*

$$\mathcal{E}[\![\, Sol(con) \,]\!]\sigma = \mathcal{E}[\![\, Sol(con) \,]\!](\mathcal{C}[\![\, l := Sol(l, L) \,]\!]\sigma).$$

Proof. Easily follows from Theorem 3.1.3.

5.4.3 Extending Local Propagation Rules

In the classical CSP environment, a local propagation function is defined (following the approach of [10, 11, 12]) as a monotone and extensive function that transforms the original problem into a new (possibly equivalent) one. This means that not only a solution over a set of locations $L \cup \{l\}$ and a projection operation over the variables in l are performed but *any* monotone and extensive transformation can be applied to the location l using the locations $L \cup \{l\}$.

For this purpose we have introduced in the language the function $\phi(l, L)$ with the meaning to compute any modification over l by using the constraints over $l \cup L$.

More generally, a local propagation function can be seen as a function that transforms a problem into a new one (possibly by modifying only a subset of it).

Definition 5.4.1 (Local propagation). *Given a problem $P = \langle C, con\rangle$ and the corresponding store σ, a local propagation function lc is a monotone and extensive (w.r.t. the semiring order $\leq_S$) function that transforms (some of) its constraints obtaining a (possibly equivalent) problem:*

$$lc : \sigma \to \sigma$$

The definition of lc is so general that it could match with any of the previously defined local consistency algorithms. We want to characterize it in a more precise way, by using a set of *general local propagation rules* by extending the rules of Section 3.1.

Definition 5.4.2 (general local propagation rule). *A general local propagation rule is defined by a subproblem and a monotone and extensive (w.r.t. the semiring order $\leq_S$) function ϕ. To indicate the application of a function ϕ to a subproblem defined by a rule $r = l \leftarrow L$, we will write $\phi(l, L)$.*

In general since the function $\phi(l, L)$ is monotone and extensive we have:

Theorem 5.4.3. *Consider a problem $P = \langle C, con\rangle$ and a general local consistency rule $\phi(l, L)$. Then, for any domain tuple l' we have that*

$$\mathcal{C}[\![\, l := \phi(l, L) \,]\!]\sigma l'\langle l'\rangle \leq_S \sigma l'\langle l'\rangle.$$

Proof. Easily follows by the hypothesis of monotonicity of ϕ.

5.5 Constraint Problem as Semiring

In this section we illustrate a possible approach to extend the notion of semiring to constraint problems. This means that we want the semiring to capture non only the levels of preference of each instantiated constraint but also the constraint itself.

To do this we define a possible semiring representing Constraint Problems. We need first to define the operator that represents the combination of two constraints. Note that since the combination of constraints has an operational behaviour that depends on the variables that are shared between the two constraints, we do not have a unique combination operator, but a collection of them.

Moreover we need also a notion of *type* for each constraint. In the following we will write $c : (D_0 \times \ldots \times D_n) \to S$ if the constraint c involve variable $x_0, \ldots, x_n$.

Definition 5.5.1 (Combination: $\otimes_D$).

$$\frac{c_1 : (D_0 \times D_1) \to S, c_2 : (D_0 \times D_2) \to S}{c_1 \otimes_{D_0} c_2 : (D_0 \times D_1 \times D_2) \to S = \lambda\langle x_0, x_1, x_2\rangle . c_1(x_0, x_1) \times_S c_2(x_0, x_2)}$$

The $\oplus$ operator can be instead defined as follows:

Definition 5.5.2 (Disjunction: $\oplus_D$).

$$\frac{c_1 : (D_0 \times D_1) \to S, c_2 : (D_0 \times D_2) \to S}{c_1 \oplus_{D_0} c_2 : D_0 \to S = \lambda x_0 . \nu\langle x_1, x_2\rangle . c_1(x_0, x_1) +_S c_2(x_0, x_2)}$$

We can now claim that the set of constraints C with the combination and disjunction operations is an *enhanced*[1] semiring.

To define the unit element of the two operation we need the definition of constant function over a given set of variables:

Definition 5.5.3 (constant function: I_a). *Consider a set of variables I. We define the function $I_a : D_I \to A$ as the function that assigns to each I-tuple t the semiring value a.*

Theorem 5.5.1 ($\langle C, \oplus, \otimes, V_\mathbf{0}, \emptyset_\mathbf{1}\rangle$ is an enhanced semiring). *$\langle C, \oplus, \otimes, V_\mathbf{0}, \emptyset_\mathbf{1}\rangle$ is an enhanced semiring.*

Proof. It is enough to check all the properties by using the fact that the same properties holds for the original semiring S.

5.6 Conclusions

In this chapter we have described how the *semiring* framework can be used to both embed the constraint structure and topology in a suitable semiring of functions. We have also described a small language to "program" with soft constraints. Starting from basic objects (domains and semirings), we have built functions able to represent single steps of the local consistency and dynamic programming algorithms. By using this language we could give a procedural view of the solving techniques already described in the previous chapters.

This chapter concludes our study of the soft constraint solving framework; in the next one we will study how to program with soft constraints using a declarative language.

[1] We use the adjective enhanced because the $\otimes$ and $\oplus$ operators are *overloaded*. In fact, each of them represents a class of operators $\otimes_D$ and $\oplus_D$ (by varying the domain D).

6. Soft CLP

Overview

The framework presented in the previous chapters shows how the soft constraint idea can give us an easy way to model non-crisp problems. To program applications we need, nevertheless, a language where we can use soft constraints. This is why we extend the Constraint Logic Programming (CLP) formalism in order to handle semiring-based constraints.

This allows us to perform in the same language both constraint solving and optimization. In fact, constraints based on semirings are able to model both classical constraint solving and more sophisticated features like uncertainty, probability, fuzziness, and optimization. We then provide this class of languages with three equivalent semantics: model-theoretic, fix-point, and proof-theoretic, in the style of classical CLP programs.

Constraint logic programming (CLP) [126] languages extend logic programming (LP) by replacing term equalities with constraints and unification with constraint solving. Programming in CLP means choosing a constraint system for a specific class of constraints (for example, linear arithmetic constraints, or finite domain constraints) and embedding it into a logic programming engine. This approach is very flexible since one can choose among many constraint systems without changing the overall programming language, and has shown to be very successful in specifying and solving complex problems in terms of constraints of various kind [190]. It can, however, only handle classical constraint solving. Thus, it is natural to try to extend the CLP formalism in order to also be able to handle soft constraints. In fact, this new programming paradigm, which we will call SCLP (for Semiring-based CLP, or also Soft CLP), has the advantage of treating in a uniform way, and with the same underlying machinery, all constraints that can be seen as instances of the semiring-based approach: from optimization to satisfaction problems, from fuzzy to probabilistic, prioritised, or uncertain constraints, and also multi-criteria problems, without losing the ability to treat and solve classical hard constraints. This leads to a high-level declarative programming formalism where real-life problems involving constraints of all these kinds can be easily modeled and solved.

In passing from CLP to SCLP languages, we will replace classical constraints with the more general SCSP constraints. By doing this, from a technical point of view, we have to modify the notions of interpretation, model, model intersection, and others, since we have to take into account the semiring operations and not the usual CLP operations. For example, while CLP interpretations associate a truth value (either *true* or *false*) to each ground atom, in this case ground atoms must be given one of the elements of the semiring. Furthermore, whereas in CLP the value associated with an existentially quantified atom is the *logical or* among the truth values associated to each of its instantiations, here we have to replace the *or* with another operation that refers to one of the semiring operations.

After describing the syntax of SCLP programs, we will define three equivalent semantics for such languages: model-theoretic, fix-point, and operational. These semantics are conservative extensions of the corresponding ones for LP, since by choosing a particular semiring (the one with just two elements, *true* and *false*, and the logical *and* and *or* as the two semiring operations) we get exactly the LP semantics. The extension is in some cases predictable but it possesses some crucial new features. For example, the presence of a partial order among the semiring elements (and not a *total* order like it is in the LP/CLP case, where we just have two comparable elements) brings some conceptual complexity in some aspects of the semantics. In fact, in the operational semantics, there could be two refutations for a goal leading to different semiring elements that are not comparable in the partial order. In this case, these elements have to be combined in order to get the solution corresponding to the given goal, and their combination could not be reachable by any derivation path in the search tree. This means that any constructive way to get such a solution by visiting the search tree would have to follow all the incomparable paths before being able to find the correct answer. In practice, however, classical branch and bound techniques can be adapted to this framework to cut some useless branches.

We also show the equivalence of the three semantics. In particular, we prove that, given the set of all refutations starting from a given goal, it is possible to derive the declarative meaning of both the existential closure of the goal and its universal closure.

Additionally, we investigate the decidability of the semantics of SCLP programs, obtaining an interesting semi-decidability result: if a goal has a semiring value greater than, or greater than or equal to, a certain value in the semiring, then we can discover this in finite time. Moreover, for SCLP programs without functions, the problem is completely decidable: the semantics of a goal can be computed in finite and bounded time. In fact, in this case we can consider only a finite number of finite and bounded-length refutations (see Section 6.6): infinite refutations do not bring more information due to the properties of the semiring operations. Notice that the absence of functions is obviously a restriction, however, not all sources of infiniteness are taken away, since nothing is said about the semiring, which could still be infinite.

The chapter is organized as follows. Section 6.1 defines the syntax of SCLP programs. Afterwards, sections 6.2, 6.3, and 6.4 provide SCLP programs with a model-theoretic, a fix-point, and an operational semantics, respectively. Then, Section 6.5 presents a semi-decidability result for SCLP programs, and Section 6.6 adds some more decidability results that hold for programs without functions. Section 6.7, instead, presents an abstract model for the operational semantics of the SCLP language without functions, using the *Gurevich's Abstract State Machines* (ASMs). Finally, Section 6.8 concludes the chapter by discussing the relationship with related work. The SCLP framework described in this chapter appeared in [46, 50].

The chapter is based on the results introduced in [46,53] and extended in [50].

6.1 Syntax of SCLP Programs

For readers familiar with Constraint Logic Programming (CLP) programs, we can say that SCLP(S) programs (also written SCLP when the semiring is obvious or not important) are just CLP programs [126] where constraints are defined over a certain c-semiring $S = \langle A, +, \times, \mathbf{0}, \mathbf{1} \rangle$.

As usual, a program is a set of *clauses*. Each clause is composed by a *head* and a *body*. The head is just an *atom* and the body is either a collection of atoms, or a value of the semiring, or a special symbol ($\Box$) to denote that it is empty. Clauses where the body is empty or it is just a semiring element are called *facts* and define predicates which represent constraints. When the body is empty, we interpret it as having the best semiring element (that is, $\mathbf{1}$).

Atoms are n-ary predicate symbols followed by a tuple of n *terms*. Each term is either a constant or a variable or an n-ary function symbol followed by n terms. *Ground* terms are terms without variables. Finally, a *goal* is a collection of atoms.

The BNF for this syntax follows in table 6.1.

As an example, consider the following SCLP(S) program, represented in Table 6.2 where the semiring is $S = \langle [0, 1], max, min, 0, 1 \rangle$. We recall that this is the fuzzy semiring, where tuples of values are given values between 0 and 1, and constraints are combined via the min operator and compared via the max operator. Note that the ordering $\leq_S$ in this semiring coincides with the $\leq$ ordering over the reals in $[0, 1]$.

Our example is a generalization of the usual n-queens problem, which can be found for example in [121]. The classical formulation requires that n queens are placed on a $n \times n$ chessboard in such a way that they do not attack each other. In our formulation, we allow also attacking queens but we give a higher preference to solutions where queens attacking each other are farther apart. Thus, if there are solutions where no queens attack each other, these solutions will remain the best ones. But there are also other solutions, and among these additional

$P ::$	$CL \mid CL, P$
$CL ::$	$H : -B$
$H ::$	AT where AT is the category of atoms
$LAT ::$	$\Box \mid LAT'$
$LAT' ::$	$AT \mid AT, LAT'$
$B ::$	$LAT \mid \mathbf{a}$ where $\mathbf{a} \in A$
$G ::$	$: -LAT$

Table 6.1. BNF for the SCLP syntax.

solutions the best ones are those where the queens attacking each other are as far apart as possible.

We assume here that the reader is familiar at least with the logic programming [135] concepts and notation.

In this program, the n queens are represented by a list of n variables, say `[X1,...,Xn]`, where variable `Xi` represents the queen in column i and its value represents the row index where this queen is located. This is one of the usual formulations of the n-queens problem, and it is worth noticing that by adopting this formulation we assume that different queens are in different columns.

The first predicate, `myqueens/2`, traverses the whole list of n variables and sets the constraints between any pair of queen, by using the predicate `noattack/4`, which sets the constraints between any queen and all the queens in subsequent columns. The actual constraints are set by predicates `row/4`, `diag1/4`, and `diag2/4`. In the classical (hard) formulation, these predicates basically say that different queens cannot stay in the same row, nor in the same diagonal. In our case, these constraints are made soft by also accepting situations where different queens are on the same row or diagonal. For example, for the row constraint, we give the semiring value Nb/N to two queens in the same row, where Nb is the distance between the two queens (that is, the number of columns between them) and N is the total number of columns. This means that the farther apart the two queens are, the higher this value will be. The same reasoning holds also for the two diagonals, except that the presence of both queens X and Y on the same diagonal corresponds to having $Y = X + Nb$ or $Y = X - Nb$.

Notice that each solution of this generalized n-queens problem has a semiring value which is obtained by minimizing the semiring values of all its constraints. This comes from the choice of the fuzzy semiring, where the multiplicative operation is the min. Therefore, if a solution contains three pairs of attacking queens,

```
myqueens([],N).
myqueens([X|Y],N) :-
      noattack(X,Y,N), myqueens(Y,N).
noattack(X,Xs,N) :-
      noattack(X,Xs,N,1).
noattack(X,[],N,Nb).
noattack(X,[Y|Ys],N,Nb) :-
      row(X,Y,N,Nb), diag1(X,Y,N,Nb), diag2(X,Y,N,Nb),
      noattack(X,Ys,N,Nb+1).
row(X,X,N,Nb) :- Nb/N.
row(X,Y,N,Nb) :- different(X,Y).
diag1(X,X+Nb,N,Nb) :- Nb/N.
diag1(X,Y,N,Nb) :- different(Y,X+Nb).
diag2(X,X-Nb,N,Nb) :- Nb/N.
diag2(X,Y,N,Nb) :- different(Y,X-Nb).
```

Table 6.2. Soft n-queens problem.

each of such pairs will have a semiring value given by one of the clauses defining predicates `row/4`, `diag1/4`, or `diag2/4` (proportional to the distance between the two queens), and then the value of this solution will be the minimum among such three values. Different solutions are then ordered using the other semiring operation, which in this case is the max. Note that this same program can also be used with a different semiring, obtaining a different way of computing a solution and a different ordering. For example, we could have chosen the semiring $\{\mathcal{R} \cup +\infty, min, +, +\infty, 0\}$, where the value of each solution would have been obtained by summing the values of each attacking pair, and solutions would have been compared using the min operator.

To use this program over a specific value for N, we need to add some clauses to set the domain for the constraint variables (from 1 to N) and to define predicate `different/2`. For example, for $N = 5$, we have to write the clauses represented in Table 6.3:

We now anticipate the behavior of this SCLP program, which obeys the semantic development of the future sections.

– Given the goal `:- fivequeens(L).`, the program will instantiate L to be a list of 5 values, indicating the row positions of the 5 queens. Since there are solutions where queens do not attack each other, the solution returned by the program will be one of these, with semiring value 1.
– Given the goal `:- fivequeens([1,3,X3,X4,X5]).`, the program will return a position for each queen such that the first queen is in row 1 and the second one in row 3. Given these additional constraints (i.e., the positions of the first two queens), it is possible that all solutions will have some attacking pairs of queens. Then, among these possible solutions, the program will return one with the highest level of preference, which means that the attacking queens are as far apart as possible.
– Assume to delete one or more of the facts defining predicate `domain5`. Then, if we give the goal `:- fivequeens(L).`, it means that we have five queens (thus five columns) but a smaller number of domain elements (thus rows). Even in this case, it is possible that all solutions will have some attacking

```
fivequeens([X1,X2,X3,X4,X5]) :-
      domain5(X1), domain5(X2), domain5(X3),
      domain5(X4), domain5(X5),
      myqueens([X1,X2,X3,X4,X5],5).

domain5(a) :- 1.
```
(for all $a \in \{1, \ldots, 5\}$)

```
different(a,b) :- 1.
```
(for all $a, b \in \{1, \ldots, 5\}$ such that $a \neq b$)

Table 6.3. Clauses for the 5-queens problem.

pairs of queens. Thus the program will return one where the attacking queens are as far apart as possible.

While this example of an SCLP program uses semiring values to be able to find the best quasi-solutions in an otherwise over-constrained problem, other examples exploit the power of the chosen semiring to express features that are intrinsic to the considered problem. Consider the problem that can arise when a client in a restaurant wants to select items from the menu in a way that his/her preferences over the combinations of drinks and dishes are satisfied in the best way. In this example, each combination (for example, beer and pizza, or white wine and fish) is associated with a level of preference, according to the taste of the client. Then, solving the problem means finding the menu with the highest level of preference. As in the previous example, here the semiring that is used is the fuzzy one, but its role is definitely different. This menu problem is one of those that have been used to show the expressive power of the clp(fd,S) system [112], a very general implementation of SCLP programming[1].

Notice that, by just changing the semiring, but maintaining the same program structure of the menu example, we could model rather different situations. Consider for example the situation in which a conference organizer has to decide the menu for the conference dinner, trying to satisfy the participants as much as possible. Of course the organizer cannot ask for their preferences, but can reason with probabilities. Therefore, he/she can associate, with each combination of drink and dish, the estimated probability that it will please the conference attendees. Then, solving the problem means finding the menu which has the highest probability to please the participants. To model this situation, it is enough to keep the same program above (modulo the new semiring values), but choose the semiring $\langle [0,1], max, \times, 0, 1\rangle$, which can represent probabilities combined via $\times$ (assuming their independence) and compared via the max operator.

6.2 Model-Theoretic Semantics

In this section we will generalize the usual development of the model-theoretic semantics in logic programming [135] to be able to deal correctly with semiring values. The main generalization will involve the assignment of semiring values to atoms and formulas, instead of truth values. As usual, we will just consider Herbrand interpretations in the following, which, however, we will call just interpretations for sake of conciseness.

Definition 6.2.1. (PRE-INTERPRETATION) *A pre-interpretation maps each ground term in a program into a chosen domain. More precisely, it consists of a domain D plus a mapping from each constant to an element of D and, for each n-ary function, a mapping from D^n to D.*

[1] We will give a short description of the `SCLP(FD,S)` framework in a paragraph in Section 6.8.

Notice that the domain D of the pre-interpretations contains the domain of the constraints.

Definition 6.2.2. (INTERPRETATION) *An interpretation I is a pre-interpretation plus a function, which takes a predicate and an instantiation of its arguments (that is, a ground atom), and returns an element of the semiring:*

$$I : \bigcup_n (P_n \to (D^n \to A)),$$

where P_n is the set of n-ary predicates and A is the carrier set of the semiring.

This notion of interpretation can now be extended and used to associate elements of the semiring also to formulas that are more complex than ground atoms. In the following, this extension of an interpretation I will be called an *interpretation* and denoted by I as well, since the extension is uniquely determined. When we will want to consider the restriction of an interpretation I to ground atoms we will sometimes write $GA(I)$.

- The value associated with a formula of the form $F = \exists x.F'(x)$ is computed by considering the lub of the values associated with all ground formulas $F'(x/d)$, where d is any domain element. That is, $I(F) = lub\{I(F'(d))$, for all $d \in D\}$. Formulas of this kind occur in SCLP languages since variables appearing in the body of a clause but not in its head are considered to be existentially quantified. For example, in the special case of logic programming the clause `p(a) :- q(X,a)` is just a shorthand for the formula $p(a) \leftarrow \exists x.q(x,a)$.
- The value associated with a formula of the form $F = \forall x.F'(x)$ is computed by considering the greatest lower bound (glb) of the values associated with all the ground formulas $F'(x/d)$, where d is any domain element. That is, $I(F) = glb\{I(F'(d))$,for all $d \in D\}$. Formulas of this kind occur when a variable appears in the head of a clause. In fact, for example, in logic programming, a clause like `p(X) :- q(X,a)` is a shorthand for the formula $\forall x.(p(x) \leftarrow q(x,a))$.
- The value associated with a conjunction of atomic formulas of the form (A, B) is the semiring product of the values associated to A and B: $I(A, B) = I(A) \times I(B)$. Such formulas appear in the body of the clauses, when the body contains more than one atom.
- For any semiring element a, $I(a) = a$. Such elements appear in the body of the facts.

Note that the meaning associated to formulas by function I coincides with the usual logic programming interpretation [135] when considering constraints over the semiring $S_{CSP} = \langle\{true, false\}, \vee, \wedge, false, true\rangle$. In fact, in this case the ordering $\leq_S$ is defined by $false \leq_S true$, the lub operation of the lattice $\langle\{true, false\}, \leq_S\rangle$ is $\vee$, and the glb is $\wedge$. Thus, for example, $I(\exists x.A(x)) = lub\{I(A(d))$, for all $d \in D\} = \vee\{I(A(d))$, for all $d \in D\}$. Thus, it is enough that one of the $A(d)$ is assigned the value *true* that the value associated to the whole formula $\exists x.A(x)$ is *true*. Note also that in this special instance the lub and glb of the lattice coincide with the two semiring operations, but this is not true in general for the multiplicative operation.

Definition 6.2.3. (CLAUSE SATISFACTION) *Given a clause of the form $H :- B$ and an interpretation I, we say that the clause is satisfied in I if and only if, for any ground instantiation of H, say $H\theta$, we have that $I(H\theta) \geq_S I(\exists B\theta)$.*

Note that the existential quantification over the body $B\theta$ is needed, since there may be variables in B that do not appear in H. Thus $B\theta$ could be not ground. This definition of clause satisfiability is consistent with the usual treatment of clauses in logic programming, where a clause is considered to be satisfied if the body logically implies the head, and by noting that logical implication in the semiring S_{CSP} coincides with the ordering $\leq_{S_{CSP}}$.

Our running example As our running example in this chapter, we will use the program represented in Table 6.4, which is not as expressive as the program in the previous section but is simple enough to be analyzed in detail. This is an SCLP(S) program over the semiring $S = \langle N \cup \{+\infty\}, min, +, +\infty, 0\rangle$, where N is the set of non-negative integers. This semiring allows to model constraint optimization problems where each tuple of values is assigned an integer, to be interpreted as its cost, constraints are combined by summing their costs, and are compared by using the min operator. Note that the ordering $\leq_S$ in this semiring coincides with the $\geq$ ordering over integers.

In this program, the constraints are represented by predicates `t` and `r`. The intuitive meaning of a semiring value like 3 associated to the atom $r(a)$ is that $r(a)$ costs 3 units. Thus the set $N \cup \{+\infty\}$ contains all possible costs, and the choice of the two operations min and $+$ implies that we intend to minimize the sum of the costs. This gives us the possibility to select the atom instantiation that gives the minimal cost overall. □

As an example of clause satisfiability, consider the following four clauses:

- the clause `p(a) :- q(b)` is satisfied in I if $I(p(a)) \geq_S I(q(b))$;
- the clause `p(X) :- q(X,a)` is satisfied if $\forall x.(I(p(x)) \geq_S I(q(x,a)))$;
- the clause `p(a) :- q(X,a)` is satisfied if $I(p(a)) \geq_S I(\exists x.q(x,a))$;
- the clause `p(X) :- q(X,Y)` is satisfied if $\forall x.(I(p(x)) \geq_S I(\exists y.q(x,y)))$.

```
s(X)   :- p(X,Y).
p(a,b) :- q(a).
p(a,c) :- r(a).
q(a)   :- t(a).
t(a)   :- 2.
r(a)   :- 3.
```

Table 6.4. Our running example.

As in logic programming, an interpretation I is a *model* for a program P if all clauses of P are satisfied in I. Given a program and all its models, one would like to identify a unique single model as the representative one. In logic programming this is done by considering the minimal model [135], which is obtained by intersecting all the models of the program. This works because models in logic programming are assimilable to sets of ground atoms, those with associated value *true*. Here we follow the same approach, but we have to generalize the notion of intersection of two models, written as "$\circ$", as their greatest lower bound in the lattice $\langle A, \leq_S \rangle$.

Definition 6.2.4. (MODEL INTERSECTION) *Consider an SCLP program over the c-semiring $\langle A, +, \times, \mathbf{0}, \mathbf{1} \rangle$, the corresponding ordering $\leq_S$ and the lattice $\langle A, \leq_S \rangle$. For every ground atomic formula F and a family of models $\{M_i\}_{i \in I}$, we define $\circ_{i \in I} M_i(F) = glb_{i \in I}\{M_i(F)\}$, where glb is the greatest lower bound over the lattice $\langle A, \leq_S \rangle$.*

Theorem 6.2.1. (MODEL INTERSECTION) *Consider a family of models $\{M_i\}_{i \in I}$ for a CLP(S,D) program P. Then $\circ_{i \in I} M_i$ is a model for P as well.*

Proof. Since M_i is a model for P for all $i \in I$, it must be that, for every clause $H : -B$, and for all θ such that $H\theta$ is ground, $M_i(\exists B\theta) \leq_S M_i(H\theta)$. Consider now the model $M = \circ_{i \in I} M_i$. We need to prove that, for all $H : -B$, and for all θ such that $H\theta$ is ground, also $M(\exists B\theta) \leq_S M(H\theta)$ holds.

Without loss of generality, assume that $B = A_1, A_2$. Thus, for all $i \in I$, $M_i(\exists B\theta) = lub\{M_i(B\theta\theta')$, for all θ' such that $B\theta\theta'$ is ground$\} = lub\{ M_i(A_1\theta\theta') \times M_i(A_2\theta\theta')$, for all θ' such that $B\theta\theta'$ is ground$\}$. Moreover, $M(\exists B\theta) = lub\{ M(B\theta\theta')$, for all θ' such that $B\theta\theta'$ is ground$\} = lub\{M(A_1\theta\theta') \times M(A_2\theta\theta')$, for all θ' such that $B\theta\theta'$ is ground$\}= lub\{glb_{i \in I}\{ M_i(A_1\theta\theta')\} \times glb_{i \in I}\{ M_i(A_2\theta\theta')\}$ for all $\theta'\}$. Also, $M(H\theta) = glb_{i \in I}\{M_i(H\theta)\}$.

Consider any model M_j with $j \in I$. Since $glb_{i \in I}\{M_i(A_1\theta\theta')\} \leq_S M_j(A_1\theta\theta')$ and $glb_{i \in I}\{ M_i(A_2\theta\theta')\} \leq_S M_j(A_2\theta\theta')$ by definition of glb, and recalling that $\times$ is monotone, we have that $M(\exists B\theta) \leq_S M_j(\exists B\theta)$. By transitivity of $\leq_S$, we thus get $M(\exists B\theta) \leq_S M_j(H\theta)$. Since $M(H\theta)$ is the glb of all $M_i(H\theta)$ for $i \in I$, and since the glb of a set of elements is the greatest among the elements which are smaller than all of them, we have that $M(\exists B\theta) \leq_S M(H\theta)$.

It is easy to see that the operation of model intersection is associative, idempotent, and commutative.

Definition 6.2.5. (MINIMAL MODEL) *Given a program P and the set of all its models, its* minimal model *is obtained by intersecting all models: $M_P = \circ(\{M \mid M \text{ is a model for } P\})$. The* model-theoretic semantics *of a program P is its minimal model, M_P.*

Consider our running example program P. The minimal model M_P for such a program must assign an integer to each formula, and when restricted to ground atoms it is the following function: $M_P(t(a)) = 2$, $M_P(q(a)) = 2$, $M_P(r(a)) = 3$, $M_P(p(a,c)) = 3$, $M_P(p(a,b)) = 2$, $M_P(s(a)) = min(2,3) = 2$. For each atom different from the ones considered above, M_P returns $+\infty$. To explain why function M_P returns these values, we give some examples:

– Any model must assign to $t(a)$ a semiring value smaller (that is, better) than 2, because of the clause `t(a) :- 2`. Since M_P is the minimal model, it must assign to $t(a)$ the glb (that is, the max) of all such values, that is, 2.
– The value assigned to $p(a,c)$ must be smaller than that of $r(a)$, which in turn must be smaller than 3. Being in the minimal model, we have that the value of $r(a)$ is exactly 3, and the value of $p(a,c)$ is again 3. The same reasoning holds also for $p(a,b)$, whose value is 2. Instead, $p(a,v)$, for any $v \neq b,c$, gets the value $+\infty$, because a model can give any value to $p(a,v)$, since there is no clause about it, and thus the minimal model gives to it the glb (that is, the max) of all values, that is, the worst element of the semiring.
– For $s(a)$, we know that every model must assign to it a value smaller than the value assigned to $\exists y.p(a,y)$. Now, for any model M, $M(\exists y.p(a,y))$ is the lub (that is, the min) of all the values assigned by M to $p(a,v)$ for any v in the domain. We know that $p(a,b)$ has value 2, $p(a,c)$ has value 3, and any $p(a,v)$, with $v \neq b,c$, has value $+\infty$. Therefore the lub of all such values is 2. Thus any model must assign to $s(a)$ a value smaller than 2, and the minimal model M_P must give it value 2.

For the same program, it is also useful to notice which semiring value is assigned to formulas like $\forall y.p(a,y)$ by the function M_P. In fact, this is one of the kinds of formulas we will consider when studying the relationship between the operational and the model-theoretic semantics, in Section 6.4. By definition, to get $M_P(\forall y.p(a,y))$ we must compute the glb of all the semiring values assigned by M_P to the ground atoms of the form $p(a,v)$ where v is any element of the domain D. We know, by the paragraph above, that $M_P(p(a,c)) = 3$, $M_P(p(a,b)) = 2$, and $M_P(p(a,v)) = +\infty$ if v is different from both b and c. Thus the glb (that is, the max) of all these values is $+\infty$. Therefore $M_P(\forall y.p(a,y)) = +\infty$.

6.3 Fix-Point Semantics

In order to describe the fix-point semantics, we need to define the operator T_P which extends the one used in logic programming [135]. We will do that by following the same approach as in the previous section. The resulting operator maps interpretations into interpretations, that is, $T_P : IS_P \rightarrow IS_P$, where IS_P is the set of all interpretations for P.

Definition 6.3.1. (T_P OPERATOR) *Given an interpretation I and a ground atom A, assume that program P contains k clauses defining the predicate in A. Clause i is of the form $A : -B_1^i, \ldots, B_{n_i}^i$. Then*

$$T_P(I)(A) = \textstyle\sum_{i=1}^{k}(\prod_{j=1}^{n_i} I(B_j^i)).$$

This function coincides with the usual immediate consequence operator of logic programming (see [135]) when considering the semiring S_{CSP}.

Consider now an ordering $\preceq$ among interpretations which respects the semiring ordering.

Definition 6.3.2. (PARTIAL ORDER OF INTERPRETATIONS) *Given a program P and the set of all its interpretations IS_P, we define the structure $\langle IS_P, \preceq\rangle$, where for any $I_1, I_2 \in IS_P$, $I_1 \preceq I_2$ if $I_1(A) \leq_S I_2(A)$ for any ground atom A.*

It is easy to see that $\langle IS_P, \preceq\rangle$ is a complete partial order, whose greatest lower bound coincides with the glb operation in the lattice A (suitably extended to interpretations). It is also possible to prove that function T_P is monotone and continuous over the complete partial order $\langle IS_P, \preceq\rangle$.

By using these properties, classical results on partial orders [188] allow us to conclude the following:

- T_P has a least fix-point, $lfp(T_P)$, which coincides with $glb(\{I \mid T_P(I) \preceq I\})$;
- the least fix-point of T_P can be obtained by computing $T_P \uparrow \omega$. This means starting the application of T_P from the bottom of the partial order of interpretations, called I_0, and then repeatedly applying T_P until a fix-point.

Consider again our running example program. We recall that in this specific case the semiring is $S = \langle N \cup \{+\infty\}, min, +, +\infty, 0\rangle$ and $D = \{a, b, c\}$. Thus function T_P is:

$$T_P(I)(A) = min\{\textstyle\sum_{j=1}^{n_1} I(B_j^1), \ldots, \sum_{j=1}^{n_k} I(B_j^k)\}.$$

In this semiring the bottom interpretation I_0 is the interpretation that maps each semiring element into itself and each ground atom into the bottom of the lattice associated to the semiring, that is, $+\infty$. Note that we slightly abused the notation since interpretations are functions whose domain contains only ground atoms (see Section 6.2), while here we also included semiring elements. This simplifies the definition of I_0; however, it is possible to obtain the same result with a more complex definition of I_0 which satisfies the definition of interpretation. Given I_0, we obtain I_1 by applying function T_P above. For example, $I_1(r(a)) = +3$. Instead, $I_1(p(a, c)) = +\infty$, and $I_2(p(a, c)) = I_1(r(a)) = +3$. The Table 6.5 gives the value associated by the interpretation I_i to each ground atom. Some of the atoms[2] are not listed because each interpretation I_i gives them value $+\infty$. All interpretation I_i with $i > 4$ coincide with I_4, thus I_4 is the fix-point of T_P.

The most interesting case is the computation of the value associated to $s(a)$. In fact, $I_3(s(a)) = min\{I_2(p(a, a)),\ I_2(p(a, b)),\ I_2(p(a, c))\} = min\{+\infty, +\infty, 3\} = 3$. Instead, $I_4(s(a)) = min\{I_3(p(a, a)),\ I_3(p(a, b)),\ I_3(p(a, c))\} = min\{+\infty, 2, 3\} = 2$. Note that the clause `s(X) :- p(X,Y)` is considered equivalent to all its instantiations. In particular, when $x = a$, we have the three clauses

[2] Actually, an infinite number of them.

	I_1	I_2	I_3	I_4
t(a)	2	2	2	2
r(a)	3	3	3	3
q(a)	$+\infty$	2	2	2
p(a,c)	$+\infty$	3	3	3
p(a,b)	$+\infty$	$+\infty$	2	2
s(a)	$+\infty$	$+\infty$	3	2
s(b)	$+\infty$	$+\infty$	$+\infty$	$+\infty$
s(c)	$+\infty$	$+\infty$	$+\infty$	$+\infty$

Table 6.5. The T_P operator applied at the running example.

`s(a) :- p(a,a)`, `s(a) :- p(a,b)`, and `s(a) :- p(a,c)`. These are the clauses to consider when computing $I(s(a))$.

We will now prove that the least fix-point of function T_P coincides with the minimal model of program P, when restricted to ground atoms. To do that, we need an intermediate result that shows that the ground models of a given program P are the solutions of the equation $T_P(I) \preceq I$.

Theorem 6.3.1. (MODELS AND T_P) *Given any interpretation I for a program P, I is a ground model for P if and only if $T_P(I) \preceq I$.*

Proof. Consider any ground atom H and assume there are two clauses with H as their head: $H : -B_1$ and $H : -B_2$. By definition of model, each clause $H : -B_i$ is satisfied in I. Thus $I(H) \geq_S I(\exists B_i)$. Now, function T_P assigns to H the sum of the values assigned by I to $\exists B_1$ and $\exists B_2$, thus $T_P(I)(H) = I(\exists B_1) + I(\exists B_2)$. But the $+$ operation coincides with the lub of the semiring, thus any value of the semiring which is greater than both $I(\exists B_1)$ and $I(\exists B_2)$ is also greater than their sum. Therefore $T_P(I)(H) \leq I(H)$. A similar reasoning works also for proving that if $T_P(I)(H) \leq I(H)$ for any ground atom H then I is a model.

Theorem 6.3.2. (GROUND MODELS AND FIX-POINT SEMANTICS) *Given an SCLP program P, we have that $GA(M_P) = lfp(T_P)$*[3].

Proof. By definition of ground minimal model, $GA(M_P) = glb(\{I \mid I$ is a model for $P\})$. By the previous theorem, we get $GA(M_P) = glb(\{I \mid T_P(I) \preceq I\})$. By the classical results cited above [188], this coincides with the least fix-point of T_P.

Notice that the result of Theorem 6.2.1 (that is, the glb of two models is a model) can be proven also by using Theorem 6.3.1 (which states that a model is a prefixpoint of T_P) and by easily showing that the glb of two prefixpoints of a monotone operator (as T_P is) is a prefixpoint as well.

[3] We recall that, for any interpretation I, $GA(I)$ is the restriction of I to ground atoms.

6.4 Proof-Theoretic Semantics

We will define here a proof-theoretic semantics in the style of CLP [126]. We first, however, rewrite the program into a form that allows us to make the semantics treatment more uniform and simpler.

First, we rewrite each clause so that the head is an atom whose arguments are different variables. This means that we must explicitly specify the substitution that was written in the head, by inserting it in the body. That is, given a clause of the form $p(t_1, \ldots, t_n) : -B$, we transform it into $p(x_1, \ldots, x_n) : -\langle B, \theta \rangle$ where $\theta = \{x_1/t_1, \ldots, x_n/t_n\}$. Thus, bodies now have the following syntax: $B1 :: \langle B, \theta \rangle$. We recall that B can be either a collection of atoms or a value of the semiring. To give a uniform representation to bodies, we can define them as triples containing a collection of atoms (possibly empty), a substitution, and a value of the semiring (possibly, $\mathbf{1}$). Thus, bodies are now of the form $B2 :: \langle LAT, \theta, \mathbf{a} \rangle$. If we have a body belonging to the syntactic category $B1$ of the form $\langle \mathbf{a}, \theta \rangle$, we get $\langle \Box, \theta, \mathbf{a} \rangle$. If instead we have $\langle C, \theta \rangle$, where C is a collection of atoms, we get $\langle C, \theta, \mathbf{1} \rangle$. Therefore, clauses have now the syntax $CL1 :: H : -B2$.

Initial goals need to be transformed as well: given a goal $G = (: -C)$, where C is a collection of atoms, we get the goal $G' = (: -\langle C, \varepsilon, \mathbf{1} \rangle)$. The reason why we write the value $\mathbf{1}$ of the semiring is that this element is the unit element w.r.t. the operation we want to perform on it, that is, constraint combination.

In summary, given a SCLP program, we get a program in an intermediate language, whose syntax is as follows:

$B2 ::\ \langle LAT, \theta, \mathbf{a} \rangle$
$CL1 :: H : -B2$
$P1 ::\ CL1 \mid CL1, P1$
$G1 ::\ B2$

Consider again our running example. The transformed program is then

```
s(X)   :-   ⟨ p(X,Y), ε,0⟩.
p(X,Y) :-   ⟨ q(a), {X=a,Y=b}, 0⟩.
p(X,Y) :-   ⟨ r(a), {X=a,Y=c}, 0⟩.
q(X)   :-   ⟨ t(a), {X=a}, 0⟩.
t(X)   :-   ⟨□, {X=a}, 2 ⟩.
r(X)   :-   ⟨□, {X=a}, 3 ⟩.
```

Once we have transformed the given SCLP program into a program in the syntax just given, we can apply the following semantic rule. This rule defines the transitions of a nondeterministic transition system whose states are goals (according to the syntactic category G1).

$$\frac{\begin{array}{c} C = A, Cr \\ Cl = (A' : -\langle C_1, \theta_1, \mathbf{a_1} \rangle) \text{ is a variant of a clause or a fact} \\ \theta' = mgu(A\theta, A'\theta_1) \end{array}}{\langle C, \theta, \mathbf{a} \rangle \xrightarrow{Cl,\theta'} \langle (C_1, Cr), (\theta \circ \theta' \circ \theta_1), \mathbf{a} \times \mathbf{a_1} \rangle}$$

If the current goal contains an atom that unifies with the head of a clause, then we can replace that atom with the body of the considered clause, performing a step similar to the resolution step in CLP. The main difference here is that we must update the third element of the goal, that is, the semiring value associated to the goal: if before the transition this value is $\mathbf{a}$ and the transition uses a clause whose body has value $\mathbf{a_1}$, then the value associated to the new goal is $\mathbf{a} \times \mathbf{a_1}$. The reason for using the $\times$ operation of the semiring is that this is exactly the operation used when accumulating constraints in the SCSP framework.
Notice that we must use a *variant* of the clause involved in the rule, because we need fresh variables to avoid confusion between the variables of the clause and those of the current goal.

Definition 6.4.1. (DERIVATIONS AND REFUTATIONS) *A derivation is a finite or infinite sequence of applications of the above rule. A refutation is a finite derivation whose final goal is of the form $\langle \Box, \theta, \mathbf{a} \rangle$.*

Note that in this chapter we give a simplified view of the solver for soft constraints, where the solver is *implemented* by some clauses in the program (see the last two clauses of our running example). These clauses have a special shape, since they are ground facts. Also, they are in a finite number, since our constraints have finite domains. Therefore, when executing such a program, the soft constraints (like `t` and `r` in our example) are *solved* using these clauses and, thus, they do not appear in the resulting final goal. This allows us to simplify the usual CLP operational semantic rules (like those in [127]), so that, instead of having computation states made by a current goal and a store, we can just have a goal, a substitution for the variables (this would be in the store in CLP) and a semiring value. Therefore, our operational semantic rule and states are consistent with those of CLP, in the case that we have 1) finite domain variables in the constraints, 2) solver specified by program clauses (this again can be done in this straightforward way only because we have finite domains), and 3) soft constraints. While the first two points are special cases, the third is an extension of CLP. Notice also that we can have just one rewriting rule (instead of several rules, as in [127]) because we first rewrite the program into a syntax that makes clauses and facts uniform. A last point to notice is that we do not test at each step if the constraint store is consistent (that is, if the semiring value collected is greater than $\mathbf{0}$) because the notion of inconsistency is not so strong in soft constraints, and we perform this check only at the end.

Definition 6.4.2. (COMPACT REFUTATION SET) *Given an SCLP program P, its compact refutation set $S(P)$ is defined as follows:*

$$S(P) = \langle C, \theta_{|_{var(C)}}, \mathbf{a} \rangle \mid \langle C, \varepsilon, \mathbf{1} \rangle \rightarrow^* \langle \Box, \theta, \mathbf{a} \rangle \}.$$

$S(P)$ contains all triples representing all refutations for the given program. Note that we only record the part of θ which involves the variables in C. Notice also that a triple $\langle C, \theta, \mathbf{a} \rangle$ may represent more than one refutation, but all these refutations start from the same goal C, build the same substitution θ, and generate the same semiring value $\mathbf{a}$. Thus, we do not mind making them indistinguishable.

A property of the set $S(P)$ is that for any triple $\langle C, \theta, \mathbf{a} \rangle$ in this set, there is also the triple $\langle C\theta, \varepsilon, \mathbf{a} \rangle$. Notice that this is a generalized version of the lifting lemma of [135].

Theorem 6.4.1. (GOAL SPECIALIZATION) *Given an SCLP program P and the corresponding set $S(P)$, let us consider any triple $\langle C, \theta, \mathbf{a} \rangle$ in $S(P)$. Then also the triple $\langle C\theta, \varepsilon, \mathbf{a} \rangle$ is in $S(P)$.*

Proof. Triple $\langle C, \theta, \mathbf{a} \rangle$ says that there is a refutation, which starts from C, builds θ, and obtains the semiring value $\mathbf{a}$. If we start from $C\theta$, we can construct a refutation which follows exactly the same steps as the previous one, and thus obtains the same semiring value, since θ is compatible with all these steps.

Our goal now is to study the correspondence between the operational semantics of a goal and its model-theoretic meaning. In particular, given a goal C, we will define the operational semantics of C in two different ways, to model the meaning of both $\forall C$ and $\exists C$.

6.4.1 Universal Closure

Among all refutations represented by the triples in $S(P)$, there are some that have the same first element, say C, and which build the empty substitution, ε, during the computation. These refutations have to be merged by the operational semantics since they represent different alternative branches for goal C, which lead to possibly different semiring values and that hold for any value of the variables of C. Thus, they naturally correspond to the intuitive meaning of $\forall C$.

When merging such elements of $S(P)$, the respective semiring values must be merged as well. This is done by performing the $+$ operation. Actually, we should also merge those refutations that start from different goals, say C and C', such that C' is more instantiated than C. That is, such that there exists a substitution θ such that $C' = C\theta$. In fact, the semiring value obtained from C holds also for all goals that are more instantiated than C. For example, if we have the triples $\langle p(x), \varepsilon, \mathbf{v_1} \rangle$ and $\langle p(a), \varepsilon, \mathbf{v_2} \rangle$, the first refutation says that for all values of x, $p(x)$ gets the semiring value $\mathbf{v_1}$. Thus, also $p(a)$ will get this value. On the other hand, the other refutation says that $p(a)$ gets the value $\mathbf{v_2}$. Therefore, the value to be assigned to $p(a)$ is the lub between $\mathbf{v_1}$ and $\mathbf{v_2}$, which is $\mathbf{v_1} + \mathbf{v_2}$ by definition of $+$. It is possible, however, to show that if there is the triple $\langle C, \varepsilon, \mathbf{v} \rangle$ in $S(P)$, then there is also the triple $\langle C\theta, \varepsilon, \mathbf{v} \rangle$ for any θ. In other words, if we have a refutation starting from C, building the empty substitution and obtaining the semiring value $\mathbf{v}$, then there is also a refutation starting from any goal more instantiated than C (that is, $C\theta$), which follows the same steps as the refutation for C and thus obtains the same semiring value. Therefore, we just need to merge those refutations that start from the same goal.

Theorem 6.4.2. (MORE ABOUT GOAL SPECIALIZATIONS) *Given an SCLP program P and the corresponding set $S(P)$, let us consider any triple $\langle C, \varepsilon, \mathbf{a} \rangle$ in $S(P)$. Then, for any θ, also the triple $\langle C\theta, \varepsilon, \mathbf{a} \rangle$ is in $S(P)$.*

Proof. The refutation represented by the triple $\langle C, \varepsilon, \mathbf{a} \rangle$ does not bind any variable present in C. By starting from a goal $C\theta$, which is more instantiated than C, we can follow exactly the same steps as in the refutation for C. In fact, at each step we can use the same clause as before, since we know that such a step did not bind the variables in C.

Let us now define a function $OS1_P$ (for Operational Semantics), which, given a goal, returns a value of the semiring by looking at all refutations for that goal that build the empty substitution.

Definition 6.4.3. (FUNCTION $OS1_P$) *Given an SCLP program P, function $OS1_P : LAT \to A$, where LAT is the set of conjunctions of atoms and A is the semiring set, is defined as follows: $OS1_P(C) = \sum_{\langle C, \varepsilon, \mathbf{a} \rangle \in S(P)} \mathbf{a}$.*

Note that, when there is no triple $\langle C, \varepsilon, \mathbf{a} \rangle \in S(P)$, function $OS1_P(C)$ returns the unit element of the $+$ operation, that is, $\mathbf{0}$. This is reasonable, since the absence of such triples means that there is no refutation starting from C and builds the empty substitution.

Notice that, if $OS1_P(C) = \mathbf{a}$, there is not necessarily a refutation starting from C and obtaining the semiring value $\mathbf{a}$. In fact, as noted above, the value $\mathbf{a}$ may have been obtained by combining several refutations, which may be incomparable with respect to $\leq_S$.

Let us now consider some examples of goal refutations and their operational semantics via function $OS1$. By considering the goal $\langle s(a), \{\varepsilon\}, 0 \rangle$ in our running example, we get two refutations, which end respectively with the goals $\langle \square, \{\varepsilon\}, 2 \rangle$ and $\langle \square, \{\varepsilon\}, 3 \rangle$. Thus $S(P)$ contains the triples $\langle s(a), \{\varepsilon\}, 2 \rangle$ and $\langle s(a), \{\varepsilon\}, 3 \rangle$. Therefore $OS1_P(s(a)) = min(2,3) = 2$. Instead, $OS1_P(s(b)) = +\infty$ (which is the bottom of the semiring), since there is no refutation for $\langle s(b), \{\varepsilon\}, 0 \rangle$. Consider now the goal $\langle s(x), \varepsilon, 0 \rangle$. In this case, we get two refutations with the same final goals as above, and thus $S(P)$ contains the same elements as above. More precisely, $S(P)$ does not contain any triple of the form $\langle s(x), \varepsilon, \mathbf{a} \rangle$, and therefore $OS1_P(s(x)) = +\infty$.

A more complex example is related to the goal $\langle p(a,y), \{\varepsilon\}, 0 \rangle$, which has two refutations ending with the goals $\langle \square, \{y = b\}, 2 \rangle$ and $\langle \square, \{y = c\}, 3 \rangle$, which are therefore represented in $S(P)$ by the triples $\langle p(a,y), \{y = b\}, 2 \rangle$ and $\langle p(a,y), \{y = c\}, 3 \rangle$. Therefore, the operational semantics of $p(a,y)$ is $OS1_P(p(a,y)) = +\infty$, even if there are refutations starting from $p(a,y)$ and obtaining a semiring value smaller than $+\infty$. Formally, this is due to the fact that such refutations all build a substitution different from ε, and thus are not considered by function $OS1_P$. Intuitively, the fact that the operational semantics assigns to $p(a,y)$ the worst semiring value, even if there are refutations that lead to better values, can be explained by considering that by only looking at the refutations, as the operational semantics does, there will be other domain values, say d, for which $p(a, d)$ does not hold. We will come back to this issue in the following paragraph, when we will formally compare the operational and the model-theoretic semantics. We can, however, already notice that this behavior leads to the same result (that is, the same semiring value) we computed via the model-theoretic tools (that is, function M_P) at the end of Section 6.2 for the formula $\forall y.p(a, y)$.

We now prove formally that in general the operational semantics of SCLP programs, as just defined via function $OS1_P$, for each goal C, computes the same semiring value as the model-theoretic semantics of $\forall C$, defined in Section 6.2, and thus also as the fix-point semantics of Section 6.3 when applied to any ground instantiation of C.

Before stating and proving the main theorem, however, it is useful to prove two lemmas, which will be used in the proof of the theorem. The first lemma basically proves that the operational semantics of a collection of ground atoms is the same as the multiplication of the operational semantics of the single atoms, while the second one uses the fact that we have an infinite domain, but a finite number of domain elements in the program, to prove that the operational semantics of a goal coincides with the glb of the operational semantics of all its ground instances.

Lemma 6.4.1. (DISTRIBUTION) *Given an SCLP program P, consider two collections of atoms C_1 and C_2. Then we have that $OS1_P(C_1, C_2) = OS1_P(C_1) \times OS1_P(C_2)$.*

Proof. By definition of $OS1_P$, we have that $OS1_P(C_1) \times OS1_P(C_2) = (\sum_{\langle C_1,\epsilon,\mathbf{a}\rangle \in S(P)} \mathbf{a}) \times (\sum_{\langle C_2,\epsilon,\mathbf{b}\rangle \in S(P)} \mathbf{b})$. Let us assume that we have n elements of the form $\langle C_1, \epsilon, \mathbf{a}\rangle$ and m elements of the form $\langle C_2, \epsilon, \mathbf{b}\rangle$ in $S'(P)$. Therefore, we will consider the semiring values $\mathbf{a_1}, \ldots, \mathbf{a_n}$ and $\mathbf{b_1}, \ldots, \mathbf{b_m}$. Notice that n and m may be infinite; however, since we work with complete lattices, the lub ($+$) and glb are defined also in the infinite case. Thus, we have $OS1_P(C_1) \times OS1_P(C_2) = (\sum_{i=1,\ldots,n} \mathbf{a_i}) \times (\sum_{j=1,\ldots,m} \mathbf{b_j})$. Now, since $\times$ distributes over $+$ (by the definition of semiring), this can be rewritten as $\sum_{i=1,\ldots,n,j=1,\ldots,m}(\mathbf{a_i} \times \mathbf{b_j})$. Consider now any refutation for the goal (C_1, C_2) which does not instantiate any variable, and assume, without loss of generality, that its selection strategy first "consumes" C_1 and then C_2. Then, after C_1 has been "consumed", we have a goal of the form $\langle C_2, \epsilon, \mathbf{a_i}\rangle$. Then the refutation continues by "consuming" C_2, leading to a final goal of the form $\langle \Box, \epsilon, \mathbf{a_i} \times \mathbf{b_j}\rangle$. Now, $OS1_P(C_1, C_2)$ just sums all the semiring values of all such simple refutations, therefore, we have that $OS1_P(C_1, C_2) = \sum_{i,j}(\mathbf{a_i} \times \mathbf{b_j})$, which is the same as $OS1_P(C_1) \times OS1_P(C_2)$ for what we have said before.

Lemma 6.4.2. (INFINITE DOMAIN) *Given an SCLP program P, consider any collection of atoms C. Then $OS1_P(C) = glb\{OS1_P(C\theta)$, for all θ such that $C\theta$ is ground$\}$.*

Proof. Given a goal C, function $OS1_P$ returns the sum of all the semiring values obtained by all those refutations of C which build the empty substitution. Let us call the set of such refutations $E(C)$. Since $C\theta$ is more instantiated than C, then the refutations of $C\theta$ will be able to use more clauses than those for C. In fact, if the head of a clause is as instantiated as a subgoal of $C\theta$ but more instantiated than the corresponding subgoal of C, then a refutation of $C\theta$ will be able to use it without building a non-empty substitution, while any refutation of C using such a clause will have to add a non-empty substitution to the substitution of the current goal. Therefore, we have that $E(C) \subseteq E(C\theta)$, which, by definition of $OS1_P$ and by the fact that the $+$ operation is the lub, means that $OS1_P(C) \leq_S OS1_P(C\theta)$. Now, this holds for any θ, but it is possible to show that for some θ we actually have $OS1_P(C) = OS1_P(C\theta)$. To do that, it is enough to consider a θ which binds all variables of C to domain elements not present in P. In this way, the refutations of $C\theta$ will not be able to use more clauses than those of C. Therefore, the glb of all $OS1_P(C\theta)$ for all θ is actually equal to $OS1_P(C)$, since we know that $OS1_P(C)$ is smaller than (or equal to) all elements $OS1_P(C\theta)$ and that it is actually equal to one of them.

Theorem 6.4.3. (OPERATIONAL MEANING OF $\forall C$) *Given a SCLP program P, consider a collection of atoms C. Then we have that $M_P(\forall C) = OS1_P(C)$.*

Proof. If $OS1_P(C) = \mathbf{a}$, it means that $S(P)$ contains triples of the form $\langle C, \varepsilon, \mathbf{a_i} \rangle$, for $i = 1, \ldots, n$, such that $\sum_{i=1,\ldots,n} \mathbf{a_i} = \mathbf{a}$. Again, we recall that n may be infinite. We work with complete lattices, however, and thus both infinite lub (that is, sum) and infinite glb are defined.

We will prove the statement of the theorem by induction on the length of the longest of the refutations corresponding to such triples.

- **base case:** If all refutations $\langle C, \varepsilon, \mathbf{a_i} \rangle$ have length 1, it means that there are n facts in the program of the form $C' : -\langle \Box, \theta_i, \mathbf{a_i} \rangle$, where C' contains only variables and $C'\theta_i$ is equally or less instantiated than C. In fact, if the head of such facts would be more instantiated than C, then we would have a substitution different from ε in the refutation.
  By definition of model-theoretic semantics, $M_P(\forall C)$ is the greatest lower bound of the values given to each ground instantiation of C by function M_P. Now, consider any of such ground instantiations, say $C\theta$. Since $C'\theta_i$ is equally or less instantiated than C, it is also equally or less instantiated than $C\theta$. Therefore $M_P(C\theta) = \sum_{i=1,\ldots,n} \mathbf{a_i} + \sum_{j=1,\ldots,m} \mathbf{b_j}$, where the semiring values $\mathbf{b_j}$ are present in other m facts whose head is less instantiated than $C\theta$ but more instantiated than C. Now we must compute the glb of all such $M_P(C\theta)$ for all θ. But it is possible to see that there is a θ such that $M_P(C\theta)$ involves only the n facts with semiring values $\mathbf{a_i}$: just choose a substitution, say θ', which binds all variables in C to domain elements not present in P. Therefore $M_P(C\theta') = \sum_{i=1,\ldots,n} \mathbf{a_i}$. This means that the glb of all $M_P(C\theta)$ is exactly $M_P(C\theta')$, that is, $\mathbf{a}$.
- **inductive case:** Let us assume that the statement of the theorem holds when the longest of the refutations of the form $\langle C, \varepsilon, \mathbf{a_i} \rangle$ has length n. Let us now consider a collection of atoms C such that the longest refutations

for C have length $n+1$. We have two cases to consider: one occurs when C contains just one atom, and the other one when C has two or more atoms.

- **1st case (one atom):** The reasoning we will use in this case is very similar to that of the base case. To consider the refutation $\langle A, \varepsilon, \mathbf{a}\rangle$, let us assume to start from the goal $\langle A, \varepsilon, \mathbf{1}\rangle$, where A is an atom. Consider now the set of n clauses contained in the program P of the form $A' :- \langle B_i, \theta_i, \mathbf{0}\rangle$, for $i = 1, \ldots, n$, where B_i is a collection of atoms and A' contains only variables and the same predicate symbol as A. To be used to expand our goal, it must be that $A'\theta_i$ is equally or less instantiated than A. In fact, if the head of such clauses were more instantiated than A, then we would have a substitution different from ε in the refutation. By definition of $OS1_P$, we have that $OS1_P(A) = \sum_{i=1,\ldots,n} OS1_P(B_i\theta_i)$. By definition of model-theoretic semantics, $M_P(\forall A)$ is the greatest lower bound of the values given to each ground instantiation of A by function M_P. Now, consider any of such ground instantiations, say $A\theta$. Since $A'\theta_i$ is equally or less instantiated than A, it is also equally or less instantiated than $A\theta$. Therefore $M_P(A\theta) = \sum_{i=1,\ldots,n} \mathbf{a_i} + \sum_{j=1,\ldots,m} \mathbf{b_j}$, where $\mathbf{a_i} = M_P(\forall B_i\theta_i)$ for $i = 1, \ldots, n$, and $\mathbf{b_j} = M_P(\forall B_j\theta_j)$ for $j = 1, \ldots, m$, with $A' :- \langle B_j, \theta_j, \mathbf{0}\rangle$, for $j = 1, \ldots, m$ other clauses where $A'\theta_j$ is more instantiated than A but less instantiated than $A\theta$.
  Now we must compute the glb of all such $M_P(A\theta)$ for all θ. But it is possible to see that there is a θ such that $M_P(A\theta)$ involves only the n clauses with semiring values $\mathbf{a_i}$: just choose a substitution, say θ', which binds all variables in A to domain elements not present in P. Therefore $M_P(A\theta') = \sum_{i=1,\ldots,n} \mathbf{a_i} = \mathbf{a}$. This means that the glb of all $M_P(A\theta)$ is exactly $M_P(A\theta')$, that is, $\sum_{i=1,\ldots,n} M_P(\forall B_i\theta_i)$. Now, by inductive hypothesis, this coincides with $\sum_{i=1,\ldots,n} OS1_P(B_i\theta_i)$, which is $OS1_P(A)$ as we stated before.
- **2nd case (two or more atoms):** Let us assume that we start from the goal $\langle C, \varepsilon, \mathbf{1}\rangle$, and that $C = A, C'$. Then we have that:
  $M_P(\forall C) = \{$by definition of C $\}$
  $M_P(\forall(A, C')) = \{$by definition of interpretation over a universally quantified formula$\}$
  $glb_\theta\{M_P((A, C')\theta)\} =$
  $glb_\theta\{M_P((A\theta, C'\theta)\} = \{$by definition of interpretation over a conjunction of formulas$\}$
  $glb_\theta\{M_P(A\theta) \times M_P(C'\theta)\} = \{$by inductive hypothesis and by the assumption that C' is not empty$\}$
  $glb_\theta\{OS1_P(A\theta) \times OS1_P(C'\theta)\} = \{$by 6.4.1$\}$
  $glb_\theta\{OS1_P(A\theta, C'\theta)\} =$
  $glb_\theta\{OS1_P((A, C')\theta)\} = \{$by 6.4.2$\}$
  $OS1_P(A, C') = \{$by definition of $C\}$
  $OS1_P(C)$.

6.4.2 Existential Closure

Theorem 6.4.3 relates the operational semantics of a goal C to the model-theoretic meaning of its universal quantification, that is, $\forall C$. We will now show that also the declarative meaning of the existential quantification of C, that is, $\exists C$, can be computed operationally. First we need to define a function $OS2_P(C)$ which combines all the refutations for C by summing all the corresponding semiring values.

Definition 6.4.4. (FUNCTION $OS2_P$) *Given an SCLP program P, function $OS2_P : LAT \to A$, where LAT is the set of conjunctions of atoms and A is the semiring set, is defined as follows: $OS2_P(C) = \sum_{\langle C,\theta,\mathbf{a}\rangle \in S(P)} \mathbf{a}$.*

An intuitive explanation of why $OS2_P$ is defined this way is the following: since we are interested in describing the operational meaning of $\exists C$, we do not care which substitution is built during a refutation starting from C. Therefore, we take all refutations starting from C. Then, we choose the best among all such refutations by using the $+$ operator.

It is worth noting at this point that this definition of $OS2_P$ coincides with the following one: $\sum_{\langle C\theta,\varepsilon,\mathbf{a}\rangle \in S(P)} \mathbf{a}$. First we need the following lemma.

Lemma 6.4.3. *Given an SCLP program P and a conjunction of atoms C, we have that, if $\langle C\theta, \varepsilon, \mathbf{a}\rangle$ is in $S(P)$, then there exists θ' such that $\langle C, \theta', \mathbf{a}\rangle$, with $\theta' \leq \theta$ is in $S(P)$ as well.*

Proof. If we start a refutation from C instead of $C\theta$, we can follow the same steps as in refutation $\langle C\theta, \varepsilon, \mathbf{a}\rangle$, because we have a more general goal. The only difference is that we may need to build a substitution, say θ', different from ε. Such a substitution, however, will be compatible with θ since we followed the same steps. Moreover, θ' cannot be more specific than θ, otherwise the refutation starting from $C\theta$ would have built a substitution different from ε.

Theorem 6.4.4. *Given an SCLP program P and a conjunction of atoms C, we have that $\sum_{\langle C,\theta,\mathbf{a}\rangle \in S(P)} \mathbf{a} = \sum_{\langle C\theta,\varepsilon,\mathbf{a}\rangle \in S(P)} \mathbf{a}$.*

Proof. We will prove this theorem by showing that the set S_1 of semiring values reached by refutations of the form $\langle C, \theta, \mathbf{a}\rangle$ coincides with the set S_2 of values reached by refutations of the form $\langle C\theta, \varepsilon, \mathbf{a}\rangle$. In fact, if we show this, then, since $+$ is idempotent, the two sums coincide as well.

One direction of the proof has already been proven in Theorem 6.4.1: if $\langle C, \theta, \mathbf{a}\rangle$ is a refutation, then $\langle C\theta, \varepsilon, \mathbf{a}\rangle$ is a refutation as well. Therefore we have that $S_1 \subseteq S_2$. Now we have to prove that $S_2 \subseteq S_1$. In general, it is not true that, if $\langle C\theta, \varepsilon, \mathbf{a}\rangle$ is a refutation, then $\langle C, \theta, \mathbf{a}\rangle$ is a refutation as well. By Lemma 6.4.3, however, we have that, if $\langle C\theta, \varepsilon, \mathbf{a}\rangle$ is a refutation, then there exists θ' such that $\langle C, \theta', \mathbf{a}\rangle$, with $\theta' \leq \theta$, is a refutation as well. This is enough to prove that $S_2 \subseteq S_1$, since we are proving the equality between sets of semiring values and not sets of refutations.

Another alternative way to define $OS2_P$ is by using the definition of $OS1_P$, as follows.

Theorem 6.4.5. *Given an SCLP program P and a conjunction of atoms C, we have that* $OS2_P(C) = \sum_{C\theta \text{ ground}} OS1_P(C\theta)$.

Proof. The statement of the theorem comes from the definition of $OS1_P$, the associativity and idempotency of $+$, and Theorem 6.4.2.

Now we formally show that the result of the application of function $OS2_P$ over any goal C is a semiring value that coincides with the model-theoretic meaning of $\exists C$.

Theorem 6.4.6. (OPERATIONAL MEANING OF $\exists C$) *Given an SCLP program P, consider a collection of atoms C. Then we have that* $M_P(\exists C) = OS2_P(C)$.

Proof. By definition of interpretation, $M_P(\exists C) = lub\{M_P(C\theta)$ for all θ s.t. $C\theta$ is ground$\} = \sum_{C\theta \text{ ground}} M_P(C\theta)$. Since $C\theta$ is ground, Theorem 6.4.3 says that $M_P(C\theta) = OS1_P(C\theta)$. Therefore, we have that $M_P(\exists C) = \sum_{C\theta \text{ ground}} OS1_P(C\theta)$, which is exactly $OS2_P(C)$ by the Theorem 6.4.5.

6.5 A Semi-decidability Result

In logic programming, looking for an answer of a given goal is a semi-decidable problem: if the goal is satisfiable, then a refutation for such a goal can be found in finite time; but if the goal is not satisfiable, then there are cases in which we can go on forever without detecting such an unsatisfiability.

We will now show that a similar semi-decidability result holds also for SCLP programs. More precisely, if the semantics of an SCLP goal is a semiring value greater than (or greater than or equal to) a certain semiring value **k**, then we can discover this in finite time.

The main idea, as in the logic programming case, is to visit the derivation tree of the given goal in a breadth-first way. We recall that the derivation tree of a goal is the tree whose root is the given goal, each node is a state in a derivation, and each path from the root to a leaf represents a derivation for that goal. If we visit this tree in a breadth-first way, it means that after k steps (that is, after examining k levels) we have seen all refutations of length k or less, and the first k steps of all derivations/refutations longer than k.

During this visit, we construct a sum of semiring values: starting from the **0** of the semiring at the root level, when we are at level k we add to the current sum the sum of all the semiring values associated to all refutations of length k. From the fact that the sum always leads to better values, as we go on we get new values that are better than or equal to the old ones.

Now, the crucial point is that, given any semiring value representing a partial sum, even though the values better than it are infinite (because a semiring can be infinite), the number of times that this value can be improved is finite. Thus, after examining at most a finite number of refutations, we will have computed the semiring value of the goal.

To formally prove this result, we will basically define an ordering among partial sums, and we will show that such an ordering is well-founded. Any partial sum will be represented by a suitable set of semiring values, obtained from the refutations examined so far.

We first need to give some formal definitions.

Definition 6.5.1. (C-PRODUCTS) *Let $C = \{c_i\}_{i=1,\dots,k}$ be a finite set of elements of a semiring S. A (symbolic) product $p = \prod_{i=1,\dots,k} c_i^{n_i}$ is called a C-product and its value in S is denoted by $[\![p]\!]$.*

Informally, a C-product represents the semiring value obtained by a refutation. In fact, such a value is obtained by multiplying all the semiring values of the various clauses used by the refutation. Such values belong to a specific set: the set of all semiring values that appear in the given program, which we call C here. The exponents in the product are needed because each semiring value in C may be used several times during a refutation, and all these occurrences have to be considered, since in general the multiplicative operation is not idempotent.

Definition 6.5.2. (PARTIAL ORDER OF C-PRODUCTS) *We define a partial ordering $\sqsubseteq$ on C-products as $\prod_{i=1,\dots,k} c_i^{n_i} \sqsubseteq \prod_{i=1,\dots,k} c_i^{n'_i}$ iff $n_i \geq n'_i$ for all $i = 1, \dots, k$.*

Given two C-products p and p', if $p \sqsubseteq p'$ it means that all the semiring values of C have more (or the same number of) occurrences in p than in p'. Thus, the value of p is worse than that of p' in the semiring: multiplying more items leads to worse results, by the intensivity of $\times$. Formally: $p \sqsubseteq p'$ implies $[\![p]\!] \leq_S [\![p']\!]$. Thus, we can say that p is "dominated" by p'.

Definition 6.5.3. (SATURATION OF A SET OF C-PRODUCTS) *Given a finite or countably infinite set P of C-products, its saturation $\overline{P}$ is defined as $\overline{P} = \{p \mid \exists p' \in P. p \sqsubseteq p'\}$.*

By saturating a set of C-products P, we basically add to P all those other C-products which are dominated by some element in P. Notice that $[\![P]\!] = [\![\overline{P}]\!]$, where we extended the use of the semantic parenthesis $[\![]\!]$ from C-products to sets of C-products: $[\![Q]\!] = \sum_{p \in Q} [\![p]\!]$.

In our method to compute the semantics of a goal, every time we add an element to the current partial sum, such an element is a C-product. If the semiring value of this element is dominated by another one already in the partial sum, in reality the sum value does not change, because of the properties of the $+$ operation: if $a \leq_S b$ then $a + b = b$.

The main result of this section is that such a chain of partial sums has a finite number of distinct elements, and that, thus, after such a finite number of steps we have computed the semantics of a goal in an SCLP program. But before stating this results we need a lemma.

Lemma 6.5.1. (WELL-FOUNDEDNESS) *Let $\mathcal{P}_C$ be the set of all saturated (i.e., $\overline{P} = P$) sets of C-products. This set is well-founded under the inverse proper inclusion relation $\supset$, i.e. all the chains $P_0 \subset P_1 \subset \dots$ of elements of $\mathcal{P}_C$ are of finite length.*

Proof. We first consider only chains where $P_j = \overline{R_j}$, with R_j finite. Also, without loss of generality, we assume $P_0 = R_0 = \emptyset$ and $R_j = R_{j-1} \cup \{p_j\}$, with of course $\forall p \in R_{j-1}$, $p_j \not\sqsubseteq p$.

We prove the property by mathematical induction on the number k of constants in C. If $k = 1$ the property is trivial, since $P_j = \{c^n \mid n \geq \overline{n_j}\}$ for some $\overline{n_j}$, and thus $P_j \subset P_{j+1}$ means $\overline{n_j} > \overline{n_{j+1}}$.

Let us now assume that the property holds for $k-1$ and prove it for k. We work by absurd, assuming that an infinite chain exists for k and constructing an infinite chain also for $k-1$. Let us now decompose p_j as $p_j = p'_j \times c_k^{n_k^j}$, with $p'_j = \prod_{i=1,\dots,k-1} c_i^{n_i^j}$. Now, since we must have $p_h \not\sqsubseteq p_j$ when $j < h$, we either have $p'_h \not\sqsubseteq p'_j$ or $n_k^j > n_k^h$. We now use a colored graph method to help us in the proof. The nodes of the graph are the indices of the chain, and we draw a red arc from i to h in the latter case (that is, $n_k^j > n_k^h$) or when both conditions hold, a black arc otherwise. We now construct an infinite subsequence with no red arcs, i.e. where $p'_h \not\sqsubseteq p'_j$ for $h > j$.

Notice first that there is no infinite red path: in fact our graph is acyclic and exponent n_k^j cannot be indefinitely decreased. Now assume that we have already examined a finite initial segment of the chain, and we have already constructed a set of indexes I, initially empty. We assume inductively that there is no red arc outgoing from the indexes in I to the indexes of the rest of the chain. To find a new index to add to I, let us consider the first index of the rest of the chain. If it has no outgoing red arc, we are done. Otherwise, we follow any outgoing red arc, and we repeat the above procedure until an index without outgoing red arcs is found. Since there is no infinite red path, the procedure must terminate.

Finally, it is easy to see that for every P in $\mathcal{P}_C$ we have $P = \overline{R}$, for some finite R. In fact, let $R = \{p \in P \mid \forall p' \in P.p \not\sqsubseteq p'\}$. If R were infinite (but of course countable: $R = \{p_j\}_{j=1,2,\dots}$), the chain $\emptyset \subset \overline{\{p_1\}} \subset \overline{\{p_1, p_2\}} \subset \dots \subset \overline{\{p_j \mid j \leq h\}} \subset \dots$ would be infinite.

Theorem 6.5.1. (FINITE CHAINS) *Let $P = \{p_j\}_{j=1,2,\dots}$ be a finite or countably infinite set of C-products. Then there is a natural number N such that $\sum_{j=1,\dots,N} [\![p_j]\!] = \sum_{j=1,2,\dots} [\![p_j]\!]$.*

Proof. The statement is trivial if P is finite. Thus, let us assume that P is infinite. According to Lemma 6.5.1, the chain $Q_1 \subseteq Q_2 \subseteq \dots \subseteq Q_n \subseteq \dots$ with $Q_h = \overline{\{p_j \mid j \leq h\}}$, is finite, i.e. there is a natural number N such that $Q_r = Q_N$ for $r \geq N$. Since $[\![Q_h]\!] = \sum_{j=1,\dots,h} [\![p_j]\!]$ by definition, this proves the theorem.

Summarizing, the formal developments of this section show that, by examining the refutations of a given *SCLP* goal while visiting the search tree in a breadth-first way, the semantics of the goal can be computed in a finite number of steps (that is, after examining a finite number of refutations). This result, together with the property that $+$ is the lub of the lattice, leads us to the following semi-decidability statement:

> It is semi-decidable to decide whether the semantics of a goal is in relation R with a certain semiring value $\mathbf{k}$, where $R \in \{>, \geq\}$.

In fact, if it is in such a relation, then the theorem of this section tells us that after a finite number of steps we have computed such semantics. Thus, after a smaller or equal number of steps the current partial sum will have value $\mathbf{k}$ or more, at which point we can stop and say that the goal has a semantics in relation R with $\mathbf{k}$. If instead it is not in such a relation, then we do not have a method to know this in finite time. If there were, however, a method to semi-decide whether the semantics of a goal is in relation $not(R)$ with a certain semiring value $\mathbf{k}$, where $not(R) \in \{<, =, \leq, \not\equiv\}$, with $\not\equiv$ meaning "incompatible with", then by Post's theorem (if a property and its complement are semi-decidable, then the property is decidable) we would conclude that it is decidable to know whether the semantics of a goal is equal to a certain semiring value $\mathbf{k}$, which we know is not true for the special case of logic programming.

6.6 SCLPs with no Functions

In the previous section we have proven that, if a goal of an SCLP program has a semiring value greater than, or greater than or equal to, some $\mathbf{k}$, then it is possible to discover this in finite time. In this section we will show that, for the special class of SCLP programs with no functions, we have the additional result that once the program is fixed, the time for computing the value of any goal for this program is finite and bounded by a constant (see Theorem 6.6.2 later in this section). Thus, the semantics of SCLP programs without functions is decidable.

This result is based on the observation that, in SCLP programs without functions, we just have to consider a finite subclass of refutations, called in the following *simple* refutations, with a bounded length. After having considered all these refutations up to that bounded length, we have finished computing the semiring value of the given goal.

Notice that, while the absence of functions in SCLP programs is obviously a restriction, the underlying semiring could, in general, contain an infinite number of elements. Thus, this result is not so obvious as it may appear, since not all source of infiniteness are taken away.

First we define an alternative representation for refutations, which is based on trees.

Definition 6.6.1. (REFUTATION TREE) *Given a refutation r as follows:*

$$\langle A, \varepsilon, \mathbf{1}\rangle \overset{Cl_1,\theta'_1}{\rightarrow} \langle C_1, \theta_1, \mathbf{a_1}\rangle \overset{Cl_2,\theta'_2}{\rightarrow} \dots \overset{Cl_n,\theta'_n}{\rightarrow} \langle \Box, \theta_n, \mathbf{a_n}\rangle$$

its refutation tree *is a labeled tree where each node is labeled by a clause instantiation of the form $\langle Cl_i, \sigma_i\rangle$, for $i \in \{1, \dots, n\}$, where $\sigma_i = \theta'_i$ restricted* *onto the variables of the head of Cl_i. More precisely, starting from a single non-labeled node, the entire tree can be built as follows:*

```
for k:=1 to n do
    Select the first non-labeled node (which is a leaf) in a depth-first visit of
    the current tree, and label it with $\langle Cl_k, \sigma_k \rangle$.
    If $Cl_k$ is $H : -B_1, \ldots, B_m$, attach to this node m children.
end-for.
```

Definition 6.6.2. (SIMPLE REFUTATION) *Given a refutation tree, a path from the root to a leaf is called* simple *if all its nodes have different labels up to variable renaming. A refutation is a simple refutation if all paths from the root to a leaf in its refutation tree are simple.*

Thus, in a simple refutation it is not possible to use the same clause, instantiated in the same way, more than once on atoms that depend on each other.

We will now show that, if we delete all non-simple refutations from $S(P)$, we do not change the value computed by $OS1_P(C)$ for any C.

Theorem 6.6.1. (SIMPLE REFUTATIONS ONLY) *Consider an SCLP program P with no functions and its compact refutation set $S(P)$, and let us call $S'(P)$ the subset of $S(P)$ containing all its simple refutations. Then, for any goal C, we have that $\sum_{\langle C,\varepsilon,\mathbf{a}\rangle \in S'(P)} \mathbf{a} = OS1_P(C)$.*

Proof. Given any non-simple refutation r_1 for a goal, we can obtain a simple refutation r_2 for the same goal: just take the refutation tree of the non-simple refutation and delete the part of the tree between any two nodes with the same label up to variable renaming. It is easy to see that this new tree still represents a refutation, and that such a refutation is simple.

Now we can notice that refutation r_2 has fewer steps than r_1, and all the steps in r_2 are present in r_1. Therefore, by the extensivity of $\times$, the semiring value computed by r_2 is better than or equal to the one computed by r_1.

Now, $OS1_P(C)$ sums all the semiring values computed by all refutations in $S(P)$ that start from C and build ε. Therefore, considering what we have said before, the sum of the semiring values associated with r_1 and r_2 gives the semiring value of r_2, that is, of the simple refutation. Thus, by "forgetting" all non-simple refutations we do not change the result of the sum of all the semiring values, which is exactly $OS1_P(C)$.

We will now prove that, given a goal C, there is only a finite number of simple refutations starting from C and building the empty substitution.

Theorem 6.6.2. (FINITE SET OF SIMPLE REFUTATIONS) *Given an SCLP program P with no functions and a goal C, consider the set $SR(C)$ of simple refutations starting from C and building the empty substitution. Then $SR(C)$ is finite. Moreover, each refutation in $SR(C)$ has length at most N, where $N = \sum_{i=1}^{c} b^i$, c is the number of all clauses with head instantiated in all possible ways, and b is the greatest number of atoms in the body of a clause in program P.*

Proof. Since a program contains a finite number of domain elements and of clauses, the number of different labels in a path of a simple refutation tree is finite. Thus the number of different refutation trees for simple refutations of C is finite. Therefore $SR(C)$ is finite.

Moreover, each simple refutation tree has the property that no two labels in a path from the root to a leaf are the same. Since such labels are clauses plus head instantiations, each path is as long as at most the number of all possible clause head instantiations, say c. Furthermore, the branching factor of a simple refutation tree depends on the number of atoms in the bodies of the clauses used in the refutation. Thus, the number of nodes of a simple refutation tree, and thus of steps in a simple refutation, is bounded by $\sum_{i=1}^{c} b^i$.

Therefore, computing $OS1_P(C)$ involves looking at a finite number of bounded-length simple refutations. Thus, $OS1_P(C)$ can be computed in a finite number of steps.

Corollary 6.6.1. (FINITE NUMBER OF STEPS FOR $OS1_P$) *Given an SCLP program P with no functions, consider a collection of atoms C. Then we have that $OS1_P(C)$ can be computed in a finite number of steps.*

Proof. The statement follows directly from the results of Theorem 6.6.1 (we can forget about the non-simple refutations) and Theorem 6.6.2 (the number of simple refutations is finite, and the length of each simple refutation is bounded by a constant N). In fact, we can consider only those refutations with length at most N, and among these we take only the simple refutations and we sum their semiring values.

It is also easy to prove that $OS2_P(C)$ can be computed in a finite number of steps as well. Moreover, the T_P operator need to be applied only a finite number of times before reaching the fixpoint and thus computing the fixpoint semantics, as defined in Section 6.3.

6.7 An Operational Model for the SCLP Language Using ASM

In this section we develop a simple interpreter for SCLP programs without functions using the *Gurevich's Abstract State Machines* (ASMs) (previously called *evolving algebras* [118]). This formalism has been applied successfully to specify real programming languages and architectures, to validate standard language implementations, to verify real-time protocols, etc. (see annotated ASM bibliography in [59]).

We specify an operational semantic model that directly reflects the intuitive procedural understanding of SCLP programs, but is formulated at the level of abstract search spaces. This combination of procedural and abstract features, provides a tool for mathematical description and analysis of the design decisions for the language, in a machine and proof system independent manner. Moreover, it lays the ground for provably correct stepwise refinement, through a hierarchy of specifications at lower level, down to implementations. Therefore, the ASM model we propose can be viewed as the first step versus a real implementation of the language: according to the result presented in [15], that can be reached through a sequence of (correctly) refined ASMs down to a model that can be mechanically translated into an executable code.

The model also suggests a possible strategy to optimize the parallel computation of all possible solutions, formalizing useful techniques that allow to avoid infinite and/or non optimal computations.

In Subsection 6.7.1 we briefly introduce the reader to the Gurevich's Abstract State Machine and then in Subsection 6.7.2 the operational semantics of the language in terms of ASMs is given.

6.7.1 The Gurevich's Abstract State Machine

We assume that the reader is familiar with the semantics of the Abstract State Machine defined in [118], and we quote here only the essential definitions.

Given a program *Prog* (consisting of a finite number of *transition rules*) and a (class of) *initial state*(s) S_0, a *Gurevich's Abstract State Machine* $\mathcal{A}$ models the operational behavior of a real dynamic system $\mathcal{S}$ in terms of state transitions. A *state* is a first-order structure representing the instantaneous configuration of $\mathcal{S}$.

The basic form of a transition rule is the following *function update*

$$f(t_1, \ldots, t_n) := t$$

where f is an arbitrary n-ary function and $t_1, \ldots, t_n, t$ are first-order terms. To fire this rule to a state S_i, $i \geq 0$, evaluate all terms $t_1, \ldots, t_n, t$ at S_i and update the function f to t on parameters $t_1, \ldots, t_n$. This produces another state S_{i+1} which differs from S_i only in the new interpretation of the function f.

There are some rule constructors.

– The *conditional constructor* which produces "guarded" transition rules of the form:

  $$\texttt{if } g \texttt{ then } R_1 \texttt{ else } R_2$$

  where g is a ground term (the *guard* of the rule) and R_1, R_2 are transition rules. To fire that new rule to a state S_i, $i \geq 0$, evaluate the guard; if it is true, then execute R_1, otherwise execute R_2. The `else` part may be omitted.
– The *parallel constructor* which produces the "parallel" synchronous rule of form:

  $$\texttt{var } x \texttt{ ranges over } U$$
  $$R(x)$$

  where $R(x)$ is a generalized basic transition rule with a variable x ranging over the universe U.
  The operational meaning of the constructor consists in the execution of the new rule $R(x)$ for every $x \in U$.

We also make use of the following construct to let universes grow:

`extend` U `by` $x_1, \ldots, x_n$ `with` *Updates* `endextend`

where *Updates* may (and should) depend on the x_i and are used to define certain functions for (some of) the new objects x_i of the resulting universe U.

State transitions of $\mathcal{A}$ may be influenced in two ways: *internally*, through the rules of the program *Prog*, or *externally* through the modifications of the environment. A non static function f that is not updated in any transition rule is called *oracle function* if its values are given non-deterministically by an oracle.

A computation of $\mathcal{S}$ is modeled through a finite or infinite sequence $S_0, S_1, \ldots, S_n, \ldots$ of states of $\mathcal{A}$, where S_0 is an initial state and each S_{n+1} is obtained from S_n by firing simultaneously all of the rules of *Prog* to S_n.

6.7.2 The Abstract Operational Model of SCLP

Signature. This section explains the basic data types (domains and functions) which are used in the transition rules of Section 6.7.2.

A SCLP computation can be seen as a parallel search for all solutions (final goals) of an initially given query, to be found in a space structured as a tree. Each query's computation is represented by a branch of the tree and the breadth search strategy is similar to the OR-Parallel model for Prolog.

The operational result of a computation is the (semiring) sum of the values of all computed solutions; since some branches might lead to infinite computations as well as to useless solutions (final goal whose semiring value a is worse than the sum of the values associated with the already computed solutions), we introduce some optimization techniques which allow to cut away all 'bad' branches, thus avoiding useless computations.

We represent the search space by a set *NODE* of nodes and the tree structure on *NODE* is formalized by a partial function $parent : NODE \to NODE$ (undefined on *root*), to be dynamically updated through rules.

All possible SCLP *computation states* intuitively correspond to the levels of the tree, therefore they are represented by the subset of *NODE* made up of all *active* nodes (see below) at that state.

Each element n of *NODE* has to carry the relevant information for a complete description – at this abstraction level – of the (sub)computation in the subtree rooted at n. This information consists of the current goal $G = \langle C, \theta, a \rangle$, where C is the collection of atoms which still have to be computed, θ is the substitution computed so far and a is the value of the semiring representing the *cost* (or level of preference, certainty, probability, etc.) accumulated performing the previous (part of the) computation.

Hence, we introduce the function

$$goal : NODE \to GOAL$$

associating with each node the goal which has still to be computed.

GOAL is the set of SCLP goals, and it yields $GOAL = AT^* \times SUB \times A$, where AT denotes the universe of atoms, SUB denotes the set of substitutions, A is the set of the semiring $S = < A, +, \times, \mathbf{0}, \mathbf{1} >$.

GOAL comes with the following functions yielding the three components of a goal:

$$listatom : GOAL \rightarrow AT^*$$
$$sub : GOAL \rightarrow SUB$$
$$cost : GOAL \rightarrow A$$

SUB is a set of (not further specified) *substitutions* and comes together with three abstract functions

$$mgu : TERM \times TERM \rightarrow SUB \cup \{undef\}$$

associating with two terms either their most general unifier substitution or the answer that there is not any; the substitution applying function

$$apply : TERM \times SUB \rightarrow TERM$$

yielding the result of applying the given substitution to a given term; substitution concatenation

$$\circ : SUB \times SUB \rightarrow SUB$$

TERM is the set of generic SCLP terms (atoms or collections of atoms).

CLAUSE denotes the set of program statements and comes with auxiliary functions yielding *head* and *body* of SCLP clauses

$$clhead : CLAUSE \rightarrow AT$$
$$clbody : CLAUSE \rightarrow GOAL$$

The current program is represented by a distinguished element *db* (database) of a universe *PROGRAM*. The candidate clauses, which have to be considered in a call to define the alternatives of a goal computation, are accessed using an abstract function

$$procdef : AT \times PROGRAM \times \mathrm{N} \rightarrow CLAUSE^*$$

which yields the (renamed) clauses defining, in the given program, the predicate having the same functor as the selected atom. $\mathcal{N}$ represents the set of natural numbers and the following function yields the current renaming index

$$lev : NODE \rightarrow \mathrm{N}$$

This function is recursively defined as follows:

$$lev(n) = \begin{cases} 0 & \text{if } n = root \\ 1 + lev(parent(n)) & \text{otherwise} \end{cases}$$

We also allow the use of the functions *head* and *tail* on lists with obvious meaning, and the function $proj(i, L)$ to get the i-th element of a list L.

We use the abbreviation

$$act(n) \equiv head(listatom(goal(n)))$$

to select the current atom to compute at the level of the node n.

In order to control the computation of useless solutions, we define the subset $SUCCLEAF = \{n \in NODE \mid goal(n) = \langle \Box, \theta, a \rangle, \theta \in SUB, a \in A\}$ of *NODE* containing all the leaf nodes of the search tree whose paths represent successful computations already performed. We also introduce the global variable

$$estimate = \begin{cases} \sum_{n \in SUCCLEAF} cost(goal(n)) & \text{if } SUCCLEAF \neq \emptyset \\ \mathbf{0} & \text{otherwise} \end{cases}$$

which yields, at each state, the *best* semiring value among those associated to the already computed solutions. If the value a of the current goal $\langle C, \theta, a\rangle$ is worse than *estimate*, this goal computation is not to be carried on. We also make use of the predicate

$$\mathit{admissible} : A \longrightarrow \{true, false\}$$

such that

$$\mathit{admissible}(c) = \begin{cases} true & \text{if } \mathit{estimate} \leq_S c \\ false & \text{otherwise} \end{cases}$$

where $\leq_S$ is the partial order relation on the semiring S.

Since the SCLP terms do not contain any function symbol and the initial queries consist of only ground atoms, all infinite computations can be found out by looking for cycles; a cycle occurs along a branch when there are two nodes containing the same goal (without considering variable renaming). This strategy allow us to stop all infinite computations.

Therefore, we introduce the function

$$\mathit{double} : \mathit{NODE} \longrightarrow \{true, false\}$$

which is true on a given n if there is an ancestor node of n having the same sequence of atoms to compute. Formally speaking,

$$\mathit{double}(n) = \begin{cases} true & \text{if } \exists n' \in \mathit{ancestor}(n) : \mathit{comp}(n') \doteq [\mathit{comp}(n)]\theta, \\ & \text{for some renamed substitution} \theta \\ false & \text{otherwise} \end{cases}$$

where $\mathit{ancestor}(n)$ yields the set of all nodes along the path from n to the *root*, $\doteq$ stands for the syntactical equality, and $\mathit{comp}(n) \equiv \mathit{apply}(\mathit{listatom}(\mathit{goal}(n)), \mathit{sub}(\mathit{goal}(n)))$.

To be able to speak about *termination* we will distinguish between *active* and *non-active* nodes by the function

$$\mathit{active} : \mathit{NODE} \longrightarrow \{true, false\}$$

A node is born active and becomes non-active either after performing its own computation step or if one of the following three cases occurs:

– computation failure: when the first atom of the current goal does not unify with the head of the selected clause;
– the current goal will be a useless solution: when its semiring value is worse than *estimate*;
– infinite computation: when the current node already occurs along its path.

To keep information of all computed solutions of the initial query, we define the subset $\mathit{SOL} = \{g \in \mathit{GOAL} \mid g = \langle \Box, \theta, a\rangle\}$ of GOAL.

Transition Rules. We now define the rules by which the system, starting from an *initial state*, tries to reach successful execution of the query `Q` ($\neq \Box$) by a given program `P`. The initial state satisfies the conditions: $\mathit{NODE} = \{\mathit{root}\}$; $\mathit{goal}(\mathit{root}) = \langle \mathtt{Q}, \varepsilon, \mathbf{1}\rangle$; $\mathit{active}(\mathit{root}) = true$; $\mathit{SUCCLEAF} = \emptyset$; $\mathit{SOL} = \emptyset$.

According to the operational semantic rule of a SCLP programs, the basic computation step consists of splitting the current goal $\langle (A, Cr), \theta, a\rangle$ associated with a node n, into m goals of the form $\langle (C_i, Cr), \theta \circ \theta'_i \circ \theta_i, a \times a_i \rangle$, where m is the number of candidate clauses for A, $A_i : -\langle C_i, \theta_i, a_i \rangle$, $1 \leq i \leq m$, is the i-th candidate clause for A and $\theta'_i = mgu(A\theta, A_i\theta_i)$. All the new m goals are computed in parallel. We can imagine the tree representing the computational space grows in breadth at each computation step performed by all the active nodes.

If an active node n has an admissible semiring value (i.e. $admissible(cost(goal(n)) \neq false)$, and does not belong to an infinite branch (i.e. $double(n) \neq true$) and its computation has not failed (i.e. $sub(goal(n)) \neq undef$), by the following **Reduction Rule** as many child nodes of n are created as there are the candidate clauses for the activator $act(n)$ of n; and n becomes non-active. Otherwise the active node becomes non-active, as a leaf of the tree, without performing any splitting and stopping the computation along its path. In such a way all the "bad" branches are pruned. This strategy allows one to avoid the computation of non optimal solutions and infinite computations, thus optimizing the computation of the initial query.

Each child node is created *active* and receives as goal the result of the resolution between its parent goal and the relative selected candidate clause. This resolution step is performed by the macro *set_goal*(t_i, n) according to the semantic rule of the SCLP language.

Reduction Rule

```
var $n$ ranges over $NODE$
    If $active(n)$
    then if $sub(goal(n)) \neq undef$
             $\wedge\ admissible(cost(goal(n)))$
             $\wedge\ \neg double(n)$
         then let $l = length(procdef(act(n),\ db,\ lev(n)))$
              extend $NODE$ by $t_1, t_2, \ldots, t_l$ with
                     $parent(t_i) := n$
                     $active(t_i) := true$
                     $set\_goal(t_i, n)$
              endextend
         $active(n) := false$
```

```
$set\_goal(t_i, n) \equiv$ let $clause = proj(i,\ procdef(act(n),\ db,\ lev(n)))$
                        $goal(t_i) := \langle\ [listatom(clbody(clause)) \mid tail(listatom(goal(n))],$
                                     $mgu(apply(act(n), sub(goal(n))),$
                                          $apply(clhead(clause), sub(clbody(clause)))),$
                                     $cost(goal(n)) \times cost(clbody(clause))\rangle$
```

If the tree cannot grow any more, i.e. if all the nodes are not active, by the following **Collect Rule** all the computed solutions of the initial query are collected into the set *SOL*.

Collect Rule
```
If $\forall n \in \mathit{NODE}$: $\neg \mathit{active}(n)$
then
    $\mathit{SOL} := \bigcup_{n \in \mathit{SUCCLEAF}} \mathit{goal}(n)$
```

According to the OS function definition, we can compute the operational meaning of the program P with initial query Q as $OS_{\mathtt{P}}(\mathtt{Q}) = \sum_{g \in SOL} cost(g)$.

6.8 Related Work

In [98] a bilattice structure is used to model the presence of a family of truth values in logic programming, and a fixpoint semantics for this kind of programs is given. In this approach, the *meet* and *join* operators of the bilattice are used as extensions of classical *and* and *or*, while in our approach we use such operators to model the universal and existential quantification, but we adopt a different operator (the multiplicative operation of the semiring, possibly different from the *meet* operator), to extend the *logical and*. The bilattice structure he uses, allows one to always have a negation operator, which in general we do not have. By having less properties for our structure, we can model also situations where the multiplicative operator is not idempotent, like optimization and probabilistic problems.

The approach taken in [95] associates a value to each clause. From such values, taken from $[0, 1]$, a value is also associated to each atom. Then, atoms are combined by using the min and max operators. Thus, this kind of logic programming is similar to what we have when using the fuzzy semiring. He also gives a model-theoretic semantics, a fixpoint semantics, and an operational semantics based on game theory.

A recent approach to multi-valued logic programming [148] uses bilattices with two orderings to model both truth and knowledge levels. The resulting logic programming semantics is just operational and fix-point, while no model-theoretic semantics is presented. Moreover, the presence in our approach of just one ordering (modeling truth levels) is not a restriction, since the vectorization of several semirings is still a semiring (see Section 2.3.8) and thus optimization based on multiple criteria can be cast in our framework as well.

From another point of view, where classical constraints are extended to have several degrees of satisfaction, a related approach is HCLP (Hierarchical CLP) [62], where each constraint has a level of importance (like *strong, weak, required*), and these levels are used to decide which constraints to satisfy. A constraint, however, can only be satisfied or not, and thus HCLP is a *crisp* formalism. Moreover, their treatment is only algorithmic, and they do not provide their language with a fix-point or a model-theoretic semantics.

The `CLP(FD,S)` System. Strictly related to the language described in this chapter is the `CLP(FD,S)` implementation. The `CLP(FD,S)` language was developed on top of the existing constraint language `CLP(FD)` [72] by the LOCO research group at INRIA-Roquencourt. At the URL `http://pauillac.inria.fr/~georget/clp_fds.html` there is the latest version of this system, and in [111, 112] there is a description of their implementation.

In this implementation, full semiring-based arc-consistency (as described in Section 3.1) is used to keep the computation states compact and to check them for inconsistencies. In general, each domain element, as well as each constraint tuple, can be assigned a semiring value. They also provide, however, useful built-ins that assign just two values to each arithmetic constraint: one to the tuples which satisfy it, and the other one to those which do not satisfy it. In this way, the modeling is easier and the computation faster, and in some cases it is expressive enough.

Other plug-ins that we can mention are: arithmetic constraints, boolean constraints, meta constraints, optimization constraints, global constraints, predicates for constraints retraction and some facilities to build new constraints.

The kernel of `clp(FD,S)`, is called *SFD* and is generic with respect to the semiring. Hence, the users are able to generate new languages (new solvers) by specifying semirings, the rest of the implementation being unchanged. For example, `clp(FD,S)` allows the computation of satisfaction and optimization of: usual CSPs (clp(FD,Bool)), Fuzzy CSPs, hierarchical CSPs (clp(FD,Fuzzy)), hypothetical CSPs (clp(FD,Sets)).

The language has been ported over Solaris, SunOS, GNU-Linux (i86) and requires GNU C (gcc) version 2.4.5 or higher. Some performance statistics may be found in [111].

6.9 Conclusions

In this chapter a programming framework to deal with soft constraints has been introduced. We have described its syntax and we give several (equivalent) semantics: the operational, the model-theoretic and the fix-point one. Moreover, a generalization of the semidecidability result already known for logic programs has been described and proven: we can decide if the semantics of an SCLP goal has semiring value greater than k.

In this chapter a detailed semantics of the language has been given but no real applications to specific problems are given. We will do this in the next chapter where we will use this language to describe and solve shortest path problems.

7. SCLP and Generalized Shortest Path Problems

Overview

In recent years, an increasing number of researchers, from both the Operations Research (OR) and the Artificial Intelligence (AI) communities, have investigated the possibility of integrating methodologies of the two fields in order to obtain better results in solving combinatorial optimization problems. In this chapter we show how the SCLP paradigm could be useful to merge together AI and OR interests.

In fact, in this context constraint programming may be imposed as a suitable environment for performing such an integration, and the level of preference introduced can be used to discriminate between several solutions. In particular, we study the relationship between Constraint Programming (CP) and Shortest Path (SP) problems. More specifically, we show that classical, multicriteria, partially ordered, and modality-based SP problems can be naturally modeled and solved within the Soft Constraint Logic Programming (SCLP) framework, where logic programming is coupled with soft constraints. In this way we provide this large class of SP problems with a high-level and declarative linguistic support whose semantics takes care of both finding the cost of the shortest path(s) and also of actually finding the path(s). On the other hand, some efficient algorithms for certain classes of SP problems can be exploited to provide some classes of SCLP programs with an efficient way to compute their semantics.

Shortest Path (SP) problems [75, 90] are among the most studied network optimization problems. They are mainly used to represent and solve transportation problems, where the optimization may involve different criteria, for example cost, time, resources, and so on. Most interesting is the multi-criteria case, where the optimization involves a set of criteria to be all optimized [158].

In this chapter, we propose the Soft Constraint Logic Programming (SCLP) framework [46,112] as a linguistic support and a high-level and flexible programming environment in which to model SP problems naturally and solve them efficiently.

Here we consider several versions of SP problems, from the classical one to the multi-criteria case, from partially ordered SP problems to those that are based on modalities, and we show how to model and solve them via SCLP programs. The basic idea is that soft constraints allow to faithfully represent the optimization criteria, and CLP provides a declarative way to describe the given SP problem. Moreover, this way of modeling and solving SP problems allows to associate with such problems both a declarative and an operational semantics.

The main results of the chapter are as follows:

- Both classical, multi-criteria, partially-ordered, and modality-based SP problems are given a modelization as SCLP programs; such programs are able to find both the cost of the shortest path(s) and also the shortest path itself.
- A general methodology is provided to find a non-dominated path for both multi-criteria and partially-ordered SP problems; this methodology is based on a change of semiring, but does not require any change in the underlying SCLP syntax and semantics.
- A new algorithm to obtain the semantics for a particular class of SCLP programs is given, which is obtained by using a generalized version of the Floyd-Warshall algorithm [99] for SP problems.

The chapter (based on the ideas developed in [48,51]) is organized as follows: Section 7.1 shows the construction to pass from a classical SP problem to a CLP program, while Section 7.2 considers multi-criteria and partially-ordered SP problems, and Section 7.3 deals with modality-based SP problems. Finally, Section 7.4 concludes the chapter by providing a class of SCLP programs with an efficient algorithm to compute their semantics.

7.1 Classical SP Problems

A shortest path (SP) problem can be represented as a directed graph $G = (N, E)$, where each arc $e \in E$ from node p to node q ($p, q \in N$) has associated with a label representing the cost of the arc from p to q.

There are four versions of the problem: the single pair problem, the single source problem, the single sink problem, and the all pair problem. The single source and the single sink problems, however, are directional duals of each other, the single pair problem at least partially solves the single source/sink problem, and one way to solve the all pair problem is to solve n single source/sink problems. For these reasons, the single sink (source) is fundamental and we concentrate on it. Given a set S of nodes and any node v in S (the sink), a solution of the problem consists of finding a path (or a set of paths) between any node of S and v, whose cost is minimal.

Consider for example the SP problem represented in Figure 7.1: each arc has associated with it a label representing the cost (in money, time, space, ...) of that arc. In this example node v is the sink. Thus, given any node, we want to find a path from this node to the sink v (if it exists) that minimizes the cost.

To represent the classical version of SP problems, we consider SCLP programs over the semiring $S = \langle N, \min, +, +\infty, 0\rangle$, which, as noted above, is an appropriated framework to represent constraint problems where one wants to minimize the sum of the costs of the solutions.

From any SP problem we can build an SCLP program as follows. For each arc we have two clauses: one describes the arc and the other one its cost. More precisely, the head of the first clause represents the starting node, and its body contains both the final node and a predicate, say c, representing the cost of the

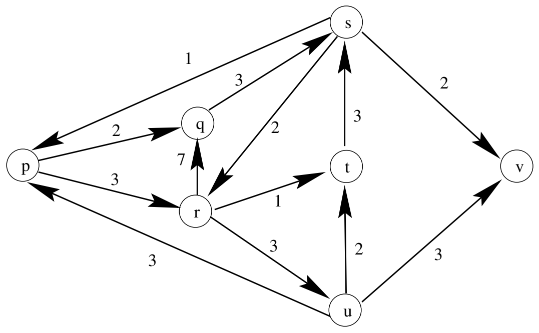


Fig. 7.1. An SP problem

arc. Then, the second clause is a fact associating with predicate c its cost (which is a semiring element). For example, if we consider the arc from p to q with cost 2, we have the clause

`p :-` c_{pq} `, q.`

and the fact

c_{pq} `:- 2.`

Finally, we must code that we want v to be the final node of the path. This is done by adding a clause of the form `v :- 0.` Note also that any node can be required to be the final one, not just those nodes without outgoing arcs (like v is in this example). The whole program corresponding to the SP problem in Figure 7.1 can be seen in Table 7.1.

To compute a solution of the SP problem it is enough to perform a query in the SCLP framework; for example, if we want to compute the cost of the path from r to v we have to perform the query `:- r`. For this query, we obtain the value 6, that represents the cost of the best path(s) from r to v.

`p :-` c_{pq} `, q.`	c_{pq} `:- 2.`
`p :-` c_{pr} `, r.`	c_{pr} `:- 3.`
`q :-` c_{qs} `, s.`	c_{qs} `:- 3.`
`r :-` c_{rq} `, q.`	c_{rq} `:- 7.`
`r :-` c_{rt} `, t.`	c_{rt} `:- 1.`
`r :-` c_{ru} `, u.`	c_{ru} `:- 3.`
`s :-` c_{sp} `, p.`	c_{sp} `:- 1.`
`s :-` c_{sr} `, r.`	c_{sr} `:- 2.`
`s :-` c_{sv} `, v.`	c_{sv} `:- 2.`
`t :-` c_{ts} `, s.`	c_{ts} `:- 3.`
`u :-` c_{up} `, p.`	c_{up} `:- 3.`
`u :-` c_{ut} `, t.`	c_{ut} `:- 2.`
`u :-` c_{uv} `, v.`	c_{uv} `:- 3.`
`v :- 0.`	

Table 7.1. The SCLP program representing the SP problem in Figure 7.1.

Notice that to represent classical SP problems in SCLP, we do not need any variable. Thus the resulting program is propositional. This program, however, while giving us the cost of the shortest paths, does not give us any information about the arcs which form such paths. This information could be obtained by providing each predicate with an argument, which represents the arc chosen at each step.

Figure 7.2 shows the same SP problem of Figure 7.1 where the arcs outgoing each node have been labeled with different labels to distinguish them. Such labels can then be coded into the corresponding SCLP program to "remember" the arcs traversed during the path corresponding to a solution. For example, clause

`p :-` c_{pq}`, q.`

would be rewritten as

`p(a) :-` c_{pq}`, q(X).`

Here, constant a represents one of the arcs going out of p: the one which goes to q. If all clauses are rewritten similarly, then the answer to a goal like `:- r(X)` will be both a semiring value (in our case 6) and a substitution for X. This substitution will identify the first arc of a shortest path from r to v. For example, if we have $X = b$, it means that the first arc is the one that goes from r to t. To find a complete shortest path, we just need to compare the semiring values associated with each instantiated goal, starting from r and following the path. For example, in our case (of the goal $\exists X.r(X)$) we have that the answer to the goal will be $X = c$ with semiring value 6. Thus, we know that a shortest path from r to v can start with the arc from r to u. To find the following arc of this path, we compare the semiring values of $u(a)$, $u(b)$, and $u(c)$. The result is that $u(c)$ has the smallest value, which is 3. Hence, the second arc of the shortest path we are constructing is the one from u to v. The path is now finished because we reached v, which is our sink.

Notice that a shortest path could be found even if variables are not allowed in the program, but more work is needed. In fact, instead of comparing different instantiations of a predicate, we need to compare the values associated with the predicates that represent nodes reachable by alternative arcs starting from

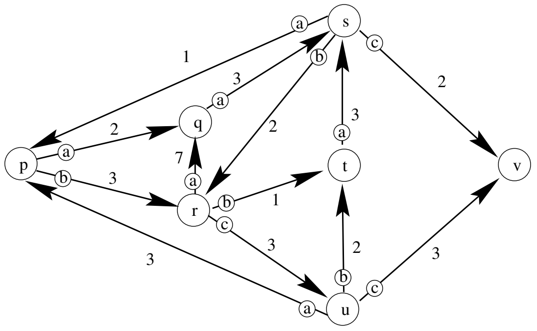


Fig. 7.2. An SP problem with labeled arcs

a certain node, and sum them to the cost of such arcs. For example, instead of comparing the values of $p(a)$ and $p(b)$ (Figure 7.2), we have to compare the values of $q + 2$ and of $r + 3$ (Figure 7.1).

A third alternative to compute a shortest path, and not only its cost, is to use lists: by replacing each clause of the form

p :- c_{xy}, q.

with the clause

p([a|T]) :- c_{xy}, q(T).

during the computation we also build the list containing all arcs that constitute the corresponding path. Thus, by giving the goal `:- p(L).`, we would get both the cost of a shortest path and also the shortest path itself, represented by the list L.

An alternative representation, probably more familiar for CLP-ers, of SP problems in SCLP is one where there are facts of the form

```
c(p,q) :- 2.
⋮
c(u,v) :- 3.
```

to model the graph, and the two clauses

```
path(X,Y) :- c(X,Y).
path(X,Y) :- c(X,Z), path(Z,Y).
```

to model paths of length one or more. In this representation the goal `:- path(p,v).` represents the cost of the shortest path from p to v. This representation is obviously more compact than the one in Table 7.1, and has equivalent results and properties. In this chapter, however, we will use the simpler representation, used in Table 7.1, where all clauses have, at most, one predicate in the body. The possibility of representing SP problems with SCLP programs containing only such kind of clauses is important, since it will allow us to use efficient algorithms to compute the semantics of such programs (see Section 7.4 for more details).

7.2 Partially-Ordered SP Problems

Sometimes, the costs of the arcs are not elements of a totally ordered set. A typical example is obtained when we consider multi-criteria SP problems.

In this case the goal of the problem is to find a solution that is *Pareto-optimal.* A solution is Pareto-optimal if by reallocation you cannot make someone better off without making someone else worse off. In Pareto's words [159]:

> We will say that the members of a collectivity enjoy *maximium ophelimity* in a certain position when it is impossible to find a way of moving from that position very slightly in such a manner that the ophelimity enjoyed by each of the individuals of that collectivity increases or decreases. That is to say, any small displacement in departing from that position necessarily has the effect of increasing the ophelimity which cer-

tain individuals enjoy, and decreasing that which others enjoy, of being agreeable to some, and disagreeable to others.

Perhaps the best description of Pareto-optimality is the underutilized one coined by Maurice Allais [6]: an allocation is "Pareto-optimal" if there is an "absence of distributable surplus".

As an example of multi-criteria SP problem we can consider the situation where an arc represents a piece of highway between two cities; in this case, we can label each arc both with the cost and with the time needed to follow this piece. The goal is to find the paths that have both the minimum overall cost and the minimum overall time. In this example, there may be cases in which the labels of two arcs are not compatible, like $\langle \$5, 20' \rangle$ and $\langle \$7, 15' \rangle$. In general, when we have a partially ordered set of costs, it may be possible to have several paths, all of which are not *dominated* by others, but which have different incomparable costs.

Consider, for example, the multi-criteria SP problem shown in Figure 7.3: each arc has associated with it a pair representing the weight of the arc in terms of *cost* and *time*. Given any node, we want to find a path from this node to v (if it exists) that minimizes both criteria.

We can translate this SP problem into the SCLP program in Table 7.2. This program works over the semiring

$$\langle N^2, \text{min'}, +', \langle +\infty, +\infty \rangle, \langle 0, 0 \rangle \rangle,$$

where min' and $+'$ are classical min and $+$, suitably extended to pairs. In practice, this semiring is obtained by putting together, via the Cartesian product, two instances of the semiring $\langle N, \min, +, +\infty, 0 \rangle$ (we recall that the Cartesian product of two c-semirings is a c-semiring as well [47]). One of the instances is used to deal with the cost criteria, the other one is for the time criteria. By working on the combined semiring, we can deal with both criteria simultaneously: the partial order will tell us when a pair $\langle$ cost, time $\rangle$ is preferable to another one, and also when they are not comparable.

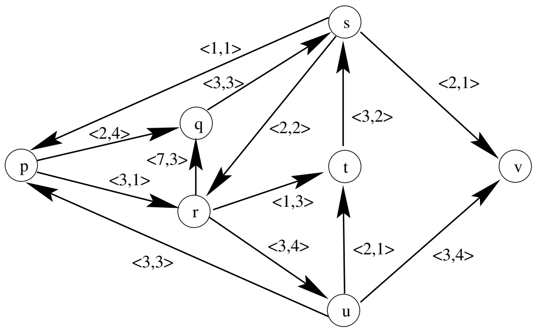


Fig. 7.3. A multi-criteria SP problem

```
p  :- c_pq, q.              c_pq :- < 2,4 >.
p  :- c_pr, r.              c_pr :- < 3,1 >.
q  :- c_qs, s.              c_qs :- < 3,3 >.
r  :- c_rq, q.              c_rq :- < 7,3 >.
r  :- c_rt, t.              c_rt :- < 1,3 >.
r  :- c_ru, u.              c_ru :- < 3,4 >.
s  :- c_sp, p.              c_sp :- < 1,1 >.
s  :- c_sr, r.              c_sr :- < 2,2 >.
s  :- c_sv, v.              c_sv :- < 2,1 >.
t  :- c_ts, s.              c_ts :- < 3,2 >.
u  :- c_up, p.              c_up :- < 3,3 >.
u  :- c_ut, t.              c_ut :- < 2,1 >.
u  :- c_uv, v.              c_uv :- < 3,4 >.
v  :- < 0,0 >.
```

Table 7.2. The SCLP program representing the multi-criteria SP problem in Figure 7.3.

To give an idea of another practical application of partially ordered SP problems, just think of network routing problems where we need to optimize according to the following criteria: minimize the time, minimize the cost, minimize the number of arcs traversed, and maximize the bandwidth. The first three criteria correspond to the same semiring, which is $\langle N, min, +, +\infty, 0\rangle$, while the fourth criteria can be characterized by the semiring $\langle [0, U], max, min, 0, U\rangle$, where U is the maximal bandwidth in an arc. In this example, we have to work on a semiring that is obtained by vectorizing all these four semirings. Each of the semirings is totally ordered but the resulting semiring, whose elements are four-tuples, is partially ordered.

Notice that the only difference between the structure of the program in Table 7.2 and the one in the previous section is that here we have costs represented by pairs of values, and, since we have a partial order, two such pairs may possibly be incomparable. This may lead to a strange situation while computing the semantics of a given goal. For example, if we want to compute the cost and time of a best path from p to v, by giving the query `:- p.`, the resulting answer in this case is the value $\langle 7, 7\rangle$. While the semiring value obtained in totally ordered SCLP programs represented the cost of one of the shortest paths, here it is possible that there are no paths with this cost: the obtained semiring value is in fact in general the greatest lower bound of the costs of all the paths from p to v. This behavior comes from the fact that, if different refutations for the same goal have different semiring values, the SCLP framework combines them via the $+$ operator of the semiring (which, in the case of our example, is the min' operator). If the semiring is partially ordered, it may be that $a + b$ is different from both a and b. On the contrary, if we have a total order $a+b$ is always either a or b.

This, of course, is not satisfactory, because usually one does not want to find the greatest lower bound of the costs of all paths from the given node to the sink, but rather prefers to have one of the non-dominated paths. To solve this problem, we can add variables to the SCLP program, as we did in the previous section, and also change the semiring. In fact, we now need a semiring that allows us to associate with each node the set of the costs of all non-dominated paths from there to the sink. In other words, starting from the semiring $S = \langle A, +, \times, 0, 1\rangle$ (which, we recall, in our case is $\langle N^2, min', +', \langle +\infty, +\infty\rangle, \langle 0, 0\rangle\rangle$), we now have to work with the semiring $P^H(S) = \langle P^H(A), \uplus, \times^*, \emptyset, A\rangle$, where:

- $P^H(A)$ is the Hoare Power Domain [182] of A, that is, $P^H(A) = \{S \subseteq A \mid x \in S, y \leq_S x \text{ implies } y \in S\}$. In words, $P^H(A)$ is the set of all subsets of A which are downward closed under the ordering $\leq_S$. It is easy to show that such sets are isomorphic to those containing just the non-dominated values. Thus, in the following we will use this more compact representation for efficiency purposes. In this compact representation, each element of $P^H(A)$ will represent the costs of all non-dominated paths from a node to the sink;
- the top element of the semiring is the set A (its compact form is $\{1\}$, which in our example is $\{\langle 0, 0\rangle\}$);
- the bottom element is the empty set;
- the additive operation $\uplus$ is the *formal union* [182] that takes two sets and obtains their union;
- the multiplicative operation $\times^*$ takes two sets and produces another set obtained by multiplying (using the multiplicative operation $\times$ of the original semiring, in our case +') each element of the first set with each element of the second one;
- the partial order of this semiring is as follows: $a \leq_{P^H(S)} b$ iff $a \uplus b = b$, that is, for each element of a, there is an element in b which dominates it (in the partial order $\leq_S$ of the original semiring).

From the theoretical results in [182], adapted to consider c-semirings, we can prove that $P^H(S)$ and its more compact form are indeed isomorphic. Moreover, we can also prove that given a c-semiring S, the structure $P^H(S)$ is a c-semiring as well.

Theorem 7.2.1 ($P^H(S)$ is a c-semiring). *Given a c-semiring $S = \langle A, +, \times, 0, 1\rangle$, the structure $P^H(S) = \langle P^H(A), \uplus, \times *, \emptyset, A\rangle$ obtained using the* Power domain of Hoare *operator is a c-semiring.*

Proof. It easily follows from the properties of the $\times$ operator in the c-semiring S and from the properties (commutativity, associativity, and idempotency) of the formal union $\uplus$ in $P^H(S)$.

Note that in this theorem we do not need any assumption over the c-semiring S. Thus, the construction of $P^H(S)$ can be done for any c-semiring S. Notice also that, if S is totally ordered, the c-semiring $P^H(S)$ does not give any additional information w.r.t. S. In fact, if we consider together the empty set (with the meaning that there are no paths) and the set containing only the bottom of A (meaning that there exists a path whose cost is 0), it is possible to build an isomorphism between S and $P^H(S)$ by mapping each element p (a set) of $P^H(A)$ onto the element a of A such that $a \in p$ and a dominates all elements in the set p.

The only change we need to make to the program with variables, in order to work with this new semiring, is that costs now have to be represented as singleton sets. For example, clause c_{pq} `:- < 2, 4 >.` will become c_{pq} `:-` $\{$`< 2, 4 >`$\}$`..`

Let us now see what happens in our example if we move to this new semiring. First we give a goal like `:- p(X)..` As the answer, we get a set of pairs, representing the costs of all non-dominated paths from p to v. All these costs are non-comparable in the partial order, thus the user is requested to make a choice. This choice, however, could identify a single cost or also a set of them. In this second case, it means that the user does not want to commit to a single path from the beginning and rather prefers to maintain some alternatives. The choice of one cost of a specific non-dominated path will thus be delayed until later. If instead the user wants to commit to one specific cost at the beginning, say $\langle c_1, c_2\rangle$, he/she then proceeds to find a path that costs exactly $\langle c_1, c_2\rangle$. By comparing the answers for all goals of the form $p(a)$, where a represents one of the arcs out of p, we can see which arc can start a path with cost $\langle c_1, c_2\rangle$. In fact, such an arc will be labeled $\langle c_{1a}, c_{2a}\rangle$ and will lead to a node with an associated set of costs $\langle c_3, c_4\rangle$ such that $\langle c_3, c_4\rangle \times \langle c_{1a}, c_{2a}\rangle = \langle c_1, c_2\rangle$. Suppose it is the arc from p to q, which has cost $\langle 7, 3\rangle$. Now we do the same with goals of the form $q(a)$, to see which is the next arc to follow. We now, however, have to look for the presence of a pair $\langle c_3, c_4\rangle$ such that $\langle c_3, c_4\rangle \times \langle 7, 3\rangle = \langle c_1, c_2\rangle$.

Notice that each time we look for the next arc, we choose just one alternative and we disregard the others. If we used a fixpoint (or any bottom-up) semantics to compute the answer of the initial goal :- p(X)., then all the other answers we need for this method have already been computed, thus the method does not require any additional computational effort to find a non-dominated path.

Notice also that the sets of costs associated with each node are non-dominated. Thus, the size of these sets in the worst case is the size of the maximal "horizontal slice" of the partial order: if we can have at most N non-dominated elements in the partial order, then each of such sets will have size at most N. Of course, in the worst case N could be the size of the whole semiring (when the partial order is completely "flat").

Most classical methods to handle multi-criteria SP problems find the shortest paths by considering each criteria separately, while our method deals with all criteria at once. This allows us to obtain optimal solutions, which are not generated by looking at each single criteria. In fact, some optimal solutions could be non-optimal in each of the single criteria, but still are incomparable in the overall ordering. Thus, we offer the user a greater set of non-comparable optimal solutions. For example, by using the time-cost multi-criteria scenario, the optimal solution w.r.t. time could be 1 minute (with cost of 10 dollars), while the optimal solution w.r.t. cost could be 1 dollar (with time of 10 minutes).

By considering both criteria together, we could also obtain the solution with 2 minutes and 2 dollars!

Finally, this method is applicable not only to the multi-criteria case, but to any partial order, giving us a general way to find a non-dominated path in a partially-ordered SP problem. It is important to notice here the flexibility of the semiring approach, which allows us to use the same syntax and computational engine, but on a different semiring, to compute different objects.

7.3 Modality-Based SP Problems

Until now we have considered situations in which an arc is labeled by its cost, be it one element or a tuple of elements as in the multi-criteria case. Sometimes, however, it may be useful to also associate with each arc information about the *modality* to be used to traverse the arc.

For example, interpreting arcs as links between cities, we may want to model the fact that we can cover such an arc by *car*, or by *train*, or by *plane*. Another example of a modality could be the *time of the day* in which we cover the arc, like *morning*, *afternoon*, and *night*. In all these cases, the cost of an arc may depend on its modality.

An important thing to notice is that a path could be made of arcs which are not necessarily all covered with the same modality. For example, the path between two cities can be made of arcs, some of which are covered in the morning and others in the afternoon. Moreover, it can be that different arcs have different sets of modalities. For example, from city A to city B we can use both the car or the train, and from city B to city C we can only use the plane. Thus, modalities cannot be simply treated by selecting a subset of arcs (all those with the same modality).

An example of an SP problem with the three modalities representing car (c), plane (p), and train (t) can be seen in Figure 7.4. Here the problem is to find a shortest path from any node to v, and to know both its cost and also the modalities of its arcs.

This SP problem with modalities can be modeled via the SCLP program in Table 7.3. In this program, the variables represent the modalities.

If we ask the query `:-p(c).`, it means that we want to know the smallest cost for a trip from p to v using the *car*. The result of this query in our example is $p(c) = 9$ (using the path $p - r - u - v$).

Notice that the formulation shown in Figure 7.3 puts some possibly undesired constraints on the shortest path to be found. In fact, by using the same variable in all the predicates of a rule, we make sure that the same modality (in our case the same transport mean) is used throughout the whole path. If instead we want to allow different modalities in different arcs of the path, then we just need to change the rules by putting a new variable on the last predicate of each rule. For example, rule

`p(X) :-` c_{pq}`(X), q(X).`

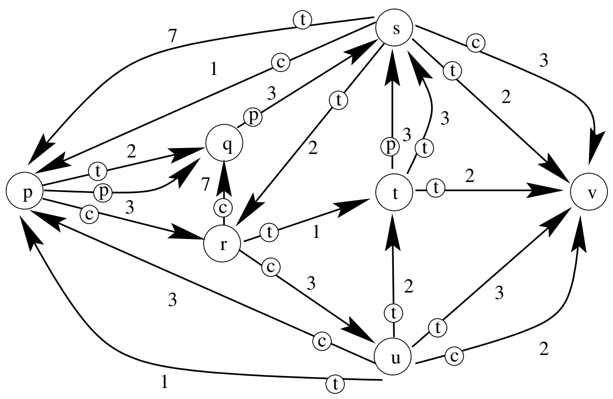


Fig. 7.4. An SP problem with modalities

p(X) :- c_{pq}(X), q(X).	c_{pq}(t) :- 2.
p(X) :- c_{pr}(X), r(X).	c_{pq}(p) :- 3.
q(X) :- c_{qs}(X), s(X).	c_{pr}(c) :- 3.
r(X) :- c_{rq}(X), q(X).	c_{qs}(p) :- 3.
r(X) :- c_{rt}(X), t(X).	c_{rq}(c) :- 7.
r(X) :- c_{ru}(X), u(X).	c_{rt}(t) :- 1.
s(X) :- c_{sp}(X), p(X).	c_{ru}(c) :- 3.
s(X) :- c_{sr}(X), r(X).	c_{sp}(c) :- 1.
s(X) :- c_{sv}(X), v(X).	c_{sp}(t) :- 7.
t(X) :- c_{ts}(X), s(X).	c_{sr}(t) :- 2.
u(X) :- c_{up}(X), p(X).	c_{sv}(t) :- 2.
u(X) :- c_{ut}(X), t(X).	c_{sv}(c) :- 3.
u(X) :- c_{uv}(X), v(X).	c_{ts}(p) :- 3.
v(X) :- 0.	c_{ts}(t) :- 3.
	c_{up}(c) :- 3.
	c_{up}(t) :- 1.
	c_{ut}(t) :- 2.
	c_{uv}(t) :- 3.
	c_{uv}(c) :- 2.

Table 7.3. The SCLP program representing the SP problem with modalities.

would become
p(X) :- c_{pq}(X), q(Y).
Now we can use a modality for the arc from p to q, and another one for the next arc. In this new program, asking the query `:-p(c).` means that we want to know the smallest cost for a trip from p to v using the car in the first arc.

The same methods used in the previous sections to find a shortest path, or a non-dominated path in the case of a partial order, can be used in this kind of SCLP programs as well. Thus, we can put additional variables in the predicates to represent alternative arcs outgoing the corresponding nodes, and we can shift to the semiring containing sets of costs to find a non-dominated

path. In particular, a clause like
`p(X) :-` c_{pq}`(X), q(Y).`
would be rewritten as
`p(X,a) :-` c_{pq}`(X), q(Y,Z).`

7.4 An SP Algorithm for a Class of SCLP Programs

Summarizing what we did in the previous sections, we can say that the general form of the SCLP programs we use to represent SP problems consists of several clauses for each predicate p_i, where each clause body has one constraint and one other predicate p_j, plus one special clause for the sink predicate (with a **0** in the body), plus the facts defining the costs of the arcs. Table 7.4 shows this general form (only the clauses and without variables). It is important to notice that these are not general SCLP programs, since there is always one predicate symbol in each clause body (since the constraint can be replaced by its cost). In the logic programming literature, these programs are called *linear*, because of the restriction on the number of predicates in the bodies of the clauses.

Given any SCLP program of the form shown in Table 7.4, its semantics can be obtained using classical methodologies (for the CLP literature). We now, however, will show an additional method, based on a classical algorithm for SP problems, suitably adapted to c-semirings, to obtain the same semantics. The algorithm we will use is the one developed by Floyd and Warshall [99]. To put the SCLP program into a shape that can be handled by this algorithm, we first need to transform it into a system of linear equations, one for each predicate: the left hand side of the equation is the chosen predicate, while the right hand side is obtained by combining the bodies of all the clauses defining the predicates

p_1 :- c_{12}, p_2.
$\vdots$
p_1 :- c_{1n}, p_n.
p_2 :- c_{12}, p_1.
p_2 :- c_{13}, p_3.
$\vdots$
p_2 :- c_{2n}, p_n.
$\vdots$
p_v :- $\mathbf{0}$.
$\vdots$
p_n :- c_{n1}, p_1.
$\vdots$
p_n :- $c_{n,n-1}, p_{n-1}$.

Table 7.4. The general form of an SCLP program representing an SP problem.

via the + operator, and replacing the comma with the $\times$ operator in each body. The result of this transformation, applied to the program of Table 7.4, can be seen in Table 7.5.

From this system of equations we can now build a graph with a node i for each predicate p_i, and where the cost of the arc between nodes i and j is given by predicate c_{ij}. In this way, this system of equations can be interpreted also as the matrix showing, for each pair of nodes, the cost of the arc (if it exists) between such nodes. In fact, each column (and row) can be associated with one node.

Given this matrix, the algorithm works by selecting each triple of nodes, say i, j and k, and performing the following assignment: $c_{ij} := c_{ij} + (c_{ik} \times c_{kj})$, where the + and $\times$ operators in this assignment refer to the operations of the chosen semiring.

The triples are chosen by first selecting a value for index k (which is the intermediate node in the two-step path), and then varying the indices i and j in all possible ways (to consider all two-step paths from i to j). After considering all values for k, that is, after n^3 steps, the value of the element in row i and column v represents the cost of the shortest path from node i to v. In terms of SCLP semantics, this is the semantics of predicate p_i.

This method to obtain the semantics of an SCLP program of the form in Table 7.4 can be used independently of the semiring underlying the given program. In fact, as noted above, the version we use of the original Floyd-Warshall algorithm does not depend on the meaning of the + and $\times$ operators. In particular, it can also be used for partially ordered semirings. If the semiring we consider is totally ordered, however, we can also use any other classical algorithm for solving SP problems.

7.5 Best-Tree Search in AND-OR Graphs

In this section we extend our reasoning to Best-Tree Search over $AND - OR$ graphs. The main idea underlying this extension concerns the use of non-linear

$$
\begin{array}{lllllll}
p_1 & = & & c_{12} \times p_2 + \dots & & + c_{1n} \times p_n & \\
p_2 & = & c_{21} \times p_1 + & & c_{23} \times p_3 + \dots & + c_{2n} \times p_n & \\
\vdots & & & & & & \\
p_v & = & & & & & \mathbf{0} \\
\vdots & & & & & & \\
p_n & = & c_{n1} \times p_n + \dots & & + c_{nn-1} \times p_{n-1} & &
\end{array}
$$

Table 7.5. The system of equations corresponding to the SCLP program form of Table 7.4.

clauses in SCLP, that is, clauses which have more than one atom in their body. In fact, in this way we can represent trees instead of paths.

7.5.1 AND-OR Graphs and Best Solution Trees

An AND-OR graph is defined essentially as a hypergraph. Namely, instead of arcs connecting pairs of nodes there are hyperarcs connecting n-tuple of nodes ($n = 1, 2, 3, \ldots$). Hyperarcs are called *connectors* and they must be considered as directed from their first node to all others. Formally an AND-OR graph is a pair $G = (N, C)$, where N is a set of *nodes* and C is a set of connectors

$$C \subseteq N \times \bigcup_{i=1}^{k} N^i.$$

Each k-connector $(n_{i_0}, n_{i_1}, \ldots, n_{i_k})$ is an ordered $(k+1)$-tuple, where n_{i_0} is the *input* node and $n_{i_1}, \ldots, n_{i_k}$ are the *output* nodes. We say that n_{i_0} is the *predecessor* of $n_{i_1}, \ldots, n_{i_k}$ and these nodes are the *successors* of n_{i_0}. Note that when $C \subseteq N^2$ we have a usual graph whose arcs are the 1-connectors. Note that there are also 0-connectors, i.e., connectors with one input and no output node.

In Figure 7.5 we give an example of an AND-OR graph. The nodes are $n_0, \ldots, n_8$. The 0-connectors are represented as a line ending with a square, whereas k-connectors ($k \geq 0$) are represented as k directed lines connected together. For instance, (n_0, n_1) and (n_0, n_5, n_4) are the 1-connector and 2-connector, respectively, with input node n_0.

An AND tree is a special case of an AND-OR graph, where every node appears exactly twice in the set of connectors C, once as an input node of some connector,

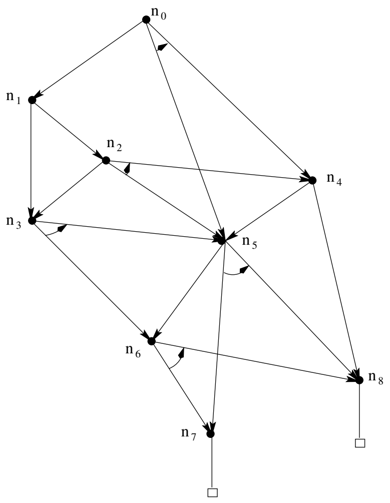


Fig. 7.5. An example of an AND-OR graph

and once as an output node of some other connector. The only exception is a node called *root* which appears only once, as an input node of some connector. The *leaves* of an AND tree are those nodes which are input nodes of a 0-connector. An example of an AND tree is given in Figure 7.6. Here n'_7, n'_8, n''_7 and n''_8 are leaves.

Given an AND-OR graph G, an AND tree H is a *solution tree of G with start node n_0*, if there is a function g mapping nodes of H into nodes of G such that:

– the root of H is mapped in n_0; and
– if $(n_{i_0}, n_{i_1}, \ldots, n_{i_k})$ is a connector of H, then $(g(n_{i_0}), g(n_{i_1}), \ldots, g(n_{i_k}))$ is a connector of G.

Informally, a solution tree of an AND-OR graph is analogous to a path of an ordinary graph. It can be obtained by selecting exactly one outgoing connector for each node. For instance, the AND tree in Figure 7.6 is a solution tree of the graph in Figure 7.5 with start node n_0, if we let $g(n'_0) = n_0, g(n'_5) = n_5, g(n'_7) = n_7, g(n'_8) = n_8, g(n'_4) = n_4, g(n''_5) = n_5, g(n''_6) = n_6, g(n''_7) = n_7, g(n''_8) = n_8$. Note that distinct nodes of the tree can be mapped into the same node of the graph.

A *functionally weighted* AND-OR *graph* is an AND-OR graph with a k-adic function over the reals (*cost function*) associated with each k-connector. In particular, a constant is associated with each 0-connector. It is easy to see that if the functionally weighted graph is an AND tree H, a *cost* can be given to it, just

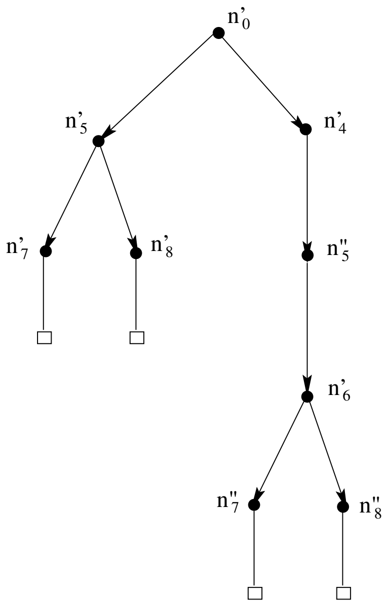


Fig. 7.6. An example of an AND tree

evaluating the functions associated with the connectors. Recursively, to every subtree of H with root in node n_{i_0} a cost c_{i_0} is given as follows:

- If n_{i_0} is a leaf, then its cost is the associated constant.
- If n_{i_0} is the input node of a connector $(n_{i_0}, n_{i_1}, \ldots, n_{i_k})$, then its cost is $c_{i_0} = f_r(c_{i_1}, \ldots, c_{i_k})$ where f_r is the function cost associated with the connector, and $c_{i_1}, \ldots, c_{i_k}$ are the costs of the subtrees rooted at nodes $n_{i_1}, \ldots, n_{i_k}$.

The general optimization problem can be stated as follows: *given a functionally weighted* AND-OR *graph, find a minimal cost solution tree with start node* n_0. In general, the function used to assign a value to the input node n_{i_0} of a k-connector $(n_{i_0}, n_{i_1}, \ldots, n_{i_k})$ is of the form $f_r(c_{i_1}, \ldots, c_{i_k}) = a_r + c_{i_1} + \ldots + c_{i_k}$ where a_r is a constant associated to the connector and $c_{i_1}, \ldots, c_{i_k}$ are the costs of the subtrees rooted at nodes $n_{i_1}, \ldots, n_{i_k}$.

In the following we will show that this cost function is only an instantiation of a more general one based on the notion of *c-semiring.*

7.5.2 AND-OR Graphs Using SCLP

In this section we will show how to represent an AND-OR graph as an SCLP program over a specific c-semiring. As soon as we have described the AND-OR graph as an SCLP program, the problem of finding the *best* tree in the graph rooted at n_i becomes the problem of finding the *best* refutation for the goal $\mathtt{n}_i$, and this is *exactly* what the semantic of the SCLP language does.

To represent the classical problem where the meaning of *best* tree is the tree whose *sum* of the costs of its connectors is *minimum*, we consider an SCLP program over the semiring $S = \langle N, \min, +, +\infty, 0\rangle$, which, as noted above, is an appropriated framework to represent constraint problems where one wants to minimize the sum of the costs of the solutions.

Consider for example the AND-OR graph problem represented in Figure 7.7: each connector $(n_{i_1}, \ldots, n_{i_k})$ has associated a label $f_{i_1,\ldots,i_k}$ representing its cost.

From this AND-OR graph problem we can build an SCLP program as follows. For each connector we have two clauses: one describes the connector and the other one its cost. More precisely, the head of the first clause represents the starting node n_{i_0}, and its body contains both the final nodes and a predicate, say $f_{i_0,\ldots,i_k}$, representing the cost of the connector from node n_{i_0} to nodes $n_{i_1}, \ldots, n_{i_k}$. Then, the second clause is a fact associating to predicate $f_{i_0,\ldots,i_k}$ its cost (which is a semiring element).

For example, if we consider the connector (n_0, n_1, n_2, n_3) with label f_0 in Figure 7.8, this can be represented by the two clauses

```
n0 :- f0, n1, n2, n3.
f0 :- c0.
```

The whole program corresponding to the AND-OR graph problem in Figure 7.7 can be seen in Table 7.5.2.

To solve the AND-OR graph problem it is enough to perform a query in the SCLP framework; for example, if we want to compute the cost of the best

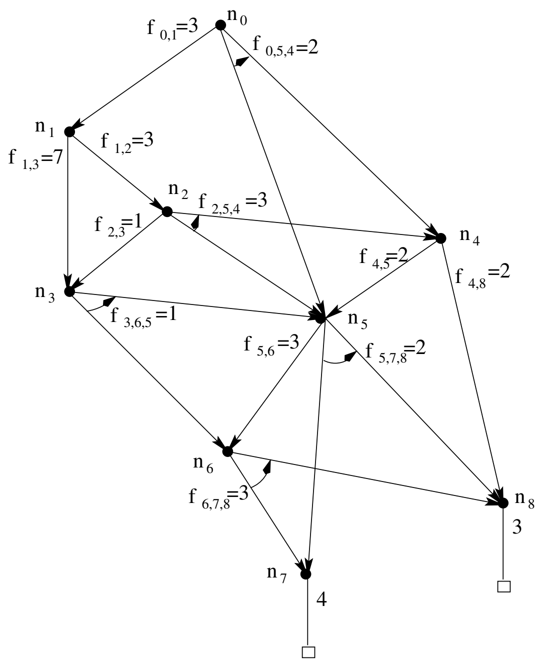


Fig. 7.7. A weighted AND-OR graph problem

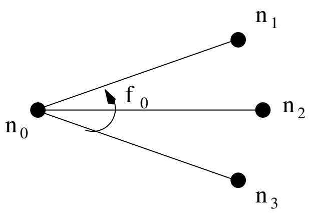


Fig. 7.8. A typical connector

tree rooted at n_2 we have to perform the query n_2. The operational semantics machinery finds all trees (modulo some cuts due to heuristics) and then combines all the solutions via the additive operation of the semiring, which in this case is *min*. For this query, we obtain the value 17 that represents the cost of the best tree rooted at n_2.

Notice that to represent classical best tree problems in SCLP, we do not need any variable. Thus the resulting program is propositional. However, this program, while giving us the cost of the best tree, does not give us any information about the connectors which form such a tree. This information could be obtained by providing each predicate with an argument which represents

```
n_0 :- f_{0,5,4},n_5, n_4.          f_{0,5,4} :- 2.
n_0 :- f_{0,1},n_1.                 f_{0,1}   :- 3.
n_1 :- f_{1,2},n_2.                 f_{1,2}   :- 3.
n_1 :- f_{1,3},n_3.                 f_{1,3}   :- 7.
n_2 :- f_{2,3},n_3.                 f_{2,3}   :- 1.
n_2 :- f_{2,5,4},n_5,n_4.           f_{2,5,4} :- 3.
n_3 :- f_{3,6,5},n_6,N_5.           f_{3,6,5} :- 1.
n_4 :- f_{4,5},n_5.                 f_{4,5}   :- 2.
n_4 :- f_{4,8},n_8.                 f_{4,8}   :- 2.
n_5 :- f_{5,6},n_6.                 f_{5,6}   :- 3.
n_5 :- f_{5,7,8},n_7,n_8.           f_{5,7,8} :- 2.
n_6 :- f_{6,7,8},n_7,n_8.           f_{6,7,8} :- 3.
n_7 :- 4.                           n_8       :- 3.
```

Table 7.6. The SCLP program representing the AND-OR graph problem in Figure 7.7.

the connector chosen at each step, as we did for shortest path problems in the previous section.

The best tree for the program in Table 7.5.2, with sum of costs 17 as given by the semantics of the program, is represented in Figure 7.9.

In the examples given so far, we were interested in finding the tree with the minimum cost. Thus, the constant associated to each connector, that is, its cost, is in $\Re$ and the costs of the subtrees rooted at the sons of a certain node are *combined* together using the $+$ operator. More generally, the operation used to combine the value of the subtrees can be mapped over the c-semiring structure. More precisely, to describe the operation of combination we will use the $\times$ operator of the semiring; the additive operator will be useful instead to compare different trees, since the partial order $\leq_S$ is induced by $+$ operation of the semiring.

Notice that we can apply to best tree search all the extensions already defined in the previous sections for shortest paths. This means that we can use also a

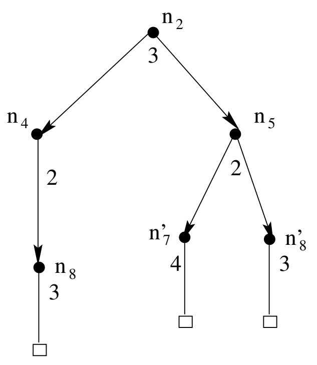


Fig. 7.9. The best tree corresponding to the program in Table 7.5.2

partially ordered set for the levels of preference of the connectors. This can be useful for multi-criteria best tree problems or for general partially ordered best tree problems. We can also use several classes of connectors and then search for a best tree where the connectors are all in the same class, or also of different classes (modality-based best tree problems).

7.6 Conclusions

In this chapter the shortest path problem taken from the operational research field has been considered and translated in a declarative fashion using the SCLP framework. This, on the one hand, enriches a class of SCLP programs with a semantics given using an "algorithm", and on the other hand gives a resolution methodology to the shortest path problems, based upon the semantics of a declarative language.

Moreover, the encoding of the shortest path in a soft constraint framework also provides for the possibility to apply all the methodologies of partial solving described in the previous chapters (that is abstraction, partial local consistency, and so on) when performing the constraint consistency steps of the resolution process.

8. Soft Concurrent Constraint Programming

Overview

Soft constraints extend classical constraints to represent multiple consistency levels, and thus provide a way to express preferences, fuzziness, and uncertainty. While there are many soft constraint solving formalisms, even distributed ones, by now there seems to be no concurrent programming framework where soft constraints can be handled. In this chapter we show how the classical concurrent constraint (cc) programming framework can work with soft constraints, and we also propose an extension of cc languages which can use soft constraints to prune and direct the search for a solution. We believe that this new programming paradigm, called soft cc (scc), can be also very useful in many web-related scenarios. In fact, the language level allows web agents to express their interaction and negotiation protocols, and also to post their requests in terms of preferences, and the underlying soft constraint solver can find an agreement among the agents even if their requests are incompatible.

The concurrent constraint (cc) paradigm [171] is a very interesting computational framework which merges together constraint solving and concurrency. The main idea is to choose a *constraint system* and use constraints to model communication and synchronization among concurrent agents.

Until now, constraints in cc were *crisp*, in the sense that they could only be satisfied or violated.

We think that many network-related problem could be represented and solved by using soft constraints. Moreover, the possibility to use a concurrent language on top of a soft constraint system, could lead to the birth of new protocols with an embedded constraint satisfaction and optimization framework.

In particular, the constraints could be related to a quantity to be minimized/maximized but they could also satisfy policy requirements given for performance or administrative reasons. This leads to change the idea of QoS in routing and to speak of *constraint-based* routing [13, 63, 71, 128]. Constraints are in fact able to represent in a declarative fashion the needs and the requirements of agents interacting over the web.

The features of soft constraints could also be useful in representing routing problems where an imprecise state information is given [66]. Moreover, since QoS is only a specific application of a more general notion of Service Level Agreement (SLA), many applications could be enhanced by using such a framework. As an example consider E-commerce: here we are always looking for establishing an agreement between a merchant, a client and possibly a bank. Also, all auction-based transactions need an agreement protocol. Moreover, also security protocol [18,20] and integrity policy analysis [40,41] have shown to be enhanced by using security levels instead of a simple notion of secure/insecure level . All these considerations advocate for the need of a soft constraint framework where optimal answers are extracted.

In this chapter, we first compare the semiring-based framework with constraint systems *"a la Saraswat"* and then we show how use it inside the cc framework. More precisely, we describe how to use soft constraints instead of classical ones within the original cc framework. In this scenario, the only addition to classical cc is the use of a function which transforms preference levels into a yees/no information of consistency.

The next step is the extension of the syntax and operational semantics of the language to deal with the semiring levels. Here, the main novelty with respect to cc is that tell and ask agents are equipped with a preference (or consistency) threshold which is used to determine their success, failure, or suspension, as well as to prune the search.

After a short summary of concurrent constraint programming (§8.1) we show how the concurrent constraint framework can be used to handle also soft constraints (§8.2). Then we integrate semirings inside the syntax of the language and we change its semantics to deal with soft levels (§8.3). Some notions of observables able to deal with a notion of optimization and with *success* (§8.5.1) and *fail* (§8.5.2) computations are then defined. Some examples (§8.4) and an application scenario (§8.6) conclude our presentation showing the expressivity of the new language. Finally, conclusions (§8.7) are added to point out the main results and possible directions for future research.

8.1 Concurrent Constraint Programming

The concurrent constraint (cc) programming paradigm [171] concerns the behaviour of a set of concurrent agents with a shared store, which is a conjunction of constraints. Each computation step possibly adds new constraints to the store. Thus information is monotonically added to the store until all agents have evolved. The final store is a refinement of the initial one and it is the result of the computation. The concurrent agents do not communicate directly with each other, but only through the shared store, by either checking if it entails a given constraint (*ask* operation) or adding a new constraint to it (*tell* operation).

Constraint Systems. A constraint is a relation among a specified set of variables. That is, a constraint gives some information on the set of possible values that these variables may assume. Such information is usually not complete since a constraint may be satisfied by several assignments of values of the variables (in contrast to the situation that we have when we consider a valuation, which tells us the only possible assignment for a variable). Therefore it is natural to describe constraint systems as systems of *partial* information [171].

The basic ingredients of a constraint system (defined following the information systems idea [177]) are a set D of *primitive constraints* or *tokens*, each expressing some partial information, and an entailment relation $\vdash$ defined on $\wp(D) \times D$ (or its extension defined on $\wp(D) \times \wp(D)$)[1] where $\wp(D)$ is the powerset of D. The entailment relation satisfies:

– $u \vdash P$ for all $P \in u$ (reflexivity) and
– if $u \vdash v$ and $v \vdash z$, then $u \vdash z$ for all $u, v, z \in \wp(D)$ (transitivity).

We also define $u \approx v$ if $u \vdash v$ and $v \vdash u$.

As an example of entailment relation, consider D as the set of equations over the integers; then $\vdash$ could include the pair $\langle \{x = 3, x = y\}, y = 3\rangle$, which means that the constraint $y = 3$ is entailed by the constraints $x = 3$ and $x = y$. Given $X \in \wp(D)$, let $\overline{X}$ be the set X closed under entailment. Then, a constraint in an information system $\langle \wp(D), \vdash\rangle$ is simply an element of $\overline{\wp(D)}$.

As it is well known [172], $\langle \overline{\wp(D)}, \subseteq\rangle$ is a complete algebraic lattice, the compactness of $\vdash$ gives the algebraic structure for $\overline{\wp(D)}$, with least element $true = \{P \mid \emptyset \vdash P\}$, greatest element D (which we will mnemonically denote $false$), glbs (denoted by $\sqcap$) given by the closure of the intersection and lubs (denoted by $\sqcup$) given by the closure of the union. The lub of chains is, however, just the union of the members in the chain. We use a, b, c, d and e to stand for elements of $\overline{\wp(D)}$; $c \supseteq d$ means $c \vdash d$.

The Hiding Operator: Cylindric Algebras. In order to treat the hiding operator of the language (see Definition 8.2.7), a general notion of existential quantifier for variables in constraints is introduced, which is formalized in terms of cylindric algebras. This leads to the concept of *cylindric constraint system* over an infinite set of variables V such that for each variable $x \in V$, $\exists_x : \overline{\wp(D)} \to \overline{\wp(D)}$ is an operation satisfying:

1. $u \vdash \exists_x u$;
2. $u \vdash v$ implies $(\exists_x u) \vdash (\exists_x v)$;
3. $\exists_x(u \sqcup \exists_x v) \approx (\exists_x u) \sqcup (\exists_x v)$;
4. $\exists_x \exists_y u \approx \exists_y \exists_x u$.

Procedure Calls. In order to model parameter passing, *diagonal elements* are added to the primitive constraints. We assume that, for x, y ranging in V, $\overline{\wp(D)}$ contains a constraint d_{xy}. If $\vdash$ models the equality theory, then the elements d_{xy} can be thought of as the formulas $x = y$. Such a constraint satisfies the following axioms:

1. $d_{xx} = true$,
2. if $z \neq x, y$ then $d_{xy} = \exists_z(d_{xz} \sqcup d_{zy})$,
3. if $x \neq y$ then $d_{xy} \sqcup \exists_x(c \sqcup d_{xy}) \vdash c$.

Note that the in the previous definition we assume the cardinality of the domain for x, y and z greater than 1 (otherwise, axioms 2 and 3 would not make sense).

[1] The extension is s.t. $u \vdash v$ iff $u \vdash P$ for every $P \in v$.

The language. The syntax of a cc program is show in Table 8.1: P is the class of programs, F is the class of sequences of procedure declarations (or clauses), A is the class of agents, c ranges over constraints, and x is a tuple of variables. Each procedure is defined (at most) once, thus nondeterminism is expressed via the + combinator only. We also assume that, in $p(x) :: A$, we have $vars(A) \subseteq x$, where $vars(A)$ is the set of all variables occurring free in agent A. In a program $P = F.A$, A is the initial agent, to be executed in the context of the set of declarations F. This corresponds to the language considered in [171], which allows only guarded nondeterminism.

In order to better understand the extension of the language that we will introduce later, let us remind here the operational semantics of the agents.

- agent "$success$" succeeds in one step,
- agent "$fail$" fails in one step,
- agent "$ask(c) \rightarrow A$" checks whether constraint c is entailed by the current store and then, if so, behaves like agent A. If c is inconsistent with the current store, it fails, and otherwise it suspends, until c is either entailed by the current store or is inconsistent with it;
- agent "$ask(c_1) \rightarrow A_1 + ask(c_2) \rightarrow A_2$" may behave either like A_1 or like A_2 if both c_1 and c_2 are entailed by the current store, it behaves like A_i if c_i only is entailed, it suspends if both c_1 and c_2 are consistent with but not entailed by the current store, and it behaves like "$ask(c_1) \rightarrow A_1$" whenever "$ask(c_2) \rightarrow A_2$" fails (and vice versa);
- agent "$tell(c) \rightarrow A$" adds constraint c to the current store and then, if the resulting store is consistent, behaves like A, otherwise it fails.
- agent $A_1 \| A_2$ behaves like A_1 and A_2 executing in parallel;
- agent $\exists_x A$ behaves like agent A, except that the variables in x are local to A;
- $p(x)$ is a call of procedure p.

A formal treatment of the cc semantics can be found in [58, 171]. Also, a denotational semantics of deterministic cc programs, based on closure operators, can be found in [171]. A more complete survey on several concurrent paradigms is given also in [81].

$$\begin{aligned}
P &::= F.A \\
F &::= p(x) :: A \mid F.F \\
A &::= success \mid fail \mid tell(c) \rightarrow A \mid E \mid A \| A \mid \exists_x A \mid p(x) \\
E &::= ask(c) \rightarrow A \mid E + E
\end{aligned}$$

Table 8.1. cc syntax

8.2 Concurrent Constraint Programming over Soft Constraints

Given a semiring $S = \langle \mathcal{A}, +, \times, \mathbf{0}, \mathbf{1} \rangle$ and an ordered set of variables V over a domain D, we will now show how soft constraints over S with a suitable pair of operators form a semiring, and then, we highlight the properties needed to map soft constraints over constraint systems *"a la Saraswat"* (as recalled in Section 8.1).

We start by giving the definition of the carrier set of the semiring.

Definition 8.2.1 (functional constraints). *We define* $\mathcal{C} = (V \to D) \to \mathcal{A}$ *as the set of all possible constraints that can be built starting from* $S = \langle \mathcal{A}, +, \times, \mathbf{0}, \mathbf{1} \rangle$*,* D *and* V*.*

A generic function describing the assignment of domain elements to variables will be denoted in the following by $\eta : V \to D$. Thus a constraint is a function which, given an assignment η of the variables, returns a value of the semiring.

Note that in this *functional* formulation, each constraint is a function and not a pair representing the variable involved and its definition. Such a function involves all the variables in V, but it depends on the assignment of only a finite subset of them. We call this subset the *support* of the constraint. For computational reasons we require each support to be finite.

Definition 8.2.2 (constraint support). *Consider a constraint* $c \in \mathcal{C}$*. We define his support as* $supp(c) = \{v \in V \mid \exists \eta, d_1, d_2 . c\eta[v := d_1] \neq c\eta[v := d_2]\}$*, where*

$$\eta[v := d]v' = \begin{cases} d & \textit{if } v = v', \\ \eta v' & \textit{otherwise.} \end{cases}$$

Note that $c\eta[v := d_1]$ means $c\eta'$ where η' is η modified with the association $v := d_1$ (that is the operator $[\ ]$ has precedence over application).

Definition 8.2.3 (functional mapping). *Given any soft constraint* $\langle def, \{v_1, \ldots, v_n\} \rangle \in C$*, we can define its corresponding function* $c \in \mathcal{C}$ *s.t.* $c\eta[v_1 := d_1] \ldots [v_n := d_n] = def(d_1, \ldots, d_n)$*. Clearly* $supp(c) \subseteq \{v_1, \ldots, v_n\}$*.*

Definition 8.2.4 (Combination and Sum). *Given the set* $\mathcal{C}$*, we can define the combination and sum functions* $\otimes, \oplus : \mathcal{C} \times \mathcal{C} \to \mathcal{C}$ *as follows:*

$$(c_1 \otimes c_2)\eta = c_1\eta \times_S c_2\eta \qquad \textit{and} \qquad (c_1 \oplus c_2)\eta = c_1\eta +_S c_2\eta.$$

Notice that function $\otimes$ has the same meaning of the already defined $\otimes$ operator (see Section 2.2) while function $\oplus$ models a sort of disjunction.

By using the $\oplus_S$ operator we can easily extend the partial order $\leq_S$ over $\mathcal{C}$ by defining $c_1 \sqsubseteq_S c_2 \iff c_1 \oplus_S c_2 = c_2$. In the following, when the semiring will be clear from the context, we will use $\sqsubseteq$.

We can also define a unary operator that will be useful to represent the unit elements of the two operations $\oplus$ and $\otimes$. To do that, we need the definition of constant functions over a given set of variables.

Definition 8.2.5 (constant function). *We define function $\bar{a}$ as the function that returns the semiring value a for all assignments η, that is, $\bar{a}\eta = a$. We will usually write $\bar{a}$ simply as a.*

An example of constants that will be useful later are $\bar{\mathbf{0}}$ and $\bar{\mathbf{1}}$ that represent respectively the constraint associating $\mathbf{0}$ and $\mathbf{1}$ to all the assignment of domain values.

It is easy to verify that each constant has an empty support. More generally we can prove the following:

Proposition 8.2.1. *The support of a constraint $c \Downarrow_I$ is always a subset of I (that is $supp(c \Downarrow_I) \subseteq I$).*

Proof. By definition of $\Downarrow_I$, for any variable $x \notin I$ we have $c \Downarrow_I \eta[x = a] = c \Downarrow_I \eta[x = b]$ for any a and b. So, by definition of support $x \notin supp(c \Downarrow_I)$.

Theorem 8.2.1 (Higher order semiring). *The structure $S_C = \langle \mathcal{C}, \oplus, \otimes, \mathbf{0}, \mathbf{1} \rangle$ where*

- *$\mathcal{C} : (V \to D) \to \mathcal{A}$ is the set of all the possible constraints that can be built starting from S, D and V as defined in Definition 8.2.1,*
- *$\otimes$ and $\oplus$ are the functions defined in Definition 8.2.4, and*
- *$\mathbf{0}$ and $\mathbf{1}$ are constant functions defined following Definition 8.2.5,*

is a c-semiring.

Proof. To prove the theorem it is enough to check all the properties with the fact that the same properties hold for semiring S. We give here only a hint, by showing the commutativity of the $\otimes$ operator:
$c_1 \otimes c_2\eta =$ (by definition of $\otimes$)
$c_1\eta \times c_2\eta =$ (by commutativity of $\times$)
$c_2\eta \times c_1\eta =$ (by definition of $\otimes$)
$c_2 \otimes c_1\eta$.
All the other properties can be proved similarly.

The next step is to look for a notion of token and of entailment relation. We define as tokens the functional constraints in $\mathcal{C}$ and we introduce a relation $\vdash$ that is an entailment relation when the multiplicative operator of the semiring is idempotent.

Definition 8.2.6 ($\vdash$ relation). *Consider the higher order semiring carrier set $\mathcal{C}$ and the partial order $\sqsubseteq$. We define the relation $\vdash \subseteq \wp(\mathcal{C}) \times \mathcal{C}$ s.t. for each $C \in \wp(\mathcal{C})$ and $c \in \mathcal{C}$, we have $C \vdash c \iff \bigotimes C \sqsubseteq c$.*

The next theorem shows that, when the multiplicative operator of the semiring is idempotent, the $\vdash$ relation satisfies all the properties needed by an entailment.

Theorem 8.2.2 ($\vdash$, with idempotent $\times$, is an entailment relation). *Consider the higher order semiring carrier set $\mathcal{C}$ and the partial order $\sqsubseteq$. Consider also the relation $\vdash$ of Definition 8.2.6. Then, if the multiplicative operation of the semiring is idempotent, $\vdash$ is an entailment relation.*

Proof. Is enough to check that for any $c \in \mathcal{C}$, and for any C_1, C_2 and C_3 subsets of $\mathcal{C}$ we have

1. $C \vdash c$ when $c \in C$: We need to show that $\bigotimes C \sqsubseteq c$ when $c \in C$. This follows from the intensivity of $\times$.
2. **if** $C_1 \vdash C_2$ **and** $C_2 \vdash C_3$ **then** $C_1 \vdash C_3$: To prove this we use the extended version of the relation $\vdash$ able to deal with subsets of $\mathcal{C} : \wp(\mathcal{C}) \times \wp(\mathcal{C})$ s.t. $C_1 \vdash C_2 \iff C_1 \vdash \bigotimes C_2$. Note that when $\times$ is idempotent we have that, $\forall c_2 \in C_2,\ C_1 \vdash c_2 \iff C_1 \vdash \bigotimes C_2$. In this case to prove the item we have to prove that if $\bigotimes C_1 \sqsubseteq \bigotimes C_2$ and $\bigotimes C_2 \sqsubseteq \bigotimes C_3$, then $\bigotimes C_1 \sqsubseteq \bigotimes C_3$. This comes from the transitivity of $\sqsubseteq$.

Note that in this setting the notion of token (constraint) and of set of tokens (set of constraints) closed under entailment is used indifferently. In fact, given a set of constraint functions C_1, its closure w.r.t. entailment is a set $\bar{C_1}$ that contains all the constraints greater than $\bigotimes C_1$. This set is univocally representable by the constraint function $\bigotimes C_1$.

The definition of the entailment operator $\vdash$ on top of the higher order semiring $S_C = \langle \mathcal{C}, \oplus, \otimes, \mathbf{0}, \mathbf{1} \rangle$ and of the $\sqsubseteq$ relation leads to the notion of *soft constraint system*. It is also important to notice that in [171] it is claimed that a constraint system is a *complete algebraic* lattice. Here we do not ask for this, since the algebraic nature of the structure $\mathcal{C}$ strictly depends on the properties of the semiring.

If the constraint system is defined on top of a non-idempotent multiplicative operator, we cannot obtain a $\vdash$ relation satisfying all the properties of an entailment. Nevertheless, we can give a *denotational* semantics to the constraint store, as described in Section 8.3, using the operations of the higher order semiring.

To treat the hiding operator of the language, a general notion of existential quantifier has to be introduced by using notions similar to those used in cylindric algebras. Note however that cylindric algebras are first of all boolean algebras. This could be possible in our framework only when the $\times$ operator is idempotent.

Definition 8.2.7 (hiding). *Consider a set of variables V with domain D and the corresponding soft constraint system $\mathcal{C}$. We define for each $x \in V$ the hiding function $(\exists_x c)\eta = \sum_{d_i \in D} c\eta[x := d_i]$.*

To make the hiding operator computationally tractable, we require that the number of domain elements in D having semiring value different from $\mathbf{0}$ is finite. In this way, the sum needed to compute $(\exists_x c)\eta$ in the above definition can consider just a finite number of element (those different from $\mathbf{0}$) since $\mathbf{0}$ is the unit element of the sum.

By using the hiding function we can represent the $\Downarrow$ operator defined in Section 2.2.

Proposition 8.2.2. *Consider a semiring $S = \langle \mathcal{A}, +, \times, \mathbf{0}, \mathbf{1} \rangle$, a domain of the variables D, an ordered set of variables V, the corresponding structure $\mathcal{C}$ and the class of hiding functions $\exists_x : \mathcal{C} \to \mathcal{C}$ as defined in Definition 8.2.7. Then, for any constraint c and any variable $x \subseteq V$, $c \Downarrow_{V-x} = \exists_x c$.*

Proof. Is enough to apply the definition of $\Downarrow_{V-x}$ and $\exists_x$ and check that both are equal to $\sum_{d_i \in D} c\eta[x := d_i]$.

Notice that by the previous theorem x does not belong to the support of $\exists_x c$.

We now show how the hiding function so defined satisfies the properties of cylindric algebras.

Theorem 8.2.3. *Consider a semiring $S = \langle \mathcal{A}, +, \times, \mathbf{0}, \mathbf{1} \rangle$, a domain of the variables D, an ordered set of variables V, the corresponding structure $\mathcal{C}$ and the class of hiding functions $\exists_x : \mathcal{C} \to \mathcal{C}$ as defined in Definition 8.2.7. Then $\mathcal{C}$ is a cylindric algebra satisfying:*

1. $c \vdash \exists_x c$
2. $c_1 \vdash c_2$ *implies* $\exists_x c_1 \vdash \exists_x c_2$
3. $\exists_x(c_1 \otimes \exists_x c_2) \approx \exists_x c_1 \otimes \exists_x c_2$,
4. $\exists_x \exists_y c \approx \exists_y \exists_x c$

Proof. Let us consider all the items:

1. It follows from the intensivity of $+$;
2. It follows from the monotonicity of $+$;
3. $\exists_x(c_1 \otimes \exists_x c_2) =$
   $(c_1 \otimes \exists_x c_2) \Downarrow_{V-x} =$
   $(c_1 \otimes c_2 \Downarrow_{V-x}) \Downarrow_{V-x}$ (since $con(c_2 \Downarrow_{V-x}) = V - x$, and $V - x \cap x = \emptyset$, from Theorem 19 of [47] this is equivalent to)
   $c_1 \Downarrow_{V-x} \otimes c_2 \Downarrow_{V-x} =$
   $\exists_x c_1 \otimes \exists_x c_2$;
4. It follows from commutativity and associativity of $+$.

To model parameter passing we need also to define what diagonal elements are.

Definition 8.2.8 (diagonal elements). *Consider an ordered set of variables V and the corresponding soft constraint system $\mathcal{C}$. Let us define for each $x, y \in V$ a constraint $d_{xy} \in \mathcal{C}$ s.t., $d_{xy}\eta[x := a, y := b] = \mathbf{1}$ if $a = b$ and $d_{xy}\eta[x := a, y := b] = \mathbf{0}$ if $a \neq b$. Notice that $supp(d_{xy}) = \{x, y\}$.*

We can prove that the constraints just defined are diagonal elements.

Theorem 8.2.4. *Consider a semiring $S = \langle \mathcal{A}, +, \times, \mathbf{0}, \mathbf{1} \rangle$, a domain of the variables D, an ordered set of variables V, and the corresponding structure $\mathcal{C}$. The constraints d_{xy} defined in Definition 8.2.8 represent diagonal elements, that is*

1. $d_{xx} = \mathbf{1}$,
2. *if* $z \neq x, y$ *then* $d_{xy} = \exists_z(d_{xz} \otimes d_{zy})$,
3. *if* $x \neq y$ *then* $d_{xy} \otimes \exists_x(c \otimes d_{xy}) \vdash c$.

Proof. 1. It follows from the definition of the **1** constant and of the diagonal constraint;

2. The constraint $d_{xz} \otimes d_{zy}$ is equal to $\mathbf{1}$ when $x = y = z$, and is equal to $\mathbf{0}$ in all the other cases. If we project this constraint over z, we obtain the constraint $\exists_z(d_{xz} \otimes d_{zy})$ that is equal to $\mathbf{1}$ only when $x = y$;
3. The constraint $(c \otimes d_{xy})\eta$ has value $\mathbf{0}$ whenever $\eta(x) \neq \eta(y)$ and $c\eta$ elsewhere. Now, $(\exists_x(c \otimes d_{xy}))\eta$ is by definition equal to $c\eta[x := y]$. Thus $(d_{xy} \otimes \exists_x(c \otimes d_{xy}))\eta$ is equal to $c\eta$ when $\eta(x) = \eta(y)$ and $\mathbf{0}$ elsewhere. So, since for any c, $\mathbf{0} \vdash c$ and $c \vdash c$, we easily have the claim of the theorem.

Using cc on Top of a Soft Constraint System. When using a soft constraint system in a cc language, the notion of consistency should be generalised. In fact, SCSPs with best level of consistency equal to $\mathbf{0}$ can be interpreted as inconsistent, and those with level greater than $\mathbf{0}$ as consistent, but we can also be more general: we can define a suitable function α that, given the best level of the current store, maps such a level over the classical notion of consistency/inconsistency.

More precisely, given a semiring $S = \langle \mathcal{A}, +, \times, \mathbf{0}, \mathbf{1} \rangle$, we can define a function $\alpha : \mathcal{A} \rightarrow \{false, true\}$. Function α has to be at least monotone, but functions with a richer set of properties could be used. Whenever we need to check the consistency of the store, we will first compute the best level and then we will map such a value by using function α over *true* or $false$.

It is important to notice that changing the α function (that is, by mapping in a different way the set of values $\mathcal{A}$ over the boolean elements *true* and $false$), the same cc agent yields different results: by using a high cut level, the cc agent will either finish with a failure or succeed with a high final best level of consistency of the store. On the other hand, by using a low level, more programs will end in a success state.

8.3 Soft Concurrent Constraint Programming

The next step in our work is now to extend the syntax of the language in order to directly handle the cut level. This means that the syntax and semantics of the tell and ask agents have to be enriched with a threshold to specify when tell/ask agents have to fail, succeed or suspend.

Given a soft constraint system $\langle S, D, V \rangle$, the corresponding structure $\mathcal{C}$, and any constraint $\phi \in \mathcal{C}$, the syntax of agents in soft concurrent constraint programming is given in Table 8.2. The main difference w.r.t. the original *cc* syntax is

$$
\begin{aligned}
&P ::= F.A \\
&F ::= p(X) :: A \mid F.F \\
&A ::= \mathit{success} \mid \mathit{fail} \mid \mathit{tell}(c) \rightarrow_{\phi} A \mid \mathit{tell}(c) \rightarrow^{a} A \mid E \mid A \| A \mid \exists X.A \mid p(X) \\
&E ::= \mathit{ask}(c) \rightarrow_{\phi} A \mid \mathit{ask}(c) \rightarrow^{a} A \mid E + E
\end{aligned}
$$

Table 8.2. scc syntax

the presence of a semiring element a and of a constraint ϕ to be checked whenever an *ask* or *tell* operation is performed. More precisely, the level a (resp., ϕ) will be used as a cut level to prune computations that are not good enough.

Notice also the we add the $fail$ state but the failure Semantics will be considered in Section 8.5.2.

We present here a structured operational semantics for scc programs, in the SOS style, which consists of defining the semantics of the programming language by specifying a set of *configurations* Γ, which define the states during execution, a relation $\rightarrow \subseteq \Gamma \times \Gamma$ which describes the *transition* relation between the configurations, and a set T of *terminal* configurations. To give an operational semantics to our language, we need to describe an appropriate transition system.

Definition 8.3.1 (transition system). *A transition system is a triple $\langle \Gamma, T, \rightarrow \rangle$ where Γ is a set of possible configurations, $T \subseteq \Gamma$ is the set of* terminal *configurations and $\rightarrow \subseteq \Gamma \times \Gamma$ is a binary relation between configurations.*

The set of configurations represent the evolutions of the agents and the modifications in the constraint store. We define the transition system of soft cc as follows:

Definition 8.3.2 (configurations). *The set of configurations for a soft cc system is the set $\Gamma = \{\langle A, \sigma \rangle\}\}$, where $\sigma \in \mathcal{C}$. The set of terminal configurations is the set $T = \{\langle success, \sigma \rangle\}$ and the transition rule for the scc language are defined in Table 8.3.*

Here is a brief description of the transition rules:

Valued-tell The valued-tell rule checks for the α-consistency of the SCSP defined by the store $\sigma \otimes c$. The rule can be applied only if the store $\sigma \otimes c$ is b-consistent with $b \not< a$[2]. In this case the agent evolves to the new agent A over the store $\sigma \otimes c$. Note that different choices of the *cut level* a could possibly lead to different computations.

Tell The tell action is a finer check of the store. In this case, a pointwise comparison between the store $\sigma \otimes c$ and the constraint ϕ is performed. The idea is to perform an overall check of the store and to continue the computation only if there is the possibility to compute a solution not worse than ϕ. Notice that this notion of tell could be also applied to the classical cc framework. In this case the tell operation would succeed when the set of tuples satisfying constraint ϕ is a subset of the set of tuples allowed by $\sigma \cap c$.[3]

Valued-ask The semantics of the valued-ask is extended in a way similar to what we have done for the valued-tell action. This means that, to apply the rule, we need to check if the store σ entails the constraint c and also if the store is "consistent enough" w.r.t. the threshold a set by the programmer.

[2] Notice that we use $b \not< a$ instead of $b \geq a$ because we can possibly deal with partial orders. The same happens also in other transition rules with $\not\sqsubset$ instead of $\sqsupseteq$.

[3] notice that the $\otimes$ operator in the crisp case reduces to set intersection.

$$\frac{(\sigma \otimes c) \Downarrow_{\emptyset} \not< a}{\langle \mathit{tell}(c) \rightarrow^{a} A, \sigma\rangle \longrightarrow \langle A, \sigma \otimes c\rangle} \qquad \text{(Valued-tell)}$$

$$\frac{\sigma \otimes c \not\sqsubseteq \phi}{\langle \mathit{tell}(c) \rightarrow_{\phi} A, \sigma\rangle \longrightarrow \langle A, \sigma \otimes c\rangle} \qquad \text{(Tell)}$$

$$\frac{\sigma \vdash c, \sigma \Downarrow_{\emptyset} \not< a}{\langle \mathit{ask}(c) \rightarrow^{a} A, \sigma\rangle \longrightarrow \langle A, \sigma\rangle} \qquad \text{(Valued-ask)}$$

$$\frac{\sigma \vdash c, \sigma \not\sqsubseteq \phi}{\langle \mathit{ask}(c) \rightarrow_{\phi} A, \sigma\rangle \longrightarrow \langle A, \sigma\rangle} \qquad \text{(Ask)}$$

$$\frac{\langle A_1, \sigma\rangle \longrightarrow \langle A_1', \sigma'\rangle}{\begin{array}{l}\langle A_1 \| A_2, \sigma\rangle \longrightarrow \langle A_1' \| A_2, \sigma'\rangle \\ \langle A_2 \| A_1, \sigma\rangle \longrightarrow \langle A_2 \| A_1', \sigma'\rangle\end{array}} \qquad \frac{\langle A_1, \sigma\rangle \longrightarrow \langle \mathit{success}, \sigma'\rangle}{\begin{array}{l}\langle A_1 \| A_2, \sigma\rangle \longrightarrow \langle A_2, \sigma'\rangle \\ \langle A_2 \| A_1, \sigma\rangle \longrightarrow \langle A_2, \sigma'\rangle\end{array}} \qquad \text{(Parallelism)}$$

$$\frac{\langle E_1, \sigma\rangle \longrightarrow \langle A_1, \sigma'\rangle}{\begin{array}{l}\langle E_1 + E_2, \sigma\rangle \longrightarrow \langle A_1, \sigma'\rangle \\ \langle E_2 + E_1, \sigma\rangle \longrightarrow \langle A_1, \sigma'\rangle\end{array}} \qquad \text{(Nondeterminism)}$$

$$\frac{\langle A[y/x], \sigma\rangle \longrightarrow \langle A', \sigma'\rangle}{\langle \exists_x A, \sigma\rangle \longrightarrow \langle A', \sigma'\rangle} \text{ with } y \textit{ fresh} \qquad \text{(Hidden variables)}$$

$$\langle p(y), \sigma\rangle \longrightarrow \langle A[y/x], \sigma\rangle \text{ when } p(x) :: A \qquad \text{(Procedure call)}$$

Table 8.3. Transition rules for scc

Ask Similar to the *tell* rule, here a finer (pointwise) threshold ϕ is compared to the store σ. Notice that we need to check $\sigma \not\sqsubseteq \phi$ because previous tell could have a different threshold ϕ' and could not guarantee the consistency of the resultant store.

Nondeterminism and parallelism The composition operators $+$ and $\|$ are not modified w.r.t. the classical ones: a parallel agent will succeed if all the agents succeeds; a nondeterministic rule chooses any agent whose guard succeeds.

Hidden variables The semantics of the existential quantifier is similar to that described in [171] by using the notion of *freshness* of the new variable added to the store.

Procedure calls The semantics of the procedure call is not modified w.r.t. the classical one. Also here we use notion of diagonal constraint (as defined in Definition 8.2.8) to represent parameter passing.

Eventual Tell/Ask. We recall that both ask and tell operations in cc could be either atomic (that is, if the corresponding check is not satisfied, the agent does not evolve) or eventual (that is, the agent evolves regardless of the result of the check). It is interesting to notice that the transition rules defined in Table 8.3 could be used to provide both interpretations of the ask and tell operations. In fact, while the generic tell/ask rule represents an atomic behaviour, by setting

$\phi = \mathbf{0}$ or $a = \mathbf{0}$ we obtain their *eventual* version:

$$\langle tell(c) \rightarrow A, \sigma \rangle \longrightarrow \langle A, \sigma \otimes c \rangle \qquad \text{(Eventual tell)}$$

$$\frac{\sigma \vdash c}{\langle ask(c) \rightarrow A, \sigma \rangle \longrightarrow \langle A, \sigma \rangle} \qquad \text{(Eventual ask)}$$

Notice that, by using an eventual interpretation, the transition rules of the scc become the same as those of cc (with an eventual interpretation too). This happens since, in the eventual version, the tell/ask agent never checks for consistency and so the soft notion of α-consistency does not play any role.

8.4 A Simple Example

In this section we will show the behaviour of some of the rules of our transition system. We consider in this example a soft constraint system over the fuzzy semiring. Consider the fuzzy constraints

$$c : \{x, y\} \rightarrow \mathcal{R}^2 \rightarrow [0,1] \qquad \text{s.t. } c(x,y) = \frac{1}{1 + |x - y|} \qquad \text{and}$$

$$c' : \{x\} \rightarrow \mathcal{R} \rightarrow [0,1] \qquad \text{s.t. } c'(x) = \begin{cases} 1 & \text{if } x \leq 10, \\ 0 & \text{otherwise.} \end{cases}$$

Notice that the domain of both variables x and y is in this example any integer (or real) number. As any fuzzy CSP, the codomain of the constraints is instead in the interval $[0, 1]$.

Let's now evaluate the agent

$$\langle tell(c) \rightarrow^{0.4} ask(c') \rightarrow^{0.8} success, 1 \rangle$$

in the empty starting store 1. Note that also here the empty store 1 is just the store containing the constraint $\emptyset \rightarrow \emptyset \rightarrow 1$ with empty support and that assign always the semiring level 1 to any assignment.

By applying the *Valued-tell* rule we need to check $(1 \otimes c) \Downarrow_\emptyset \not< 0.4$. Since $1 \otimes c = c$ and $c \Downarrow_\emptyset = 1$, the agent can perform the step, and it reaches the state

$$\langle ask(c') \rightarrow^{0.8} success, c \rangle.$$

Now we need to check (by following the rule of *Valued-ask*) if $c \vdash c'$ and $c \Downarrow_\emptyset \not< 0.8$. While the second relation easily holds, the first one does not hold (in fact, for $x = 11$ and $y = 10$ we have $c'(x) = 0$ and $c(x, y) = 0.5$).

If instead we consider the constraint $c''(x, y) = \frac{1}{1+2\times|x-y|}$ in place of c', then we have

$$\langle ask(c'') \rightarrow^{0.8} success, c \rangle.$$

Here the condition $c \vdash c$" easily holds and the agent $ask(c'') \rightarrow^{0.8}$ *success* can perform its last step, reaching the *success* state:

$$\langle success, c \otimes c'' \rangle.$$

8.5 Observables and Cuts

Sometimes one could desire to see an agent, and a corresponding program, execute with a cut level which is different from the one originally given. We will therefore define $cut_\psi(A)$ the agent A where all the occurrences of any cut level, say ϕ, in any subagent of A or in any clause of the program, are replaced by ψ if $\phi \sqsubseteq \psi$. This means that the cut level of each subagent and clause becomes at least ψ, or is left to the original level. Informally, using the cut ψ we want to obtain (if possible) a solution not lower than ψ, so, all the ask/tell check have to be increased in order to cut away computation with a store not better than ψ.

In this chapter, for simplicity reasons, this cut level change applies only to those programs with cut levels which are constraints (ϕ), and not single semiring levels (a). A similar semantics treatment could be developed also for the other kind of cut, by suitably changing the notion of observable (see Section 8.5.1).

Definition 8.5.1 (cut function). *Consider an scc agent A; we define the function $cut_\psi : A \rightarrow A$ that transforms ask and tell subagents as follows:*

$$cut_\psi(ask/tell(c) \rightarrow_\phi) = \begin{cases} ask/tell(c) \rightarrow_\psi & \text{if } \phi \sqsubset \psi, \\ ask/tell(c) \rightarrow_\phi & \text{otherwise.} \end{cases}$$

By definition of cut_ψ, it is easy to see that $cut_{\mathbf{0}}(A) = A$.

We can then prove the following Lemma (that will be useful later):

Lemma 8.5.1 (tell and ask cut). *Consider the Tell and Ask rules of Table 8.3, and the constraints σ and c as defined in such rules. Then:*

- *If the* Tell *rule can be applied to agent A, then the rule can be applied also to $cut_\psi(A)$ when $\psi \sqsubseteq \sigma \otimes c$.*
- *If the* Ask *rule can be applied to agent A, then the rule can be applied also to $cut_\psi(A)$ when $\psi \sqsubseteq \sigma$.*

Proof. We will prove only the first item; the second can be easily proved by using the same ideas. By the definition of the tell transition rules of Table 8.3, if we can apply the rule it means that $A ::= tell(c) \rightarrow_\phi A'$ and if σ is the store we have $\sigma \otimes c \not\sqsubseteq \phi$. Now, by definition of cut_ψ, we can have

- $cut_\psi(A) ::= tell(c) \rightarrow_\psi cut_\psi(A')$ when $\phi \sqsubset \psi$.
- $cut_\psi(A) ::= tell(c) \rightarrow_\phi cut_\psi(A')$ when $\phi \not\sqsubset \psi$,

In the first case, by hypothesis we have $\sigma \otimes c \not\sqsubseteq \phi$, which together with $\phi \sqsubset \psi$ implies $\sigma \otimes c \not\sqsubseteq \phi$, which is the required condition for the application of the Tell rule. In the second case, the statement directly holds by the hypothesis over A that $\sigma \otimes c \not\sqsubseteq \phi$.

It is now interesting to notice that the thresholds appearing in the program are related to the final computed stores:

Theorem 8.5.1 (thresholds). *Consider an scc computation*

$$\langle A, \mathbf{1} \rangle \rightarrow \langle A_1, \sigma_1 \rangle \rightarrow \ldots \langle A_n, \sigma_n \rangle \rightarrow \langle success, \sigma \rangle$$

for a program P. Then, also

$$\langle cut_\sigma(A), \mathbf{1} \rangle \rightarrow \langle cut_\sigma(A_1), \sigma_1 \rangle \rightarrow \ldots \langle cut_\sigma(A_n), \sigma_n \rangle \rightarrow \langle success, \sigma \rangle$$

is an scc computation for program P.

Proof. First of all, notice that during the computation an agent can only add constraints to the store. So, since $\times$ is intensive, the store can only monotonically decrease starting from the initial store $\mathbf{1}$ and ending in the final store σ. So we have

$$\mathbf{1} \sqsupseteq \sigma_1 \ldots \sqsupseteq \sigma_n \sqsupseteq \sigma.$$

Now, the statement follows by applying at each step the results of Lemma 8.5.1. In fact, at each step the hypothesis of the lemma hold:

- the cut σ is always lower than the current store ($\sigma \sqsubseteq \sigma_i \otimes c$);
- the ask and tell operations can be applied (moving from agent A_i to agent $A_i + 1$).

8.5.1 Capturing Success Computations

Given the transition system as defined in the previous section, we now define what we want to observe of the program behaviour as described by the transitions. To do this, we define for each agent A the set of constraints

$$\mathcal{S}_A = \{\sigma \Downarrow_{var(A)} \mid \langle A, \mathbf{1} \rangle \rightarrow^* \langle success, \sigma \rangle\}$$

that collects the results of the successful computations that the agent can perform. Notice that the computed store σ is projected over the variables of the agent A to discard any *fresh* variable introduced in the store by the $\exists$ operator.

The observable $\mathcal{S}_A$ could be refined by considering, instead of the set of successful computations starting from $\langle A, \mathbf{1} \rangle$, only a subset of them. For example, one could be interested in considering only the *best* computations: in this case, all the computations leading to a store worse than one already collected are disregarded. With a pessimistic view, the representative subset could instead collect all the worst computations (that is, all the computations better than others are disregarded). Finally, also a set containing both the best and the worst computations could be considered. These options are reminiscent of Hoare, Smith and Egli-Milner powerdomains respectively [162].

At this stage, the difference between don't know and don't care nondeterminism arises only in the way the *observables* are interpreted: in a don't care approach, agent A can commit to *one* of the final stores $\sigma \Downarrow_{var(A)}$, while, in a don't know approach, in classical cc programming it is enough that one of the final stores is consistent. Since existential quantification corresponds to the sum in our semiring-based approach, for us a don't know approach leads to the sum (that is, the lub) of all final stores:

$$\mathcal{S}_A^{dk} = \bigoplus_{\sigma \in \mathcal{S}_A} \sigma.$$

It is now interesting to notice that the thresholds appearing in the program are related also to the observable sets:

Proposition 8.5.1 (Thresholds and $\mathcal{S}_A$ (1)). *For each ψ, we have $\mathcal{S}_A \supseteq \mathcal{S}_{cut_\psi(A)}$.*

Proof. By definition of cuts (Definition 8.5.1), we can modify the agents only by changing the thresholds with a new level, greater than the previous one. So, easily, we can only cut away some computations.

Corollary 8.5.1 (Thresholds and $\mathcal{S}_A^{dk}$ (1)). *For each ψ, we have $\mathcal{S}_A^{dk} \supseteq \mathcal{S}_{cut_\psi(A)}^{dk}$.*

Proof. It follows from the definition of $\mathcal{S}_A^{dk}$ and from Proposition 8.5.1.

Theorem 8.5.2 (Thresholds and $\mathcal{S}_A$ (2)). *Let $\psi \sqsubseteq glb\{\sigma \in \mathcal{S}_A\}$. Then $\mathcal{S}_A = \mathcal{S}_{cut_\psi(A)}$.*

Proof. By Proposition 8.5.1, we have $\mathcal{S}_A \supseteq \mathcal{S}_{cut_\psi(A)}$. Moreover, since ψ is lower than all σ in $\mathcal{S}_A$, by Theorem 8.5.1 we have that all the computations are also in $\mathcal{S}_{cut_\psi(A)}$. So, the statement follows.

Notice that, thanks to Theorem 8.5.2 and to Proposition 8.5.1, whenever we have a lower bound ψ of the glb of the final solutions, we can use ψ as a threshold to eliminate some computations. Moreover, we can prove the following theorem:

Theorem 8.5.3. *Let $\sigma \in \mathcal{S}_A$ and $\sigma \notin \mathcal{S}_{cut_\psi(A)}$. Then we have $\sigma \sqsubset \psi$.*

Proof. If $\sigma \in \mathcal{S}_A$ and $\sigma \notin \mathcal{S}_{cut_\psi(A)}$, it means that the cut eliminates some computations. So, at some step we have changed the threshold of some tell or ask agent. In particular, since we know by Theorem 8.5.1 that when $\psi \sqsubseteq \sigma$ we do not modify the computation, we need $\psi \not\sqsubseteq \sigma$. Moreover, since the tell and ask rules fail only if $\sigma \sqsubset \psi$, we easily have the statement of the theorem.

The following theorem relates thresholds and $\mathcal{S}_A^{dk}$.

Theorem 8.5.4 (Thresholds and $\mathcal{S}_A^{dk}$ (2)). *Let $\Psi_A = \{\sigma \in \mathcal{S}_A \mid \nexists \sigma' \in \mathcal{S}_A$ with $\sigma' \sqsupset \sigma\}$ (that is, Ψ_A is the set of "greatest" elements of $\mathcal{S}_A$). Let also $\psi \sqsubseteq glb\{\sigma \in \Psi_A\}$. Then $\mathcal{S}_A^{dk} = \mathcal{S}_{cut_\psi(A)}^{dk}$.*

Proof. Since we have $a+b=b \iff a \leq b$, we easily have $\bigoplus_{\sigma \in \mathcal{S}_A} \sigma = \bigoplus_{\sigma \in \Psi_A} \sigma$. Now, by following a reasoning similar to Theorem 8.5.2, by applying a cut with a threshold $\psi \sqsubseteq glb\{\sigma \in \Psi_A\}$ we do not eliminate any computation. So we obtain $\mathcal{S}_A^{dk} = \bigoplus_{\sigma \in \mathcal{S}_A} \sigma = \bigoplus_{\sigma \in \Psi_A} \sigma = \mathcal{S}_{cut_\psi(A)}^{dk}$

Lemma 8.5.2. *Given any constraint ψ, we have:*

$$\mathcal{S}^{dk}_{A} \sqsubseteq \psi \oplus \mathcal{S}^{dk}_{cut_\psi(A)}.$$

Proof. Let S be the set of all solutions; then $\mathcal{S}^{dk}_{A} = lub(S)$ and $\mathcal{S}^{dk}_{cut_\psi(A)} = lub(S_1)$ where $S_1 \subseteq S$. The solutions that have been eliminated by the cut ψ (that is all the $\sigma \in S - S_1$) are all lower than ψ by Theorem 8.5.3. So, it easily follows that $\mathcal{S}^{dk}_{A} \sqsubseteq \psi \oplus \mathcal{S}^{dk}_{cut_\psi(A)}$.

Theorem 8.5.5. *Given any constraint ψ, we have:*

$$\mathcal{S}^{dk}_{cut_\psi(A)} \sqsubseteq \mathcal{S}^{dk}_{A} \sqsubseteq \psi \oplus \mathcal{S}^{dk}_{cut_\psi(A)}.$$

Proof. From Corollary 8.5.1, we have $\mathcal{S}^{dk}_{cut_\psi(A)} \sqsubseteq \mathcal{S}^{dk}_{A}$. From Lemma 8.5.2 we have instead $\mathcal{S}^{dk}_{A} \sqsubseteq \psi \oplus \mathcal{S}^{dk}_{cut_\psi(A)}$.

This theorem suggests a way to cut useless computations while generating the observable $\mathcal{S}^{dk}_{A}$ of an scc program P starting from agent A. A very naive way to obtain such an observable would be to first generate all final states, of the form $\langle success, \sigma_i \rangle$, and then compute their lub. An alternative, smarter way to compute this same observable would be to do the following. First we start executing the program as it is, and find a first solution, say σ_1. Then we restart the execution applying the cut level σ_1.

By Theorem 8.5.4, this new cut level cannot eliminate solutions which influence the computation of the observable: the only solutions it will cut are those that are lower than the one we already found, thus useless in terms of the computation of $\mathcal{S}^{dk}_{A}$.

In general, after having found solutions $\sigma_1, \ldots, \sigma_k$, we restart execution with cut level $\psi = \sigma_1 \oplus \ldots \oplus \sigma_k$. Again, this will not cut crucial solutions but only some that are lower than the sum of those already found. When the execution of the program terminates with no solution we can be sure that the cut level just used (which is the sum of all solutions found) is the desired observable (in fact, by Theorem 8.5.5 when $\mathcal{S}^{dk}_{cut_\psi(A)} = \psi$ we necessarily have $\mathcal{S}^{dk}_{cut_\psi(A)} = \mathcal{S}^{dk}_{A} = \psi$).

In a way, such an execution method resembles a branch & bound strategy, where the cut levels have the role of the bounds. Notice also that since classical crisp constraints can be represent in the soft CSP framework using a suitable semiring, all the branch & Bound results could be easily extended also to the original *cc*.

The following corollary is important to show the correctness of this approach.

Corollary 8.5.2. *Given any constraint $\psi \sqsubseteq \mathcal{S}^{dk}_{A}$, we have:*

$$\mathcal{S}^{dk}_{A} = \psi \oplus \mathcal{S}^{dk}_{cut_\psi(A)}.$$

Proof. It easily comes from Theorem 8.5.5.

Let us now use this corollary to prove the correctness of the whole procedure above.

Let σ_1 be the first final state reached by agent A. By stopping the algorithm after one step, what we have to prove is $\mathcal{S}_A^{dk} = \sigma_1 \oplus \mathcal{S}_{cut_{\sigma_1}(A)}^{dk}$. Since σ_1 is for sure lower than $\mathcal{S}_A^{dk}$, this is true by Corollary 8.5.2.

By applying this procedure iteratively, we will collect a superset Ψ'_A of $\Psi_A = \{\sigma \in \mathcal{S}_A \mid \nexists \sigma' \in \mathcal{S}_A \text{ with } \sigma' \sqsupseteq \sigma\}$ (Ψ'_A is a superset of Ψ_A because we could collect a final state σ_i before computing a final state $\sigma_j \sqsupseteq \sigma_i$; in this case both will be in Ψ'_A). Even if Ψ'_A contains more elements than Ψ_A, we have $\bigoplus_{\sigma \in \Psi'_A} = \bigoplus_{\sigma \in \Psi_A}$ (for the extensivity and idempotence properties of $+$).

The only difference with the procedure we have tested correct w.r.t. the algorithm is that, at each step, it performs a cut by using the sum of all the previously computed final state. This means that the algorithm can at each step eliminate more computations, but by the results of Theorem 8.5.3 the eliminated computations do not change the final result.

8.5.2 Failure

The transition system we have defined considers only successful computations. If this could be a reasonable choice in a don't know interpretation of the language it will lead to an insufficient analysis of the behaviour in a *pessimistic* interpretation of the indeterminism. To capture agents' failure, we add the transition rules of Table 8.4 to those of Table 8.3.

$$\frac{\sigma \otimes c \sqsubset \phi}{\langle tell(c) \rightarrow_{\phi} A, \sigma\rangle \longrightarrow fail} \qquad (\text{Tell}_1)$$

$$\frac{(\sigma \otimes c) \Downarrow_{\emptyset} < a}{\langle tell(c) \rightarrow^{a} A, \sigma\rangle \longrightarrow fail} \qquad (\text{Valued-tell}_1)$$

$$\frac{\sigma \sqsubset \phi}{\langle ask(c) \rightarrow_{\phi} A, \sigma\rangle \longrightarrow fail} \qquad (\text{Ask}_1)$$

$$\frac{\sigma \Downarrow_{\emptyset} < a}{\langle ask(c) \rightarrow^{a} A, \sigma\rangle \longrightarrow fail} \qquad (\text{Valued-ask}_1)$$

$$\frac{\langle E_1, \sigma\rangle \longrightarrow fail, \langle E_2, \sigma\rangle \longrightarrow fail}{\begin{array}{c}\langle E_1 + E_2, \sigma\rangle \longrightarrow fail\\ \langle E_2 + E_1, \sigma\rangle \longrightarrow fail\end{array}} \qquad (\text{Nondeterminism}_1)$$

$$\frac{\langle A_1, \sigma\rangle \longrightarrow fail}{\begin{array}{c}\langle A_1 \| A_2, \sigma\rangle \longrightarrow fail\\ \langle A_2 \| A_1, \sigma\rangle \longrightarrow fail\end{array}} \qquad (\text{Parallelism}_1)$$

Table 8.4. Failure in the scc language

(Valued)tell$_1$/ask$_1$ The failing rule for ask and tell simply checks if the added/checked constraint c is *inconsistent* with the store σ and in this case stops the computation and gives *fail* as a result. Note that since we use soft constraints we enriched this operator with a threshold (a or ϕ). This is used also to compute failure. If the level of consistency of the resulting store is lower than the threshold level, then this is considered a failure.

Nondeterminism$_1$ Since the failure of a branch arises only from the failure of a guard, and since we use angelic non-determinism (that is, we check the guards before choosing one path), we fail only when all the branches fail.

Parallelism$_1$ In this case the computation fails as soon as one of the branches ails.

The observables of each agent can now be enlarged by using the function

$$\mathcal{F}_A = \{fail \mid \langle A, \mathbf{1}_V \rangle \rightarrow^* fail\}$$

that computes a failure if at least a computation of agent A fails.

By considering also the failing computations, the difference between don't know and don't care becomes finer. In fact, in situations where we have $\mathcal{S}_A = \mathcal{S}_A^{dk}$, the failing computations could make the difference: in the don't care approach the notion of failure is *existential* and in the don't know one becomes *universal* [81]:

$$\mathcal{F}_A^{dk} = \{fail \mid \text{all computations of } A \text{ lead to } fail\}.$$

This means that in the don't know nondeterminism we are interested in observing a failure only if all the branches fail. In this way, given an agent A with an empty $\mathcal{S}_A^{dk}$ and a non-empty $\mathcal{F}_A^{dk}$, we cannot say for sure that the semantics of this agent is $fail$. In fact, the transition rules we have defined do not consider *hang* and infinite computations. Similar semi-decibility results for soft constraint logic programming are proven in [50].

8.6 An Example from the Network Scenario

We consider in this section a simple network problem, involving a set of processes running on distinct locations and sharing some variables, over which they need to synchronize, and we show how to model and solve such a problem in scc.

Each process is connected to a set of variables, shared with other processes, and it can perform several moves. Each of such moves involves performing an action over some or all the variables connected to the process. An action over a variable consists of giving a certain value to that variable. A special value "idle" models the fact that a process does not perfom any action over a variable. Each process has also the possibility of not moving at all: in this case, all its variables are given the idle value.

The set of possible moves a process can perform is represented by a constraint. The constraint assign to each possible move a semiring element representing the cost of that particular move.

The desired behavior of a network of such processes is that, at each move of the entire network:

1. processes sharing a variable perform the same action over it;
2. all processes try to perform a non-idle move.

To describe a network of processes with these features, we use an SCSP where each variable models a shared variable, and each constraint models a process and connects the variables corresponding to the shared variables of that process. The domain of each variable in this SCSP is the set of all possible actions, including the idle one. Each way of satisfying a constraint is therefore a tuple of actions that a process can perform on the corresponding shared variables.

In this scenario, softness can be introduced both in the domains and in the constraints. In particular, since we prefer to have as many moving processes as possible, we can associate a penalty to both the idle element in the domains, and to tuples containing the idle action in the constraints. As for the other domain elements and constraint tuples, we can assign them suitable preference values to model how much we like that action or that process move.

For example, we can use the semiring $S = \langle [-\infty, 0], max, +, -\infty, 0 \rangle$, where 0 is the best preference level (or, said dually, the weakest penalty), $-\infty$ is the worst level, and preferences (or penalties) are combined by summing them. According to this semiring, we can assign value $-\infty$ to the idle action or move, and suitable other preference levels to the other values and moves.

Figure 8.1 gives the details of a part of a network and it shows eight processes (that is, $c_1, \ldots, c_8$) sharing a total of six variables. In this example, we assume that processes c_1, c_2 and c_3 are located on site a, processes c_5 and c_6 are located on site b, and c_4 is located on site c. Processes c_7 and c_8 are located on site d. Site e connects this part of the network to the rest. Therefore, for example, variables x_d, y_d and z_d are shared between processes located in distinct locations.

As desired, finding the best solution for the SCSP representing the current state of the process network means finding a move for all the processes such that they perform the same action on the shared variables, the overall cost of the moves is minimized, and there is no idle process. However, since the problem is inherently distributed, it does not make sense, and it might not even be possible, to centralize all the information and give it to a single soft constraint solver. On the contrary, it may be more reasonable to use several soft constraint solvers, one for each network location, which will take care of handling only the constraints present in that location. Then, the interaction between processes in different locations, and the necessary agreement to solve the entire problem, will be modelled via the scc framework, where each agent will represent the behaviour of the processes in one location.

More precisely, each scc agent (and underlying soft constraint solver) will be in charge of receiving the necessary information from the other agents (via suitable asks) and using it to achieve the synchronization of the processes in its location. For this protocol to work, that is, for obtaining a global optimal solution without a centralization of the work, the SCSP describing the network of processes has to have a tree-like shape, where each node of the tree contains

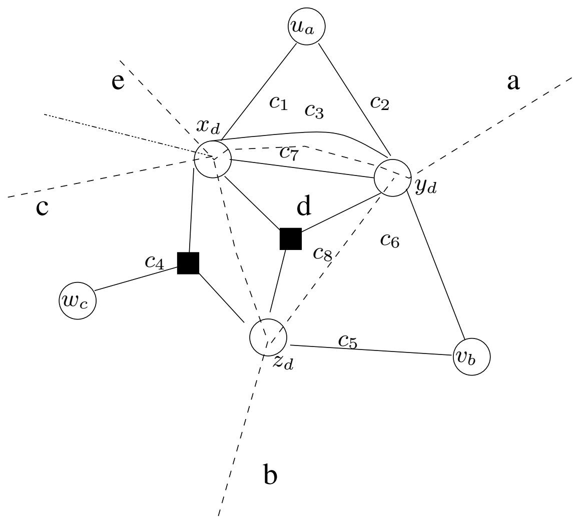
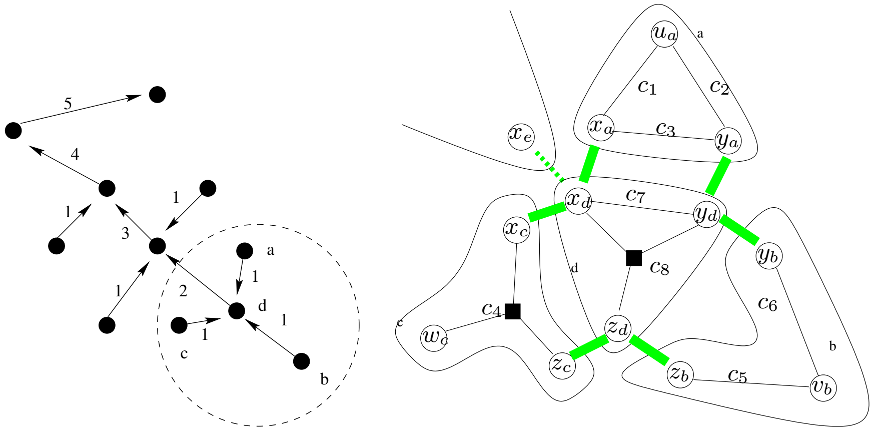


Fig. 8.1. The SCSP describing part of a process network

all the processes in a location, and the agents have to communicate from the bottom of the tree to its root. In fact, the proposed protocol uses a sort of Dynamic Programming technique to distribute the computation between the locations. In this case the use of a tree shape allows us to work, at each step of the algorithm, only locally to one of the locations. In fact, a non tree shape would lead to the construction of non-local constraints and thus require computations which involve more than one location at a time (this save to the system some possible backtracking steps).

In our example, the tree structure we will use is the one shown in Figure 8.2(a), which also shows the direction of the child-parent relation links (via arrows). Figure 8.2(b) describes instead the partition of the SCSP over the four involved locations. The gray connections represent the synchronization to be assured between distinct locations. Notice that, w.r.t. Figure 8.1, we have duplicated the variables representing variables shared between distinct locations, because of our desire to first perform a local work and then to communicate the results to the other locations. It is important to highlight that we do not need to perform any backtracking steps for the synchronization of the several local computations. The computation is performed indipendently in each locations. Only later the resulting constraint stores are combined giving raise to the final store. The final store represent the combination of all the requirement imposed in a distributive fashion over the locations.

The scc agents (one for each location plus the parallel composition of all of them) are therefore defined as follows:



(a) A possible tree structure for our network.

(b) The SCSP partitioned over the four locations.

Fig. 8.2. The ordered process network

$$A_a : \exists_{u_a}(tell(c_1(x_a,u_a) \wedge c_2(u_a,y_a) \wedge c_3(x_a,y_a)) \rightarrow_{\phi_1} tell(end_a = true) \\ \rightarrow success)$$

$$A_b : \exists_{v_b}(tell(c_5(y_b,v_b) \wedge c_6(z_b,v_b)) \rightarrow_{\phi_2} tell(end_b = true) \\ \rightarrow success)$$

$$A_c : \exists_{w_c}(tell(c_4(x_c,w_c,z_c)) \rightarrow_{\phi_3} tell(end_c = true) \\ \rightarrow success)$$

$$A_d : ask(end_a = true \wedge end_b = true \wedge end_c = true \wedge end_d = true) \rightarrow_{\phi} \\ tell(c_7(x_d,y_d) \wedge c_8(x_d,y_d,z_d) \wedge x_a = x_d = x_c \wedge y_a = y_d = y_b \wedge z_b = z_d = z_c) \\ \rightarrow success)$$

$$A : A_a \mid A_b \mid A_c \mid A_d$$

Agents A_a,A_b,A_c and A_d represent the processes running respectively in the location a, b, c and d. Note that, at each ask or tell, the underlying soft constraint solver will only check (for consistency or entailment) a part of the current set of constraints: those local to one location. Due to the tree structure chosen for this example, where agents A_a, A_b, and A_c correspond to leaf locations, only agent A_d shows all the actions of a generic process: first it needs to collect the results computed separately by the other agents (via the ask); then it performs its own constraint solving (via a tell), and finally it can set its end flag, that will be used by a parent agent (in this case the agent corresponding to location e, which we have not modelled here).

The thresholds ϕ_i of the first three agents are used to stop the computation locally if the best way to assign values to the local variables is not good enough (at least ϕ_i). In this way, the synchronization among sites is not perfomed if a local agent discovers that there is no satisfactory scenario. If all local agents pass their threshold check, then the last agent (A_d) can perform the syncronization. However, it can use a threshold as well (ϕ) to avoid the syncronization because of its own satisfactory notion. Notice that the four agents can have different thresholds, since they run on different sites with possibly different policies. For example, different administrative domains over the Internet (which would be represented by sites a,b,c,d in this example) can have different regulations over quality of services.

8.7 Conclusions

We started our work by realizing the need for handling preferences in Web-related scenarios. To address this need, we have defined soft cc, where soft constraints can be used both at the solver level, to make the notion of consistency more tolerant, and at the language level, to provide an explicit way to deal with approximations and satisfaction levels.

9. Interchangeability in Soft CSPs

Overview

Substitutability and interchangeability in constraint satisfaction problems (CSPs) have been used as a basis for search heuristics, solution adaptation and abstraction techniques. In this chapter, we consider how the same concepts can be extended to *soft* constraint satisfaction problems (SCSPs).

We introduce two notions: *threshold* α and *degradation* δ for substitutability and interchangeability, ($_{\alpha}$substitutability/interchangeability and $^{\delta}$substitutability/ interchangeability respectively). We show that they satisfy analogous theorems to the ones already known for hard constraints. In $_{\alpha}$interchangeability, values are interchangeable in any solution that is better than a threshold α, thus allowing to disregard differences among solutions that are not sufficiently good anyway. In $^{\delta}$interchangeability, values are interchangeable if their exchange could not degrade the solution by more than a factor of δ.

We give efficient algorithms to compute ($^{\delta}/_{\alpha}$)interchangeable sets of values for a large class of SCSPs, and show an example of their application.

Substitutability and interchangeability in CSPs have been introduced by Freuder [106] in 1991 with the intention of improving search efficiency for solving CSP.

Interchangeability has since found other applications in abstraction frameworks ([67,106,120,194]) and solution adaptation ([155,195]). One of the difficulties with interchangeability has been that it does not occur very frequently.

In many practical applications, constraints can be violated at a cost, and solving a CSP thus means finding a value assignment of minimum cost. Various frameworks for solving such soft constraints have been proposed [32,47,50,91, 96,108,167,174]. The soft constraints framework of c-semirings [32,47] has been shown to express most of the known variants through different instantiations of its operators, and this is the framework we are considering in this chapter.

The most straightforward generalization of interchangeability to soft CSP would require that exchanging one value for another does not change the quality of the solution at all. This generalization is likely to suffer from the same weaknesses as interchangeability in hard CSP, namely that it is very rare.

Fortunately, soft constraints also allow weaker forms of interchangeability where exchanging values may result in a degradation of solution quality by some measure δ. By allowing more degradation, it is possible to increase the amount of interchangeability in a problem to the desired level. We define $^{\delta}$substitutability/interchangeability as a concept which ensures this quality. This is particularly useful when interchangeability is used for solution adaptation.

Another use of interchangeability is to reduce search complexity by grouping together values that would never give a sufficiently good solution. In $_{\alpha}$substitutability/interchangeability, we consider values interchangeable if they give equal solution quality in all solutions better than α, but possibly different quality for solutions whose quality is $\leq \alpha$.

Just like for hard constraints, full interchangeability is hard to compute, but can be approximated by neighbourhood interchangeability which can be computed efficiently and implies full interchangeability. We define the same concepts for soft constraints, and prove that neighborhood implies full $(^{\delta}/_{\alpha})$substitutability/interchangeability. We give algorithms for neighborhood $(^{\delta}/_{\alpha})$substitutability/interchangeability, and we prove several interesting and useful properties of the concepts.

Finally, we give two examples where $(^{\delta}/_{\alpha})$interchangeability is applied to solution adaptation in configuration problems with two different soft constraint frameworks: delay and cost constraints, and show its usefulness in these practical contexts.

9.1 Interchangeability

Interchangeability in constraint networks was first proposed by Freuder [106] to capture equivalence among values of a variable in a discrete constraint satisfaction problem. Value $v = a$ is *substitutable* for $v = b$ if for any solution where $v = a$, there is an identical solution except that $v = b$. Values $v = a$ and $v = b$ are *interchangeable* if they are substitutable both ways.

Interchangeability offers three important ways for practical applications:

- by pruning the interchangeable values, which are redundant in a sense, the problem space can be simplified.
- by using it as a solution updating tool; interchangeability can be used during user-interaction to help users in taking decisions by offering alternatives;
- by structuring and classifying the solution space.

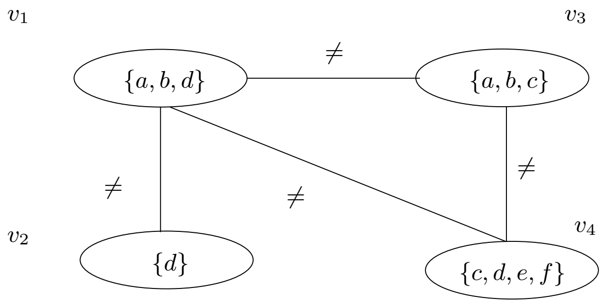


Fig. 9.1. An example of CSP with interchangeable values

Full Interchangeability considers all constraints in the problem and checks if a value a and b for a certain variable v can be interchanged without affecting the global solution. In the CSP in Figure 9.1 (taken from [68]), d, e and f are fully interchangeable for v_4. This is because we inevitably have $v_2 = d$, which implies that v_1 cannot be assigned d in any consistent global solution. Consequently, the values d, e and f can be freely permuted for v_4 in any global solution.

There is no efficient algorithm for computing full Interchangeability, as it may require computing all solutions. The localized notion of *Neighbourhood Interchangeability* considers only the constraints involving a certain variable v. In this notion, a and b are *neighbourhood interchangeable* if for every constraint involving v, for every tuple that admits $v = a$ there is an otherwise identical tuple that admits $v = b$, and vice-versa. In Figure 9.1, e and f are neighbourhood interchangeable for v_4.

Freuder showed that neighbourhood interchangeability always implies full interchangeability and can therefore be used as an approximation. He also provided an efficient algorithm (Algorithm 1) for computing neighborhood interchangeability [106], and investigated its use for preprocessing CSP before searching for solutions [27].

Every node in the discrimination tree (Figure 9.2) corresponds to a set of assignments to variables in the neighbourhood of v that are compatible with some value of v itself. Interchangeable values are found by the fact that they follows the same path and fall into the same ending node.

Algorithm 1. Discrimination Tree for variable v_i

```
Create the root of the discrimination tree for variable $v_i$;
Let $D_{v_i} = \{$the set of domain values $d_{v_i}$ for variable $v_i\}$;
Let $Neigh(\{v_i\}) = \{$all neighborhood variables $v_j$ of variable $v_i\}$;
for all $d_{v_i} \in D_{v_i}$ do
  for all $v_j \in Neigh(\{v_i\})$ do
    for all $d_{v_j} \in D_{v_j}$ s.t. $d_{v_j}$ is consistent with $d_{v_i}$ for $v_i$ do
      if there exists a child node corresponding to $v_j = d_{v_j}$ then
        move to it,
      else
        construct such a node and move to it;
  Add $v_i, \{d_{v_i}\}$ to annotation of the node;
  Go back to the root of the discrimination tree.
```

Figure 9.2 shows an example of execution of Algorithm 1 for variable v_4. Domain values e and f are shown to be interchangeable.

Another form of interchangeability introduced as well in [106] is *Partial Interchangeability*. It allows a subset of the variables V to be affected when interchanging the values of v, while the rest of the variables remains the same. To be more precise, values a and b are *partially interchangeable* with respect to a set of variables V' if for any solution where $v = a$, there is another solution where $v = b$ which otherwise differs only in values assigned to variables in V', and vice

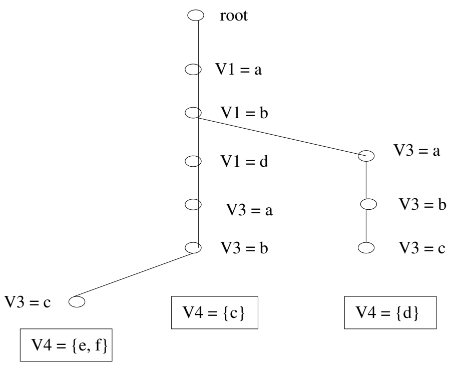


Fig. 9.2. An example of CSP with computation of neighborhood interchangeable values

versa. In Figure 9.1, a and b are partial interchangeable for v_1 with respect to the set $V' = \{v_3\}$.

There is no efficient algorithm for computing partial interchangeability, but a localized form, neighborhood partial interchangeability, was proposed by Choueiry and Noubir in [68]. They says that values a and b are *neighborhood partial interchangeable* with respect to a set of variables V' if for every constraint between a variable in V' and a variable not in V' and every tuple t of value assignments to V' that admits $v = a$ there is another tuple t' that admits $v = b$ such that t and t' are consistent with the same value combinations for variables outside of V'. Additionally, the same condition must hold with a and b exchanged.

Neighborhood Partial Interchangeability is a weak and locally computable form of interchangeability. Locally computable forms of interchangeability may involve sacrificing some solutions but there are no polynomial algorithms till now which can compute full interchangeability and partial interchangeability. A polynomial algorithm for computing neighborhood partial interchangeability sets was proposed by Choueiry and Noubir in [68].

9.2 Interchangeability in Soft CSPs

In soft CSPs, there is not any crisp notion of consistency. In fact, each tuple is a possible solution, but with different level of preference. Therefore, in this framework, the notion of interchangeability becomes finer: to say that values a and b are interchangeable we have also to consider the assigned semiring level.

More precisely, if a domain element a assigned to variable v can be substituted in each tuple solution with a domain element b without obtaining a worse semiring level we say that b is full substitutable for a.

Definition 9.2.1 (Full Substitutability (FS)). *Consider two domain values b and a for a variable v, and the set of constraints C; we say that b is Full*

Substitutable for a on v ($b \in FS_v(a)$) if and only if for all assignments η,

$$\bigotimes C\eta[v := a] \leq_S \bigotimes C\eta[v := b]$$

When we restrict this notion only to the set of constraints C_v that involves variable v we obtain a local version of substitutability.

Definition 9.2.2 (Neighborhood Substitutability (*NS*)). *Consider two domain values b and a for a variable v, and the set of constraints C_v involving v; we say that b is neighborhood substitutable for a on v ($b \in NS_v(a)$) if and only if for all assignments η,*

$$\bigotimes C_v\eta[v := a] \leq_S \bigotimes C_v\eta[v := b]$$

When the relations hold in both directions, we have the notion of Full/Neighborhood interchangeability of b with a.

Definition 9.2.3 (Full and Neighborhood Interchangeability (*FI* and *NI*)). *Consider two domain values b and a, for a variable v, the set of all constraints C and the set of constraints C_v involving v. We say that b is fully interchangeable with a on v ($FI_v(a/b)$) if and only if $b \in FS_v(a)$ and $a \in FS_v(b)$, that is, for all assignments η,*

$$\bigotimes C\eta[v := a] = \bigotimes C\eta[v := b].$$

We say that b is Neighborhood interchangeable with a on v ($NI_v(a/b)$) if and only if $b \in NS_v(a)$ and $a \in NS_v(b)$, that is, for all assignments η,

$$\bigotimes C_v\eta[v := a] = \bigotimes C_v\eta[v := b].$$

This means that when a and b are interchangeable for variable v they can be exchanged without affecting the level of any solution.

Two important results that hold in the crisp case can be proven to be satisfied also with soft CSPs: transitivity and extensivity of interchangeability/substituability.

Theorem 9.2.1 (Extensivity: $NS \implies FS$ and $NI \implies FI$). *Consider two domain values b and a for a variable v, the set of constraints C and the set of constraints C_v involving v. Then, neighborhood (substituability) interchangeability implies full (substituability) interchangeability.*

Proof. By definition of neighborhood substitutability,

$$b \in NS_v(a) \iff \forall\eta, \bigotimes C_v\eta[v := a] \leq_S \bigotimes C_v\eta[v := b].$$

Now, since the assignments $v := a$ and $v := b$ only involve constraints in C_v, and for the extensivity properties of times, we easily have that

$$\forall\eta, \bigotimes C\eta[v := a] \leq_S \bigotimes C\eta[v := b],$$

that is $b \in FS_v(a)$. Easily, we can extend the result to interchangeability.

Theorem 9.2.2 (Transitivity: $b \in NS_v(a), a \in NS_v(c) \implies b \in NS_v(c)$). *Consider three domain values a, b and c, for a variable v. Then,*

$$b \in NS_v(a), a \in NS_v(c) \implies b \in NS_v(c).$$

Similar results hold for FS, NI and FI.

Proof. By definition of neighborhood substitutability,

$$b \in NS_v(a) \iff \forall \eta, \bigotimes C_v\eta[v := a] \leq_S \bigotimes C_v\eta[v := b] \text{ and,}$$
$$a \in NS_v(c) \iff \forall \eta, \bigotimes C_v\eta[v := c] \leq_S \bigotimes C_v\eta[v := a].$$

Now, for transitivity of $\leq_S$, we easily have that

$$\forall \eta, \bigotimes C\eta[v := c] \leq_S \bigotimes C\eta[v := b],$$

that is $b \in NS_v(c)$. Easily, we can extend the result for FS, NI and FI.

As an example of interchangeability and substitutability consider the fuzzy CSP represented in Figure 9.3. The domain value c is neighborhood interchangeable with a on x ($NI_x(a/c)$); in fact, $c_1 \otimes c_2\eta[x := a] = c_1 \otimes c_2\eta[x := c]$ for all η. The domain values c and a are also neighborhood substitutable for b on x ($\{a, c\} \in NS_v(b)$). In fact, for any η we have $c_1 \otimes c_2\eta[x := b] \leq c_1 \otimes c_2\eta[x := c]$ and $c_1 \otimes c_2\eta[x := b] \leq c_1 \otimes c_2\eta[x := a]$.

9.2.1 Degradations and Thresholds

In soft CSPs, any value assignment is a solution, but may have a very bad preference value. This allows broadening the original interchangeability concept to one that also allows degrading the solution quality when values are exchanged. We call this ${}^{\delta}$interchangeability, where δ is the *degradation* factor.

When searching for solutions to soft CSP, it is possible to gain efficiency by not distinguishing values that could in any case not be part of a solution of sufficient quality. In ${}_{\alpha}$interchangeability, two values are interchangeable if they do not affect the quality of any solution with quality better than α. We call α

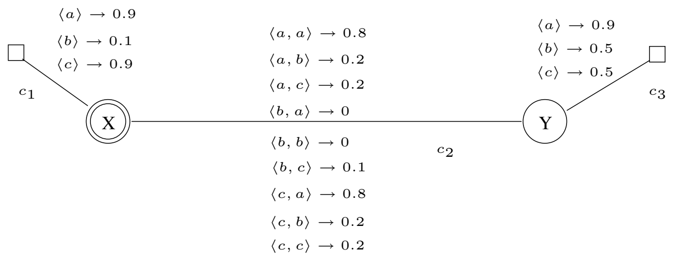

$\langle a \rangle \rightarrow 0.9$
$\langle b \rangle \rightarrow 0.1$
$\langle c \rangle \rightarrow 0.9$
c_1
X
$\langle a, a \rangle \rightarrow 0.8$
$\langle a, b \rangle \rightarrow 0.2$
$\langle a, c \rangle \rightarrow 0.2$
$\langle b, a \rangle \rightarrow 0$
$\langle b, b \rangle \rightarrow 0$
$\langle b, c \rangle \rightarrow 0.1$
$\langle c, a \rangle \rightarrow 0.8$
$\langle c, b \rangle \rightarrow 0.2$
$\langle c, c \rangle \rightarrow 0.2$
c_2
$\langle a \rangle \rightarrow 0.9$
$\langle b \rangle \rightarrow 0.5$
$\langle c \rangle \rightarrow 0.5$
Y
c_3

Fig. 9.3. A fuzzy CSP

the *threshold* factor. Moreover, sometimes we are just looking for *any* solution greater than a certain level α. In this case, also the notion of ${}_{\alpha}$interchangeability could be too strict. For this motivation we define also a more relaxed notion of threshold that we call $\alpha-$set.

Both concepts can be combined, i.e. we can allow both degradation and limit search to solutions better than a certain threshold (${}^{\delta}_{\alpha/\alpha-\mathrm{set}}$interchangeability).

By extending the previous definitions we can define thresholds and degradation version of full/neighbourhood substitutability/interchangeability.

Definition 9.2.4 (${}^{\delta}$Full/Neighbourhood Substitutability (${}^{\delta}FS/NS$)). *Consider two domain values b and a for a variable v, the set of constraints C and a semiring level δ; we say that b is* ${}^{\delta}$fully substitutable *for a on v ($b \in {}^{\delta}FS_v(a)$) if and only if for all assignments η,*

$$\bigotimes C\eta[v := a] \times_S \delta \leq_S \bigotimes C\eta[v := b]$$

It is ${}^{\delta}$neighbourhood substitutable *if the condition holds for C being the subset of the constraints that have v as a variable.*

Definition 9.2.5 (${}_{\alpha}$Full/Neighbourhood Substitutability (${}_{\alpha}FS/NS$)). *Consider two domain values b and a, for a variable v, the set of constraints C and a semiring level α; we say that b is* ${}_{\alpha}$fully substitutable *for a on v ($b \in {}_{\alpha}FS_v(a)$) if and only if for all assignments η,*

$$\bigotimes C\eta[v := a] \geq_S \alpha \implies \bigotimes C\eta[v := a] \leq_S \bigotimes C\eta[v := b]$$

It is ${}_{\alpha}$neighbourhood substitutable *if the condition holds for C being the subset of the constraints that have v as a variable.*

Definition 9.2.6 (${}_{\alpha-\mathrm{set}}$Full/Neighbourhood Substitutability (${}_{\alpha-\mathrm{set}}FS/NS$)). *Consider two domain values b and a, for a variable v, the set of constraints C and a semiring level α; we say that b is* ${}_{\alpha-\mathrm{set}}$full substitutable *for a on v ($b \in {}_{\alpha-\mathrm{set}}FS_v(a)$) if and only if for all assignments η,*

$$\bigotimes C\eta[v := a] \geq_S \alpha \implies \bigotimes C\eta[v := b] \geq_S \alpha$$

It is ${}_{\alpha-\mathrm{set}}$neighbourhood substitutable *if the condition holds for C being the subset of the constraints that have v as a variable.*

Definition 9.2.7 (Full/Neighbourhood Soft Interchangeability). *Consider two domain values b and a, for a variable v, the set of constraints C. Values a and b are*

- ${}^{\delta}$fully/neighbourhood interchangeable *if and only if they are* ${}^{\delta}$fully/neighborhood substitutable *both ways;*
- ${}_{\alpha}$fully/neighbourhood interchangeable *if and only if they are* ${}_{\alpha}$fully/neighborhood substitutable *both ways;*

– ${}_{\alpha-\text{set}}$fully/neighbourhood interchangeable *if and only if they are* ${}_{\alpha-\text{set}}$fully/ neighborhood substitutable *both ways.*

It is easy to see from the definitions that

Theorem 9.2.3 **($\alpha \implies \alpha$–set).** *Consider two domain values a and b, for a variable v, and a thresholds α. Then,*

$$a \in {}_{\alpha}NS_v(b) \implies a \in {}_{\alpha-\text{set}}NS_v(b)$$

Similar results holds for FS, NI, FI.

Proof. By definition of α and α–set substitutability,

$$\begin{aligned} b \in {}_{\alpha}FS_v(a) &\iff \\ \forall \eta, \bigotimes C\eta[v := a] \geq_S \alpha &\implies \bigotimes C\eta[v := a] \leq_S \bigotimes C\eta[v := b], \text{ and,} \\ b \in {}_{\alpha-\text{set}}FS_v(a) &\iff \\ \forall \eta, \bigotimes C\eta[v := a] \geq_S \alpha &\implies \bigotimes C\eta[v := b] \geq_S \alpha. \end{aligned}$$

Now, when $\bigotimes C\eta[v := a] <_S \alpha$ both the clauses are true; when $\bigotimes C\eta[v := a] \geq_S \alpha$, by hypothesis, we have $\bigotimes C\eta[v := a] \leq_S \bigotimes C\eta[v := b]$. For transitivity, we easily have $\bigotimes C\eta[v := b] \geq_S \alpha$. We can extend the result for NS, NI and FI.

As an example of the just given definitions, consider Figure 9.3. The domain values c and b for variable y are ${}_{0.2}$Neighborhood Interchangeable. In fact, the tuple involving c and b only differ for the tuple $\langle b, c\rangle$ that has value 0.1 and for the tuple $\langle b, b\rangle$ that has value 0. Since we are interested only to solutions greater than 0.2, these tuples are excluded from the match.

We can see also that values a and b for variable y are ${}_{0.2-\text{set}}$Neighborhood. In fact the set of solution tuples with value greater than 0.2 are the same. Notice that a and b are not ${}_{0.2}$Neighborhood Interchangeable because tuples $\langle a, a\rangle$ and $\langle a, b\rangle$ have values 0.8 and 0.2 respectively.

The meaning of degradation assume different meanings when instantiated to different semirings:

1. fuzzy CSP: $b \in {}^{\delta}FS_v(a)$ gets instantiated to:

   $$min(min_{c\in C}(c\eta[v := a]), \delta) \leq min_{c\in C}(c\eta[v := b])$$

   which means that changing $v := b$ to $v := a$ does not make the solution worse than before or worse than δ. In the practical case where we want to only consider solutions with a quality better than δ, this means that substitution will never put a solution out of this class.
2. weighted CSP: $b \in {}^{\delta}FS_v(a)$ gets instantiated to:

   $$\sum_{c\in C} c\eta[v := a] + \delta \geq_S \sum_{c\in C} c\eta[v := b]$$

which means that the penalty for the solution does not increase by more than a factor of δ. This allows for example to express that we would not want to tolerate more than δ in extra cost. Note, by the way, that $\leq_S$ translates to $\geq_S$ in this version of the soft CSP.

3. probabilistic CSP: $b \in {}^{\delta}FS_v(a)$ gets instantiated to:

$$(\prod_{c\in C} c\eta[v := a]) \cdot \delta \leq \prod_{c\in C} c\eta[v := b]$$

which means that the solution with $v = b$ is not degraded by more than a factor of δ from the one with $v = a$.

4. crisp CSP: $b \in {}^{\delta}FS_v(a)$ gets instantiated to:

$$(\bigwedge_{c\in C} c\eta[v := a]) \wedge \delta \Rightarrow (\bigwedge_{c\in C} c\eta[v := b])$$

which means that when $\delta = true$, whenever a solution with $v = a$ satisfies all constraints, so does the same solution with $v = b$. When $\delta = false$, it is trivially satisfied (i.e. δ is too loose a bound to be meaningful).

9.2.2 Properties of Degradations and Thresholds

As it is very complex to determine full interchangeability/substitutability, we start by showing the fundamental theorem that allows us to approximate ${}^{\delta}/_{\alpha/\alpha-\text{set}}FS/FI$ by ${}^{\delta}/_{\alpha/\alpha-\text{set}}NS/NI$:

Theorem 9.2.4 (Extensivity). *${}^{\delta}$neighbourhood substitutability implies ${}^{\delta}$full substitutability, ${}_{\alpha}$neighbourhood substitutability implies ${}_{\alpha}$full substitutability and ${}_{\alpha-\text{set}}$neighbourhood substitutability implies ${}_{\alpha-\text{set}}$full substitutability.*

Proof. – δ: Since the assignments $v := a$ and $v := b$ only involve constraints in C_v, and for the extensivity properties of times, we easily have that

$$\begin{aligned} b \in NS_v(a) &\iff \\ \forall\eta, \bigotimes C_v\eta[v := a] \times_S \delta \leq_S \bigotimes C_v\eta[v := b] & \\ &\Longrightarrow \\ \forall\eta, \bigotimes C\eta[v := a] \times_S \delta \leq_S \bigotimes C\eta[v := b] & \\ &\iff b \in FS_v(a). \end{aligned}$$

– α: When $\bigotimes C_v\eta[v := a] < \alpha$ also $\bigotimes C\eta[v := a] < \alpha$, so both the clauses are true; when $\bigotimes C_v\eta[v := a] \geq_S \alpha$, since $\bigotimes C_v\eta[v := a] \leq_S \bigotimes C_v\eta[v := b]$, we have by extensivity $\bigotimes C\eta[v := a] \leq_S \bigotimes C\eta[v := b]$.

– $\alpha-$set: As before, when $\bigotimes C_v\eta[v := a] < \alpha$ also $\bigotimes C\eta[v := a] < \alpha$, so both the clauses are true. When $\bigotimes C_v\eta[v := a] \geq_S \alpha$, since by hypothesis $b \in {}_{\alpha-\text{set}}FS_v(a)$, I have $\bigotimes C_v\eta[v := b] \geq_S \alpha$; now per extensivity we have also $\bigotimes C\eta[v := b] \geq_S \alpha$.

Easily, we can extend the result to interchangeability.

This theorem is of fundamental importance since it gives us a way to approximate full interchangeability by neighborhood interchangeability which is much less expensive to compute.

Theorem 9.2.5 (Transitivity using thresholds and degradations). *Consider three domain values a, b and c, for a variable v. Then,*

$$b \in {}^{\delta_1}NS_v(a), a \in {}^{\delta_2}NS_v(c) \implies b \in {}^{\delta_1 \times \delta_2}NS_v(c) \text{ and}$$
$$b \in {}_{\alpha_1}NS_v(a), a \in {}_{\alpha_2}NS_v(c) \implies b \in {}_{\alpha_1+\alpha_2}NS_v(c)$$

Similar results holds for FS, NI, FI.

Proof. – δ: By definition

$$a \in {}^{\delta_2}NS_v(c) \iff \forall\eta, \bigotimes C_v\eta[v := c] \times_S \delta_2 \leq_S \bigotimes C_v\eta[v := a].$$

For monotonicity we have

$$\forall\eta, \bigotimes C_v\eta[v := c] \times_S \delta_2 \times_S \delta_1 \leq_S \bigotimes C_v\eta[v := a] \times_S \delta_1.$$

Now, by definition

$$b \in {}^{\delta_1}NS_v(a) \iff \forall\eta, \bigotimes C_v\eta[v := a] \times_S \delta_1 \leq_S \bigotimes C_v\eta[v := b].$$

For transitivity we easily have

$$\forall\eta, \bigotimes C_v\eta[v := c] \times_S \delta_2 \times_S \delta_1 \leq_S \bigotimes C_v\eta[v := b] \iff b \in {}^{\delta_1 \times \delta_2}NS_v(c).$$

– α: By hypothesis we have

$$b \in {}_{\alpha_1}NS_v(a) \iff$$
$$\bigotimes C_v\eta[v := a] \geq_S \alpha_1 \implies \bigotimes C_v\eta[v := a] \leq_S \bigotimes C_v\eta[v := b] \text{ and,}$$
$$a \in {}_{\alpha_2}NS_v(c) \iff$$
$$\bigotimes C_v\eta[v := c] \geq_S \alpha_2 \implies \bigotimes C_v\eta[v := c] \leq_S \bigotimes C_v\eta[v := a].$$

Since $\alpha_1 +_S \alpha_2 \geq_S \alpha_1$ and $\alpha_1 +_S \alpha_2 \geq_S \alpha_2$ and transitivity of $\implies$, we have

$$\bigotimes C_v\eta[v := a] \geq_S \alpha_1 +_S \alpha_2 \implies \bigotimes C_v\eta[v := a] \leq_S \bigotimes C_v\eta[v := b]$$
and,
$$\bigotimes C_v\eta[v := c] \geq_S \alpha_1 +_S \alpha_2 \implies \bigotimes C_v\eta[v := c] \leq_S \bigotimes C_v\eta[v := a].$$

Now for transitivity of $\leq_S$, we have

$$\bigotimes C_v\eta[v := c] \geq_S \alpha_1 +_S \alpha_2 \implies \bigotimes C_v\eta[v := c] \leq_S \bigotimes C_v\eta[v := b]$$
$$\iff b \in {}_{\alpha_1 +_S \alpha_2}NS_v(c).$$

Easily, we can extend the result to FS, NI, FI.

In particular when $\alpha_1 = \alpha_2 = \alpha$ and $\delta_1 = \delta_2 = \delta$ we have:

Corollary 9.2.1 (Transitivity and equivalence classes). *Consider three domain values a, b and c, for a variable v. Then,*

- *Threshold interchangeability is a transitive relation, and partitions the set of values for a variable into equivalence classes, that is*

$$\begin{aligned} b \in {}_{\alpha}NS_v(a), a \in {}_{\alpha}NS_v(c) &\implies b \in {}_{\alpha}NS_v(c) \\ {}_{\alpha}NI_v(b/a), {}_{\alpha}NI_v(a/c) &\implies {}_{\alpha}NI_v(b/c) \\ b \in {}_{\alpha-\text{set}}NS_v(a), a \in {}_{\alpha-\text{set}}NS_v(c) &\implies b \in {}_{\alpha-\text{set}}NS_v(c) \\ {}_{\alpha-\text{set}}NI_v(b/a), {}_{\alpha-\text{set}}NI_v(a/c) &\implies {}_{\alpha-\text{set}}NI_v(b/c) \end{aligned}$$

- *If the $\times_S$-operator is idempotent, then degradation interchangeability is a transitive relation, and partitions the set of values for a variable into equivalence classes, that is*

$$\begin{aligned} b \in {}^{\delta}NS_v(a), a \in {}^{\delta}NS_v(c) &\implies b \in {}^{\delta}NS_v(c) \\ {}^{\delta}NI_v(b/a), {}^{\delta}NI_v(a/c) &\implies {}^{\delta}NI_v(b/c) \end{aligned}$$

Proof.
- δ: Suppose to have $\delta_1 = \delta_2 = \delta$. Since times is idempotent, we have $\delta_1 \times \delta_2 = \delta$. Using the results of the previous theorem the corollary easily follows.
- α: Since when $\alpha_1 = \alpha_2 = \alpha$ we have $\alpha_1 +_S \alpha_2 = \alpha$, the corollary easily follows from the previous theorem.
- $\alpha-$set: By hypothesis we have

$$b \in {}_{\alpha}NS_v(a) \iff \bigotimes C\eta[v := a] \geq_S \alpha \implies \bigotimes C\eta[v := b] \geq_S \alpha \text{ and,}$$
$$a \in {}_{\alpha}NS_v(c) \iff \bigotimes C\eta[v := c] \geq_S \alpha \implies \bigotimes C\eta[v := a] \geq_S \alpha.$$

For transitivity of $\implies$, we have

$$\bigotimes C\eta[v := c] \geq_S \alpha \implies \bigotimes C\eta[v := a] \geq_S \alpha.$$

Interchangeability easily follows.

By using degradations and thresholds we have a nice way to decide when two domain values for a variable can be substituable/interchangeable. In fact, by changing the α or δ parameter we can obtain different results.

In particular we can show that an “extensivity” result for the parameters holds. In fact, it is straightforward to notice that if two values are ${}^{\delta}_{\alpha/\alpha-\text{set}}$substitutable, they have to be also ${}^{\delta'}_{\alpha'/\alpha'-\text{set}}$substitutable for any $\delta' \leq \delta$ and $\alpha' \geq_S \alpha$.

Theorem 9.2.6 (Extensivity for α and δ). *Consider two domain values a and b, for a variable v, two thresholds α and α' s.t. $\alpha \leq \alpha'$ and two degradations δ and δ' s.t. $\delta \geq_S \delta'$. Then,*

$$a \in {}^{\delta}NS_v(b) \implies a \in {}^{\delta'}NS_v(b)$$
$$a \in {}_{\alpha}NS_v(b) \implies a \in {}_{\alpha'}NS_v(b)$$

Similar results holds for FS, NI, FI.

Proof. – δ: By definition

$$a \in {}^{\delta}NS_v(b) \iff \forall\eta, \bigotimes C_v\eta[v := b] \times_S \delta \leq_S \bigotimes C_v\eta[v := a].$$

By monotonicity of times, we have

$$\bigotimes C_v\eta[v := b] \times_S \delta' \leq_S \bigotimes C_v\eta[v := b] \times_S \delta.$$

By transitivity of $\leq_S$

$$\forall\eta, \bigotimes C_v\eta[v := b] \times_S \delta' \leq_S \bigotimes C_v\eta[v := a] \iff a \in {}^{\delta'}NS_v(b).$$

– α: By Definition we have

$$a \in {}_{\alpha}NS_v(b) \iff$$
$$\bigotimes C_v\eta[v := b] \geq_S \alpha \implies \bigotimes C_v\eta[v := b] \leq_S \bigotimes C_v\eta[v := a].$$

Since $\alpha' \geq_S \alpha$, we have

$$\bigotimes C_v\eta[v := b] \geq_S \alpha' \implies \bigotimes C_v\eta[v := b] \geq_S \alpha.$$

By Transitivity of $\implies$ we have

$$\bigotimes C_v\eta[v := b] \geq_S \alpha' \implies \bigotimes C_v\eta[v := b] \leq_S \bigotimes C_v\eta[v := a]$$
$$\iff a \in {}_{\alpha'}NS_v(b).$$

Easily, we can extend the result to FS, NI, FI.

As a corollary when threshold and degradation are $\mathbf{0}$ or $\mathbf{1}$ we have some special results.

Corollary 9.2.2. *When $\alpha = \mathbf{0}$ and $\delta = \mathbf{1}$, we obtain the non approximated versions of NS.*

$$\forall a, b,\ a \in {}_{\mathbf{0}}NS_v(b) \text{ and } a \in {}^{\mathbf{1}}NS_v(b) \iff a \in NS(b)$$

Similar results holds for FS, NI, FI.

Proof. – When $\alpha = \mathbf{0}$, we always have $\bigotimes C_v\eta[v := b] \geq_S \alpha$. So to check if $a \in {}_{\mathbf{0}}NS_v(b)$ we need only to check that $\bigotimes C_v\eta[v := b] \leq_S \bigotimes C_v\eta[v := a]$.
– When $\delta = \mathbf{1}$, we have $\bigotimes C_v\eta[v := b] \times_S \delta = \bigotimes C_v\eta[v := b]$. So to check if $a \in {}^{\mathbf{1}}NS_v(b)$ we need only to check that $\bigotimes C_v\eta[v := b] \leq_S \bigotimes C_v\eta[v := a]$.

Let us remind that degradations and thresholds can be used together; so we easily have

- ${}^{\mathbf{1}}_{\mathbf{0}}NS = {}_{\mathbf{0}}NS = {}^{\mathbf{1}}NS = NS$;
- $NS \implies {}^{\delta}NS \implies {}^{\delta}_{\alpha}NS$ for any δ and α;
- $NS \implies {}_{\alpha}NS \implies {}^{\delta}_{\alpha}NS$ for any δ and α.

9.2.3 Computing ${}^{\delta}/_{\alpha/\alpha-\mathrm{set}}$-Substitutability/Interchangeability

The result of Theorem 9.2.1 is fundamental since it gives us a way to approximate full substitutability/interchangeability by neighbourhood substituability/interchangeability which is much less costly to compute.

The most general algorithm for neighborhood substituability/interchangeability in the soft CSP framework is to check for each pair of values whether the condition given in the definition holds or not. This algorithm has a time complexity exponential in the size of the neighbourhood and quadratic in the size of the domain (which may not be a problem when neighbourhoods are small).

Better algorithms can be given when the times operator of the semiring is idempotent. In this case, instead of considering the combination of all the constraint C_v involving a certain variable v, we can check the property we need (NS/NI and their relaxed versions ${}^{\delta}_{\alpha/\alpha-\mathrm{set}}NS/NI$) on each constraint itself.

Theorem 9.2.7. *Consider two domain values b and a, for a variable v, and the set of constraints C_v involving v. Then we have $\forall c \in C_v$:*

$$c\eta[v := a] \leq_S c\eta[v := b] \implies b \in NS_v(a) \tag{9.1}$$

$$(c\eta[v := a] \geq_S \alpha \implies c\eta[v := a] \leq_S c\eta[v := b]) \implies b \in {}_{\alpha}NS_v(a) \tag{9.2}$$

If the times operator of the semiring is idempotent we also have:

$$\forall c \in C_v . c\eta[v := a] \times_S \delta \leq_S c\eta[v := b] \implies b \in {}^{\delta}NS_v(a) \tag{9.3}$$

$$(c\eta[v := a] \geq_S \alpha \implies c\eta[v := b] \geq_S \alpha) \implies b \in {}_{\alpha-\mathrm{set}}NS_v(a) \tag{9.4}$$

Proof. 1. Easily follows from the monotonicity of times.
2. For extensivity of times we have $\bigotimes C_v\eta[v := a] \leq_S \alpha \implies c\eta[v := a] \geq_S \alpha$. For monotonicity of times we have $c\eta[v := a] \leq_S c\eta[v := b] \implies \bigotimes C_v\eta[v := a] \leq_S \bigotimes C_v\eta[v := b]$. The thesis follows from transitivity of $\implies$.
3. For extensivity of times we have easily $c\eta[v := a] \times_S \delta \leq_S c\eta[v := b] \implies \bigotimes C_v\eta[v := a] \times_S \delta \leq_S \bigotimes C_v\eta[v := b]$, and this is exactly the definition of $b \in {}^{\delta}NS_v(a)$.
4. Easily follows from monotonicity and idempotency of times.

By using Theorem 9.2.7 (and Corollary 9.2.1 for ${}^{\delta}/_{\alpha/\alpha-\mathrm{set}}NS$) we can find substituable/interchangeable domain values more efficiently. Algorithm 2 shows an algorithm that can be used to find domain values that are Neighborhood Interchangeable. It uses a data structure similar to the *discrimination trees*, first introduced by Freuder in [106] . Algorithm 2 can compute different versions of neighbourhood interchangeability depending on the algorithm $NI-nodes$ used. Algorithm 3 shows the simplest version without threshold or degradation.

We can show that Algorithm 2 with procedure Algorithm 3 is sound (that is compute correct classes of equivalence for NI).

Theorem 9.2.8 (Soundness of *NI* algorithm). *Algorithm 2 using Algorithm 3 returns as a result a subset of the neighbourhood interchangeable sets.*

Algorithm 2. Algorithm to compute neighbourhood interchangeable sets for variable v_i

```
1: Create the root of the discrimination tree for variable $v_i$
2: Let $C_{v_i} = \{c \in C \mid v_i \in supp(c)\}$
3: Let $D_{v_i} = \{$the set of domain values $d_{v_i}$ for variable $v_i\}$
4: for all $d_{v_i} \in D_{v_i}$ do
5:    for all $c \in C_v$ do
6:       execute Algorithm $NI$-Nodes$(c, v, d_{v_i})$ to build the nodes associated with $c$
7:    Go back to the root of the discrimination tree.
```

Algorithm 3. NI-Nodes(c, v, d_{v_i}) for Soft-NI

```
1: for all assignments $\eta_c$ to variables in $supp(c)$ do
2:    compute the semiring level $\beta = c\eta_c[v_i := d_{v_i}]$,
3:    if there exists a child node corresponding to $\langle c = \eta_c, \beta \rangle$ then
4:       move to it,
5:    else
6:       construct such a node and move to it.
7: Add $v_i, \{d_{v_i}\}$ to annotation of the last build node,
```

Proof. By looking at Algorithm 3, two domain values d_{v_i} and d'_{v_i} will be in the same leaf node, if and only if they follow the same path. They follow the same path if and only if for all η, and for all $c \in C$, $c\eta[v_i := d_{v_i}] = c\eta[v_i := d'_{v_i}]$. This can be rewritten as $c\eta[v_i := d_{v_i}] \leq_S c\eta[v_i := d'_{v_i}]$ and $c\eta[v_i := d'_{v_i}] \leq_S c\eta[v_i := d_{v_i}]$. Now by Theorem 9.2.7 this is equivalent to $d_{v_i} \in NS_{v_i}(d'_{v_i})$ and $d'_{v_i} \in NS_{v_i}(d_{v_i})$, that is $NI_{v_i}(d_{v_i}/d'_{v_i})$.

The algorithm is very similar to that defined by Freuder in [106], and when we consider the semiring for classical CSPs $S_{CSP} = \langle \{false, true\}, \vee, \wedge, false, true \rangle$ and all constraints are binary, it computes the same result. Notice that for each node we add also an information representing the cost of the assignment η_c.

When all constraints are binary, considering all constraints involving variable v is the same as considering all variables connected to v by a constraint, and our algorithm performs steps as that given by Freuder.

We can determine the complexity of the algorithm by considering that the algorithm calls $NI - Nodes$ for each $k - ary$ constraint exactly once for each value of each the k variables; this can be bounded from above by $k * d$ with d the maximum domain size. Thus, given m constraints, we obtain a bound of

$$O(m * k * d * O(AlgorithmNI{-}nodes)).$$

The complexity of $AlgorithmNI{-}nodes$ strictly depends on the size of the domain d and from the number of variables k involved in each constraint and is given as

$$O(AlgorithmNI{-}nodes) = d^{k-1}.$$

For complete constraint graphs of binary constraints ($k = 2$), we obtain the same complexity bound of $O(n^2 d^2)$ as Freuder in [106].

Algorithm 4. NI-Nodes(c, v, d_{v_i}, α) for Soft ${}_{\alpha}NI$

```
1: for all assignments $\eta_c$ to variables in $supp(c)$ do
2:    compute the semiring level $\beta = c\eta_c[v_i := d_{v_i}]$,
3:    if $\beta \not\geq \alpha$ then
4:       $\beta := \alpha$ {I do not want to discriminate in this case},
5:    if there exists a child node corresponding to $\langle c = \eta_c, \beta \rangle$ then
6:       move to it,
7:    else
8:       construct such a node and move to it.
9: Add $v_i, \{d_{v_i}\}$ to annotation of the last build node,
```

Algorithms for the relaxed versions of NI are obtained by substituting different versions of Algorithm 3. For ${}_{\alpha}NI$, the algorithm needs to discriminate only when the semiring value is greater than α, as shown in Algorithm 4.

Theorem 9.2.9 (Soundness of the ${}_{\alpha}NI$ algorithm). *Algorithm 2 using Algorithm 4 returns as a result a subset of the ${}_{\alpha}$Neighbourhood interchangeabilities.*

Proof. By looking at Algorithm 4, two domain values d_{v_i} and d'_{v_i} will be in the same leaf node, if and only if they follow the same path. They follow the same path if and only if for all η, and for all $c \in C$,

- both $c\eta[v_i := d_{v_i}]$ and $c\eta[v_i := d'_{v_i}]$ have a semiring value less than α, or
- $c\eta[v_i := d_{v_i}] = c\eta[v_i := d'_{v_i}]$

This can be written as:

$$(\neg(c\eta[v_i := d_{v_i}] \geq \alpha) \wedge \neg(c\eta[v_i := d'_{v_i}] \geq \alpha)) \vee (c\eta[v_i := d_{v_i}] = c\eta[v_i := d'_{v_i}]) \quad (9.5)$$

which, by distributing the first two terms, is equivalent to:

$$\begin{array}{r} c\eta[v_i := d_{v_i}] \geq \alpha \implies \\ (c\eta[v_i := d_{v_i}] = c\eta[v_i := d'_{v_i}]) \implies (c\eta[v_i := d_{v_i}] \leq c\eta[v_i := d'_{v_i}]) \\ \wedge \\ c\eta[v_i := d'_{v_i}] \geq \alpha \implies \\ (c\eta[v_i := d_{v_i}] = c\eta[v_i := d'_{v_i}]) \implies (c\eta[v_i := d'_{v_i}] \leq c\eta[v_i := d_{v_i}]). \end{array}$$

Now by Theorem 9.2.7 this implies $d_{v_i} \in {}_{\alpha}NS_{v_i}(d'_{v_i})$ and $d'_{v_i} \in {}_{\alpha}NS_{v_i}(d_{v_i})$, that is ${}_{\alpha}NI_{v_i}(d_{v_i}/d'_{v_i})$.

In Algorithm 5 instead, we have to filter out the tuples whose semiring value is lower than α and we do not make any difference among tuples greater than α.

Theorem 9.2.10 (Soundness of ${}_{\alpha-\mathrm{set}}NI$ algorithm). *For semirings with idempotent $\times$ operator, Algorithm 2 using Algorithm 5 returns as result a subset of the ${}_{\alpha-\mathrm{set}}$Neighbourhood interchangeabilities.*

Algorithm 5. NI-Nodes(c, v, d_{v_i}, α) for Soft $_{\alpha-\text{set}}NI$

```
1: for all assignments $\eta_c$ to variables in $supp(c)$ do
2:    compute the semiring level $\beta = c\eta_c[v_i := d_{v_i}]$,
3:    if $\beta \not\geq \alpha$ then
4:       $\beta := \alpha$ {I do not want to discriminate in this case},
5:    else
6:       $\beta := \alpha$ {Does not matter how bigger than $\alpha$}.
7:    if there exists a child node corresponding to $\langle c = \eta_c, \beta \rangle$ then
8:       move to it,
9:    else
10:      construct such a node and move to it.
11: Add $v_i, \{d_{v_i}\}$ to annotation of the last build node,
```

Proof. By looking at Algorithm 5, two domain values d_{v_i} and d'_{v_i} will be in the same leaf node, only if they follow the same path. If they follow the same path, means that for all η, and for all $c \in C$,

- both $c\eta[v_i := d_{v_i}]$ and $c\eta[v_i := d'_{v_i}]$ have a semiring value not greater than α, or
- both have to be bigger than α.

This is is written as:

$$\begin{gathered}\neg((c\eta[v_i := d_{v_i}] \geq \alpha) \vee (c\eta[v_i := d'_{v_i}] \geq \alpha)) \\ \vee \\ (c\eta[v_i := d_{v_i}] \geq \alpha) \wedge (c\eta[v_i := d'_{v_i}] \geq \alpha).\end{gathered}$$

which by distributions transforms into:

$$\begin{aligned}&(\neg(c\eta[v_i := d_{v_i}] \geq \alpha) \vee (c\eta[v_i := d_{v_i}] \geq \alpha))\wedge \\ &(\neg(c\eta[v_i := d_{v_i}] \geq \alpha) \vee (c\eta[v_i := d'_{v_i}] \geq \alpha))\wedge \\ &(\neg(c\eta[v_i := d'_{v_i}] \geq \alpha) \vee (c\eta[v_i := d_{v_i}] \geq \alpha))\wedge \\ &(\neg(c\eta[v_i := d'_{v_i}] \geq \alpha) \vee (c\eta[v_i := d'_{v_i}] \geq \alpha))\end{aligned}$$

and by elimination of tautologies:

$$\begin{aligned}&(\neg(c\eta[v_i := d_{v_i}] \geq \alpha) \vee (c\eta[v_i := d'_{v_i}] \geq \alpha))\wedge \\ &(\neg(c\eta[v_i := d'_{v_i}] \geq \alpha) \vee (c\eta[v_i := d_{v_i}] \geq \alpha))\end{aligned}$$

Using the fact that $a \implies B \equiv \neg A \vee B$ this can be rewritten as:

$$\begin{aligned}&((c\eta[v_i := d_{v_i}] \geq \alpha) \implies (c\eta[v_i := d'_{v_i}] \geq \alpha))\wedge \\ &((c\eta[v_i := d'_{v_i}] \geq \alpha) \implies (c\eta[v_i := d_{v_i}] \geq \alpha)).\end{aligned}$$

Now by Theorem 9.2.7 this means that $d_{v_i} \in {}_{\alpha-\text{set}}NS_{v_i}(d'_{v_i})$ and $d'_{v_i} \in {}_{\alpha-\text{set}}NS_{v_i}(d_{v_i})$, that is ${}_{\alpha-\text{set}}NI_{v_i}(d_{v_i}/d'_{v_i})$.

Algorithm 6. NI-Nodes(c, v, d_{v_i}, δ) for Soft ${}^{\delta}NI$

```
1: for all assignments $\eta_c$ to variables in $supp(c)$ do
2:    compute the level $\beta = c\eta_c[v_i := d_{v_i}]$, and the bound $\kappa = \beta \times \delta$,
3:    if there exists a child node corresponding to $\langle \bar{\kappa}, (c = \eta_c), \bar{\beta} \rangle$ with $(\bar{\kappa} \leq \beta) \wedge (\kappa \leq \bar{\beta})$ then
4:       move to it and change the label to $\langle lub(\bar{\kappa}, \kappa), (c = \eta_c), glb(\bar{\beta}, \beta) \rangle$,
5:    else
6:       construct the node $\langle \kappa, (c = \eta_c), \beta \rangle$ and move to it.
7: Add $v_i, \{d_{v_i}\}$ to annotation of the last build node,
```

For ${}^{\delta}NI$, the algorithm needs to only consider tuples that can cause a degradation by more than δ, as shown in Algorithm 6. The idea here is to save in each node the information needed to check at each step ${}^{\delta}NS$ in both directions. In a semiring with total order, the information represents the "interval of degradation". As both algorithms consider the same assignments as Algorithm 3, their complexity remains unchanged at $O(d^{k-1})$.

Theorem 9.2.11 (Soundness of the ${}^{\delta}NI$ algorithm). *For semirings with idempotent $\times$ operator, Algorithm 2 using Algorithm 6 gives as result a subset of the ${}^{\delta}$interchangeabilities.*

Proof. By looking at Algorithm 6, two domain values d_{v_i} and d'_{v_i} will be in the same leaf node if and only if they follow the same path. Consider now for each node related to constraint c and to the assignment η, $c\eta_c[v_i := d_{v_i}] = \beta$, $\kappa = \beta \times \delta$, $c\eta_c[v_i := d'_{v_i}] = \beta'$, and $\kappa' = \beta' \times \delta$. If they follow the same path, each of the nodes will have a label $\langle lub(\bar{\kappa}, \kappa, \kappa'), c = \eta_c, glb(\bar{\beta}, \beta, \beta') \rangle$, where $\bar{\kappa}$ and $\bar{\beta}$ are determined by other assignments that have passed through the node.

Because of the condition in step 3 of Algorithm 6, the algorithm ensures that $lub(\bar{\kappa}, \kappa, \kappa') \leq glb(\bar{\beta}, \beta, \beta')$. It follows that $(\kappa' \leq \beta)$ and $\kappa \leq \beta'$.

By Theorem 9.2.7 this means that $d_{v_i} \in {}^{\delta}NS_{v_i}(d'_{v_i})$ and $d'_{v_i} \in {}^{\delta}NS_{v_i}(d_{v_i})$, that is ${}^{\delta}NI_{v_i}(d_{v_i}/d'_{v_i})$.

9.3 An Example

Figure 9.4 shows the graph representation of a CSP which might represent a car configuration problem.

A product catalog might represent the available choices through a soft CSP. With different choices of semiring, the CSP of Figure 9.4 can represent different problem formulations:

Example 1 *For optimizing the cost of the product, a representation as a weighted CSP might be most appropriate. Here, the semiring models the cost of the different options and their integration with the others, using the semiring:* $< \Re^{+}, min, +, +\infty, 0 >$. *We might have the constraints:*

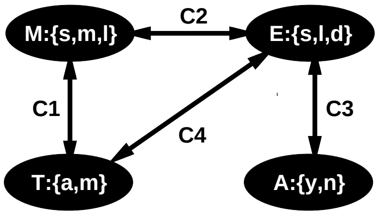


Fig. 9.4. Example of a CSP modeling car configuration with 4 variables: M = model, T = transmission, A = Air Conditioning, E = Engine

$$C_1 = \begin{array}{c|ccc} & \multicolumn{3}{c}{M} \\ & s & m & l \\ \hline T\ a & \infty & 5 & 3 \\ m & 2 & 3 & 50 \end{array} \quad C_2 = \begin{array}{c|ccc} & \multicolumn{3}{c}{M} \\ & s & m & l \\ \hline s & 3 & 5 & \infty \\ E\ l & 30 & 3 & 3 \\ d & 5 & 5 & \infty \end{array} \quad C_3 = \begin{array}{c|ccc} & \multicolumn{3}{c}{E} \\ & s & l & d \\ \hline A\ y & 5 & 2 & 7 \\ n & 0 & 30 & 0 \end{array} \quad C_4 = \begin{array}{c|ccc} & \multicolumn{3}{c}{E} \\ & s & l & d \\ \hline T\ a & \infty & 3 & \infty \\ m & 4 & 10 & 5 \end{array}$$

and also unary constraints C_M, C_E, C_T and C_A that model the cost of the components:

$$C_M = \frac{s\ \ m\ \ l}{10\ 20\ 30} \quad C_E = \frac{s\ \ l\ \ d}{10\ 20\ 20} \quad C_T = \frac{a\ \ m}{15\ 10} \quad C_A = \frac{y\ \ n}{10\ 0}$$

Figure 9.5 shows how occurrence of $^{\delta}/_{\alpha}$substitutability among values of variable E change w.r.t. δ and α for Example 1. We can see that when δ takes high values of the semiring, small degradation in the solution is allowed. Thus for $\delta = 0$ only s can substitute d. As δ decreases in the values of the semiring, here goes to ∞, there is more degradation allowed in the solution and thus more $^{\delta}$substitutability among the values of the variable E.

Let's now consider the second part of Figure 9.5. For high semiring values of α all the values are interchangeable. For $\alpha = 18$ d and l are interchangeable, and s can substitute l and d.

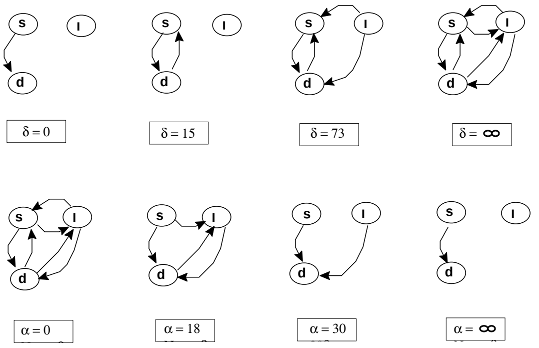


Fig. 9.5. Example of how δ-substitutability and α-substitutability varies in the weighted CSP over the values of variable E from Fig. 9.4

Notice that thresholds α and degradation δ are two different notions of approximations and compute different notions of interchangeabilities. As an example, by using degradation $\delta = 15$ we obtain s and d interchangeable, whilst, by using threshold $\alpha = 18$ we obtain l and d interchangeable.

In Figure 9.6 we represent the variance of ${}_{\alpha-\text{set}}NS$ depending on the threshold α for weighted CSP example. For small α (between 0 and 17) or big α (∞ in the Figure) all the values are ${}_{\alpha-\text{set}}$*interchangeable*. The number of ${}_{\alpha-\text{set}}$*substitutable* values is decreasing with α takes values of medium size ($\alpha = 36$ in the Figure). It is interesting to notice that for this example s is always ${}_{\alpha-\text{set}}$*substitutable* for d.

Example 2 *Another optimization criterion might be the time it takes to build the car. Delay is determined by the time it takes to obtain the components and to reserve the resources for the assembly process. For the delivery time of the car, only the longest delay would matter. This could be modelled by the semiring* $< \Re^+, min, max, +\infty, 0 >$[1]*, with the binary constraints:*

$$C_1 = \begin{array}{cc|ccc} & & & M & \\ & & s & m & l \\ \hline T & a & \infty & 3 & 4 \\ & m & 2 & 4 & \infty \end{array} \quad C_2 = \begin{array}{cc|ccc} & & & M & \\ & & s & m & l \\ \hline & s & 2 & 3 & \infty \\ E & l & 30 & 3 & 3 \\ & d & 2 & 3 & \infty \end{array} \quad C_3 = \begin{array}{cc|ccc} & & & E & \\ & & s & l & d \\ \hline A & y & 5 & 4 & 7 \\ & n & 0 & 30 & 0 \end{array} \quad C_4 = \begin{array}{cc|ccc} & & & E & \\ & & s & l & d \\ \hline T & a & \infty & 3 & \infty \\ & m & 4 & 10 & 3 \end{array}$$

and unary constraints C_M, C_E, C_T *and* C_A *that model the time to obtain the components:*

$$C_M = \begin{array}{ccc} s & m & l \\ \hline 2 & 3 & 3 \end{array} \quad C_E = \begin{array}{ccc} s & l & d \\ \hline 3 & 2 & 3 \end{array} \quad C_T = \begin{array}{cc} a & m \\ \hline 1 & 2 \end{array} \quad C_A = \begin{array}{cc} y & n \\ \hline 3 & 0 \end{array}$$

Let us now consider the variable E of Example 2 and compute ${}^{\delta}/_{\alpha}NS/NI$ between its values by using Definition 9.2.4 and Definition 9.2.5. In Figure 9.7 directed arcs are added when the source can be ${}^{\delta}/_{\alpha}$substituted to the destination

[1] This semiring and the fuzzy one are similar, but the first uses an opposite order. Let us call this semiring *opposite-fuzzy*.

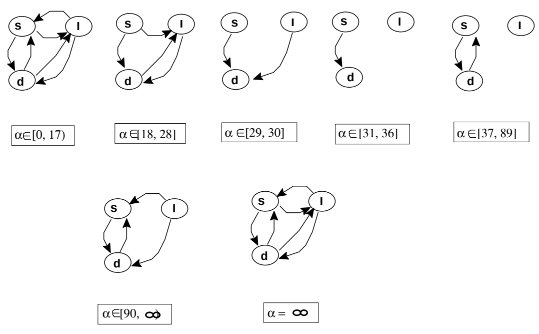


Fig. 9.6. Example of how $\alpha-$set-substitutability varies in the weighted CSP over the values of variable E from Fig. 9.4

node. It is easy to see how the occurrences of ${}^{\delta}/_{\alpha}NS$ change, depending on δ and α degrees.

We can notice that when δ takes value 0 (the **1** of the optimization semiring), small degradation is allowed in the CSP tuples when the values are substituted; thus only value s can be substituted for value d. As δ increases in value (or decreases from the semiring point of view) higher degradation of the solutions is allowed and thus the number of substitutabilities increase with it.

In the second part of Figure 9.7 we can see that for $\alpha = 0$ all the values are interchangeable (in fact, since there are no solutions better than $\alpha = 0$, by definition all the elements are ${}_{\alpha}$interchangeable).

For a certain threshold ($\alpha = 4$) values s and d are ${}_{\alpha}$interchangeable and value l can substitute values s and d. Moreover, when α is greater than 5 we only have that s can substitute d.

Further we consider the same variable E of the Example 2 for fuzzy CSP case and compute ${}_{\alpha-\mathrm{set}}NS/NI$ by using the definition Definition 9.2.5. In Figure 9.8, we can see how the occurence of ${}_{\alpha-\mathrm{set}}NS$ varies depending on the threshold α.

When α takes value 0 or ∞ all the domain values of variable E are ${}_{\alpha-\mathrm{set}}in$terchangeable. When α varies between 0 and 4, the domain value s is ${}_{\alpha-\mathrm{set}}$*interchangeable* with values l and d, while only d can be ${}_{\alpha-\mathrm{set}}$*substitutable* for value l. For an α between 4 and 29 we can interchange only values s and d, while for an α above 30 we can substitute also value l for s and d as well.

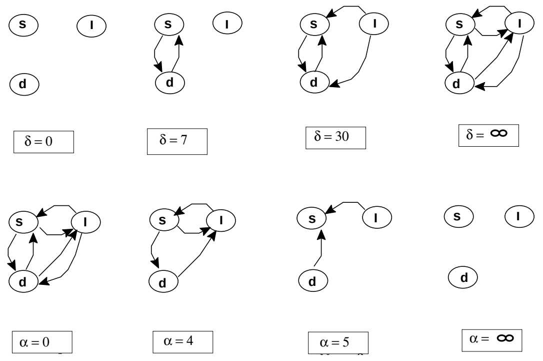


Fig. 9.7. Example of how δ-substitutability and α-substitutability varies in the opposite-fuzzy CSP over the values of variable E from Fig. 9.4

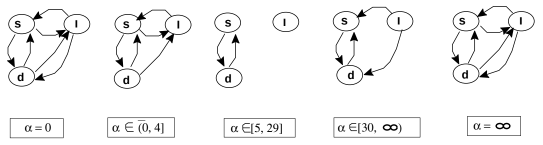


Fig. 9.8. Example of how $\alpha-$set-substitutability varies in the opposite-fuzzy CSP over the values of variable E from Fig. 9.4

We will show now how to compute interchangeabilities by using the Discrimination Tree algorithm. In Figure 9.9 the Discrimination Tree is described for variable M when $\alpha = 2$ and $\alpha = 3$. We can see that values m and l for variable M are ${}_2$interchangeable whilst there are no interchangeabilities for $\alpha = 3$.

9.4 Partial Interchangeability

Similar to Freuder [106] we define also some notions of substitutability/interchangeability that consider more than one variable. In the following definitions we admit to change the value of the variable v together some other neighborhood variables to obtain a notion of *Full Partial Substitutability (FPS).*

Definition 9.4.1 (Full Partial Substitutability (FPS)). *Consider two domain values b and a, for a variable v, and the set of constraint C; consider also a set of variable $V_1 \in V$. We say that b is* partially *substitutable for a on v with respect to a set of variables V_1 ($b \in FPS_v^{V_1}(a)$) if and only if for all assignment η there exists $\eta', \eta'' : V_1 \to D$ s.t.*

$$\bigotimes C\eta[\eta'][v := a] \leq_S \bigotimes C\eta[\eta''][v := b]$$

Similarly, all the notion of ${}^{\delta}/_{\alpha/\alpha-\mathrm{set}}$Neighborhood Partial Substitutability (${}^{\delta}/_{\alpha/\alpha-\mathrm{set}}NPS$) can be defined (just changing C with C_v). Notion of ${}^{\delta}/_{\alpha/\alpha-\mathrm{set}}$Full/Neighborhood Partial Interchangeability (${}^{\delta}/_{\alpha/\alpha-\mathrm{set}}FPI/NPI$) can be instead defined by considering the relation in both directions (and changing C with C_v for the neighborhood one).

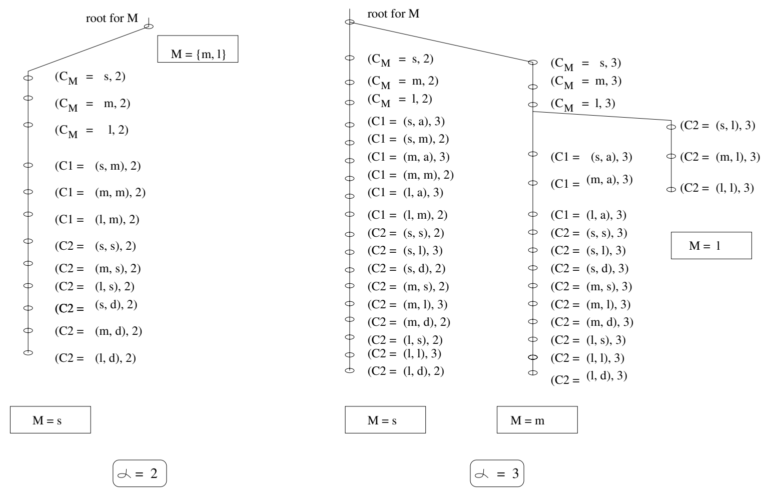


Fig. 9.9. Example of a search of α-interchangeability computing by the use of discrimination trees

Definition 9.4.2 ($^{\delta}$Neighborhood Partial Substitutability ($^{\delta}NPS$)). *Consider two domain values b and a, for a variable v, and the set of constraint C_v involving v; consider also a set of variable $V_1 \in V$. We say that b is $^{\delta}$Neighborhood Partial Substitutable for a on v with respect to a set of variables V_1 ($b \in {}^{\delta}FPS_v^{V_1}(a)$) if and only if for all assignment η there exists $\eta', \eta'' : V_1 \to D$ s.t.*

$$\bigotimes C_v\eta[\eta'][v := a] \times \delta \leq_S \bigotimes C_v\eta[\eta''][v := b]$$

Definition 9.4.3 ($_{\alpha}$Neighborhood Partial Substitutability ($_{\alpha}NPS$)). *Consider two domain values b and a, for a variable v, and the set of constraint C_v involving v; consider also a set of variable $V_1 \in V$. We say that b is $_{\alpha}$Neighborhood Partial Substitutable for a on v with respect to a set of variables V_1 ($b \in {}_{\alpha}FPS_v^{V_1}(a)$) if and only if for all assignment η there exists $\eta', \eta'' : V_1 \to D$ s.t.*

$$\bigotimes C_v\eta[\eta'][v := a] \geq_S \alpha \implies (\bigotimes C_v\eta[\eta'][v := a] \leq_S \bigotimes C_v\eta[\eta''][v := b])$$

Definition 9.4.4 ($_{\alpha-\mathrm{set}}$Neighborhood Partial Substitutability ($_{\alpha-\mathrm{set}}NPS$)). *Consider two domain values b and a, for a variable v, and the set of constraint C_v involving v; consider also a set of variable $V_1 \in V$. We say that b is $_{\alpha-\mathrm{set}}$Neighborhood Partial Substitutable for a on v with respect to a set of variables V_1 ($b \in {}_{\alpha-set}FPS_v^{V_1}(a)$) if and only if for all assignment η there exists $\eta', \eta'' : V_1 \to D$ s.t.*

$$\bigotimes C_v\eta[\eta'][v := a] \geq_S \alpha \implies \bigotimes C_v\eta[\eta''][v := b] \geq_S \alpha$$

Let's apply the definition of *NPI* to our running example in Figure 9.3, by projecting over variable x. It is easy to see that a and c are Neighbourhood Partial Interchangeable. In fact they have assigned both the semiring level 0.2. We have also that a, b and c are $_{0.15}NPI$ and $_{0.1-\mathrm{set}}NPI$.

The next theorem shows how *NI* is related to *NPI*. As we can imagine, interchangeability implies partial interchangeability.

Theorem 9.4.1. *Consider two domain values b and a, for a variable v, and the set of constraint C involving v; consider also a set of variable $V_1 \in V$ and its complement $\bar{V}_1 = V - V_1$. Then,*

$$NI_v(a/b) \implies NPI_v^{V_1}(a/b).$$

Proof. It is enough to show that $b \in NS_v(a) \implies b \in NPI_v^{V_1}(a)$ (the results for interchangeability easily follows from substitutability). By definition

$$b \in NS_v(a) \iff \bigotimes C_v\eta[v := a] \geq_S \bigotimes C_v\eta[v := b].$$

It is enough to take $\eta' = \eta'' = \emptyset$, to easily have

$$\bigotimes C\eta[\eta'][v := a] \leq_S \bigotimes C\eta[\eta''][v := b].$$

Similar results follow for the degradation and the threshold notion of partial interchangeability.

9.5 Conclusions

Interchangeability in CSPs is a general concept for formalizing and breaking symmetries. It has been proposed for improving search performance, for problem abstraction, and for solution adaptation. In this chapter, we have shown how the concept can be extended to soft CSPs in a way that maintains the attractive properties already known for hard constraints.

The two parameters α and δ allow us to express a wide range of practical situations. The threshold α is used to eliminate distinctions that would not interest us anyway, while the allowed degradation δ specifies how precisely we want to optimize our solution. We have shown a range of useful properties of these interchangeability concepts that should be useful for applying them in similar ways as interchangeability for hard constraints.

In fact, interchangeability may be practically more useful for soft constraints as it could be used to reduce the complexity of an optimization problem, which is often much harder to solve than a satisfaction problem. Furthermore, in the case of soft interchangeability it is possible to tune the parameters α and δ to create the levels of interchangeability that are required for the desired application.

10. SCSPs for Modelling Attacks to Security Protocols

Overview

Security protocols stipulate how remote principals of a computer network should interact in order to obtain specific security goals. The crucial goals of *confidentiality* and *authentication* may be achieved in various forms. Using soft (rather than crisp) constraints, we develop a uniform formal notion for the two goals. They are no longer formalised as mere yes/no properties as in the existing literature, but gain an extra parameter, the *security level.* For example, different messages can enjoy different levels of confidentiality, or a principal can achieve different levels of authentication with different principals.

The goals are formalised within a general framework for protocol analysis that is amenable to mechanisation by model checking. Following the application of the framework to analysing the asymmetric Needham-Schroeder protocol [18, 19], we have recently discovered a new attack on that protocol. We briefly comment on that attack, and demonstrate the framework on a bigger, largely deployed protocol consisting of three phases, Kerberos.

A number of applications ranging from electronic transactions over the Internet to banking transactions over financial networks make use of security protocols. It has been shown that the protocols often fail to meet their claimed goals [4, 137], so a number of approaches for analysing them formally have been developed [1, 3, 23, 56, 57, 101, 102, 136, 160]. The threats to the protocols come from malicious principals who manage to monitor the network traffic building fake messages at will. A major protocol goal is *confidentiality*, confirming that a message remains undisclosed to malicious principals. Another crucial goal is *authentication*, confirming a principal's participation in a protocol session. These goals are formalised in a mere "yes or no" fashion in the existing literature. One can just state whether a key is confidential or not, or whether a principal authenticates himself with another or not.

Security goals are not simple boolean properties. "Security is not a simple boolean predicate; it concerns how well a system performs certain functions" [8]. Indeed, experience shows that system security officers exercise care in applying any firm boolean statements to the real world even if they were formal. In general, formal security proofs are conducted within simplified models. Therefore, security officers attempt to bridge the gap between those models and the real word by adopting the largest possible variety of security measures all together. For example, firewalls accompany SSH connections. Limiting the access to certain ports of a server is both stated on the firewall *and* on the server itself. Biometric technology recently set aside the use of passwords to strengthen authentication levels of principals. Still, principals' credentials can be constrained within a validity time interval. The officer shall balance the cost of an extra security measure with his perception of the unmanaged risks. Any decision will only achieve a certain *security level*.

Security levels also characterise security patches [103]. Each patch in fact comes with a recommendation that is proportionate to the relevance of the security hole the patch is meant to fix. Patches may be critical, or recommended, or suggested, or software upgrade, etc. Depending on the cost of the patch and on the relevance of the hole, the security officer can decide whether or not to upgrade the system. It is a security policy what establishes the maximum level up until a patch can be ignored.

This all confirms that real-world security is based on security levels rather than on categorical, definitive, security assurances. In particular, security levels characterise the protocol goals of confidentiality and authentication. Focusing on the former goal, we remark that different messages require "specific degrees of protection against disclosure" [117]. For example, a user password requires higher protection than a *session key*, which is only used for a single protocol session. Intuitively, a password ought to be "more confidential" than a session key. Also, a confidentiality attack due to off-line cryptanalysis should not be imputed to the protocol design. Focusing on authentication, we observe that a certificate stating that K is a principal A's *public key* authenticates A very weakly. The certificate only signifies that A is a registered network principal, but in fact confers no guarantee about A's participation in a specific protocol session. A message signed by A's *private key* authenticates A more strongly, for it signifies that A participated in the protocol in order to sign the message.

Our original contributions. We have developed enriched formal notions for the two goals. Our definitions of *l-confidentiality* and of *l-authentication* highlight the security level l. One of the advantages of formalising security levels is to capture the real-world non-boolean concepts of confidentiality and authentication.

Each principal assigns his own security level to each message — different levels to different messages — expressing the principal's trust on the confidentiality of the message. So, we can formalise that different goals are granted to different principals. By a *preliminary analysis*, we can study what goals the protocol achieves in ideal conditions where no principal acts maliciously. An *empirical analysis* may follow, whereby we can study what goals the protocol achieves on a specific network configuration arising from the protocol execution in the real world. Another advantage of formalising security levels is that we can variously compare attacks — formally.

Our security levels belong to a finite linear order. Protocol messages can be combined (by concatenation or encryption) or broken down (by splitting or decryption) into new messages. We must be able to compute the security levels of the newly originated messages out of those of the message components. Therefore, we introduce a semiring whose career set is the set of security levels. Its two functions provide the necessary computational capabilities. Our use of a semiring is loosely inspired to Denning's use of a lattice to characterising secure flows of information through computer systems [87]. The idea of using levels to formalise *access rights* is in fact due to her. Denning signals an attack whenever an object is assigned a label worse than that initially specified. We formalise protocol attacks in the same spirit.

Another substantial contribution of the present work is the embedding of a novel *threat model* in a framework for protocol analysis. Our threat model regards *all principals as attackers if they perform,* either deliberately or not, *any operation that is not admitted by the protocol policy.* Crucially, it allows any number of non-colluding attackers. This overcomes the limits of Dolev and Yao's popular threat model [89], which reduces a number of colluding principals to a single attacker. The example that follows shows the deeper adherence of our threat model to the real world, where anyone may attempt to subvert a protocol for his (and only his) own sake.

Let us consider Lowe's popular attack on the asymmetric Needham-Schroeder protocol [136] within Dolev and Yao's threat model. It sees an attacker C masquerade as A with B, after A initiated a session with C. This scenario clearly contains an authentication attack following the confidentiality attack whereby C learns B's nonce Nb for A. Lowe reports that, if B is a bank for example, C can steal money from A's account as follows [136]

$$C \to B : \{\!|Na, Nb, \text{“Transfer \$ 1000 from } A\text{'s account to } C\text{'s”}|\!\}_{Kb}$$

where $\{\!|m|\!\}_K$ stands for the ciphertext obtained encrypting message m with key K (external brackets of concatenated messages are omitted). The bank B would honour the request believing it came from the account holder A.

We argue that the analysis is constrained by the limitations of the threat model. Plunging Lowe's scenario within our threat model highlights that B has mounted an indeliberate confidentiality attack on nonce Na, which was meant to be known to A and C only. As C did previously, B can equally decide to illegally exploit his knowledge of Na. If A is a bank, B can steal money from C's account as follows

$$B \to A : \{\!|Na, Nb, \text{“Transfer \$ 1000 from } C\text{'s account to } B\text{'s”}|\!\}_{Ka}$$

The bank A would honour the request believing it came from the account holder C.

The details of our findings on the Needham-Schroeder protocol can be found in Section 10.2. Our empirical analysis of the protocol uniformly detects *both* attacks in terms of decreased security levels: both C's security level on Nb and B's security level on Na become lower than they would be if C didn't act maliciously.

The framework presented throughout this chapter supersedes an existing kernel [18, 19] by extending it with five substantial features. I) The principles of the new threat model that allows all principals to behave maliciously. II) The combination of preliminary and empirical analyses. III) The study of the authentication goal. IV) The formalisation of an additional event whereby a principal discovers a secret by cryptanalysis — this allows a larger number of network configurations to be studied through an empirical analysis. V) A comprehensive study of how message manipulation and exposure to the network lowers the security level of the message — this is implemented by a new algorithm called RiskAssessment.

Since we only deal with bounded protocols and finite number of principals, our framework is amenable to mechanisation by model checking, although this exceeds the purposes of the present chapter.

Findings on the running example — Kerberos. We demonstrate our framework on a largely deployed protocol, Kerberos. Our preliminary analysis of the protocol formally highlights that the loss of an *authorisation key* would be more serious than the loss of a *service key* by showing that the former has a higher security level than the latter. By similar means, the preliminary analysis also allows us to compare the protocol goals in the forms they are granted to initiator and responder. It shows that authentication of the responder with the initiator is weaker than that of the initiator with the responder. To the best of our knowledge, developing such detailed observations formally is novel to the field of protocol analysis.

The empirical analysis that follows studies an example scenario in which a form of cryptanalysis was performed. The analysis highlights how that event lowers a number of security levels, and so lowers confidentiality and authentication for a number of principals.

Chapter outline. Our framework for protocol analysis is described in (§10.1). Then, the Needham'Schroeder (§10.2) and the Kerberos protocol are studied (§10.3) and analysed (§10.4). Some conclusions (§10.5) terminate the presentation.

10.1 Constraint Programming for Protocol Analysis

This section presents our framework for analysing security protocols. Using soft constraints requires the definition of a c-semiring.

Our *security semiring* (§10.1.1) is used to specify each principal's trust on the security of each message, that is each principal's *security level* on each message. The security levels range from the most secure (highest, greatest) level *unknown* to the least secure (lowest) level *public*. Intuitively, if A's security level on m is *unknown*, then no principal (included A) knows m according to A, and, if A's security level on m is *public*, then all principals potentially know m according to A. The lower A's security level on m, the higher the number of principals knowing m according to A. For simplicity, we state no relation between the granularity of the security levels and the number of principals.

Using the security semiring, we define the *network constraint system* (§10.1.2), which represents the computer network on which the security protocols can be executed. The development of the principals' security levels from manipulation of the messages seen during the protocol sessions can be formalised as a *security entailment* (§10.1.3), that is an entailment relation between constraints. Then, given a specific protocol to analyse, we represent its assumptions in the *initial SCSP* (§10.1.4). All admissible network configurations arising from the protocol execution as prescribed by the protocol designers can in turn be represented in the *policy SCSP* (§10.1.5). We also explain how to represent any network configuration arising from the protocol execution in the real world as an *imputable SCSP* (§10.1.7).

Given a security level l, establishing whether our definitions of *l-confidentiality* (§10.1.8) or *l-authentication* (§10.1.9) hold in an SCSP requires calculating the solution of the imputable SCSP and projecting it on certain principals of interest. The higher l, the stronger the goal. For example, *unknown-confidentiality* is stronger than *public-confidentiality*, or, A's security level on B's public key (learnt via a certification authority) being *public* enforces *public-authentication* of B with A, which is the weakest form of authentication. We can also formalise confidentiality or authentication attacks. The definitions are given within specific methodologies of analysis.

By a *preliminary analysis*, we can study what goals the protocol achieves in ideal conditions where no principal acts maliciously, namely the very best the protocol can guarantee. We concentrate on the policy SCSP, calculate its solution, and project it on a principal of interest. The process yields the principal's security levels, which allow us to study what goals the protocol grants to that principal in ideal conditions, and which potential attacks would be more serious than others for the principal. For example, the most serious confidentiality attacks would be against those messages on which the principal has the highest security level.

An *empirical analysis* may follow, whereby we can study what goals the protocol achieves on a specific network configuration arising from the protocol execution in the real world. We concentrate on the corresponding imputable SCSP, calculate its solution and project it on a principal of interest: we obtain the principal's security levels on all messages. Having done the same operations on the the policy SCSP, we can compare the outcomes. If some level in the imputable is lower than the corresponding level in the policy, then there is an attack in the imputable one. In fact, some malicious activity contributing to the network configuration modelled by the imputable SCSP has taken place so as to lower some of the security levels stated by the policy SCSP.

The following, general treatment is demonstrated in §10.3.

10.1.1 The Security Semiring

Let n be a natural number. We define the set L of *security levels* as follows.

$$L = \{\mathit{unknown},\ \mathit{private},\ \mathit{traded}_1,\ \mathit{traded}_2,\ \ldots\ ,\ \mathit{traded}_n,\ \mathit{public}\}$$

Although our security levels may appear to resemble Abadi's *types* [2], there is in fact little similarity. Abadi associates each message to either type *public*, or *secret*, or *any*, whereas we define n security levels with no bound on n, and *each principal associates a level of his own to each message* as explained in the following. Also, while Abadi's *public* and *private* cannot be compared, our levels are linearly ordered.

The security levels express each principal's trust on the security of each message. Clearly, *unknown* is the highest security level. We will show how, under a given protocol, a principal assigns *unknown* to all messages that do not pertain to the protocol, and to all messages that the principal does not know. A principal will assign *private* to all messages that, according to himself, are known to him alone, such as his own long-term keys, the nonces invented during the protocol execution, or any secrets discovered by cryptanalysis. In turn, a principal will assign $traded_i$ to the messages that are exchanged during the protocol: the higher the index i, the more the messages have been handled by the principals, and therefore the more principals have potentially learnt those messages. So, *public* is the lowest security level. These security levels generalise, by the $traded_i$ levels, the four levels that we have discussed elsewhere [18].

We introduce an additive operator, $+_{sec}$, and a multiplicative operator, $\times_{sec}$. To allow for a compact definition of the two operators, and to simplify the following treatment, let us define a convenient double naming:

- $unknown \equiv traded_{-1}$
- $private \equiv traded_0$
- $public \equiv traded_{n+1}$

Let us consider an index i and an index j both belonging to the closed interval $[-1,\ n+1]$ of integers. We define $+_{sec}$ and$\times_{sec}$ by the following axioms.

Ax. 1: $traded_i +_{sec} traded_j = traded_{min(i,j)}$
Ax. 2: $traded_i \times_{sec} traded_j = traded_{max(i,j)}$

Theorem 10.1.1 (Security Semiring). *The structure* $\mathcal{S}_{sec} = \langle L, +_{sec}, \times_{sec}, public, unknown \rangle$ *is a c-semiring.*

Proof. Clearly, $\mathcal{S}_{sec}$ enjoys the same properties as the structure $S_{finite-fuzzy} = \langle\{-1,\ldots,n+1\}, max, min, -1, n+1\rangle$. Indeed, the security levels can be mapped into the values in the range $-1,\ldots,n+1$ (*unknown* being mapped into 0, *public* being mapped into $n+1$); $+_{sec}$ can be mapped into function *max*; $\times_{sec}$ can be mapped into function *min*. Moreover, $S_{finite-fuzzy}$ can be proved a *c*-semiring as done with the fuzzy semiring [47].

10.1.2 The Network Constraint System

We define a constraint system $CS_n = \langle \mathcal{S}_{sec}, \mathcal{D}, \mathcal{V} \rangle$ where:

- $\mathcal{S}_{sec}$ is the security semiring (§10.1.1);
- $\mathcal{V}$ is bounded set of variables.
- $\mathcal{D}$ is an bounded set of values including the empty message $\{\!|\,|\!\}$ and all atomic messages, as well as all messages recursively obtained by concatenation and encryption.

We name CS_n as *network constraint system.* The elements of $\mathcal{V}$ stand for the network principals, and the elements of $\mathcal{D}$ represent all possible messages. Atomic messages typically are principal names, timestamps, nonces and cryptographic keys. Concatenation and encryption operations can be applied a bounded number of times.

Notice that CS_n does not depend on any protocols, for it merely portrays a computer network on which any protocol can be implemented. Members of $\mathcal{V}$ will be indicated by capital letters, while members of $\mathcal{D}$ will be in small letters.

10.1.3 Computing the Security Levels by Entailment

Recall that each principal associates his own security levels to the messages. Those levels evolve while the principal participates in the protocol and performs off-line operations such as encryption, concatenation, decryption, and splitting. We define four rules to compute the security levels that each principal gives to the newly generated messages. The rules are presented in Figure 10.1, where function *def* is associated to a generic constraint projected on a generic principal A.

Encryption and concatenation build up new messages from known ones. The new messages must not get a worse security level than the known ones have. So, the corresponding rules choose the better of the given levels. Precisely, if messages m_1 and m_2 have security levels v_1 and v_2 respectively, then the encrypted message $\{\!|m_1|\!\}_{m_2}$ and the compound message $\{\!|m_1, m_2|\!\}$, whose current level be some v_3, get a new level that is the better of v_1 and v_2, "normalised" by v_3. This

Encryption:

$$\frac{def(m_1) = v_1;\ def(m_2) = v_2;\ def(\{\!|m_1|\!\}_{m_2}) = v_3}{def(\{\!|m_1|\!\}_{m_2}) = (v_1 +_{sec} v_2) \times_{sec} v_3}$$

Concatenation:

$$\frac{def(m_1) = v_1;\ def(m_2) = v_2;\ def(\{\!|m_1, m_2|\!\}) = v_3;}{def(\{\!|m_1, m_2|\!\}) = (v_1 +_{sec} v_2) \times_{sec} v_3}$$

Decryption:

$$\frac{def(m_1) = v_1;\ def(m_2^{-1}) = v_2;\ def(\{\!|m_1|\!\}_{m_2}) = v_3;\ v_2, v_3 < unknown}{def(m_1) = v_1 \times_{sec} v_2 \times_{sec} v_3}$$

Splitting:

$$\frac{def(m_1) = v_1;\ def(m_2) = v_2;\ def(\{\!|m_1, m_2|\!\}) = v_3}{def(m_1) = v_1 \times_{sec} v_3;\ def(m_2) = v_2 \times_{sec} v_3}$$

Fig. 10.1. Computation rules for security levels

normalisation, which is done in terms of the $\times_{sec}$ operator, influences the result only if the new level is better than the current level.

Decryption and splitting break down known messages into new ones. The new messages must not get a better security level than the known ones have. So, the corresponding rules choose the worse of the given levels by suitable applications of $\times_{sec}$, and assign it to the new messages. Recall that, in case of asymmetric cryptography, the decryption key for a ciphertext is the inverse of the key that was used to create the ciphertext. So the rule for decryption considers the inverse of message m_2 and indicates it as m_2^{-1}. Conversely, in case of symmetric cryptography, we have $m_2^{-1} = m_2$. The rule for splitting presupposes that concatenation is transparent in the sense that, for any index n, an n-component message can be seen as a 2-component message, namely $\{\!|m_1, m_2, \ldots, m_n|\!\} = \{\!|m_1, \{\!|m_2, \ldots, m_n|\!\}|\!\}$. We now define a binary relation between constraints.

Definition 10.1.1 (Relation $\vdash$). *Consider two constraints $c_1, c_2 \in C$ such that $c_1 = \langle def_1, con \rangle$ and $c_2 = \langle def_2, con \rangle$. The binary relation $\vdash$ is such that $c_1 \vdash c_2$ iff def_2 can be obtained from def_1 by a number (possibly zero) of applications of the rules in Figure 10.1 .*

Theorem 10.1.2 (Relation $\vdash$ as entailment relation). *The binary relation $\vdash$ is an entailment relation.*

Proof (Hint). Relation $\vdash$ enjoys the reflexivity and transitivity properties that are needed to be an entailment relation.

In the following, $c^{\vdash}$ represents the reflexive, transitive closure of the entailment relation $\vdash$ applied to the constraint c. While other entailment relations (e.g. [52]) involve all constraints that are related by the partial order $\leq_S$, the security entailment only concerns the subset of those constraints obtainable by application of the four rules in Figure 10.1.

10.1.4 The Initial SCSP

The designer of a protocol must also develop a *policy* to accompany the protocol. The policy for a protocol $\mathcal{P}$ is a set of rules stating, among other things, the preconditions necessary for the protocol execution, such as which messages are public, and which messages are private for which principals.

It is intuitive to capture these policy rules by our security levels (§10.1.1). Precisely, these rules can be translated into unary constraints. For each principal $A \in \mathcal{V}$, we define a unary constraint that states A's security levels as follows. It associates security level *public* to those messages that are known to all, typically principal names and timestamps; level *private* to A's initial secrets, such as keys (e.g., A's long-term key if $\mathcal{P}$ uses symmetric cryptography, or A's private key if $\mathcal{P}$ uses asymmetric cryptography, or A's pin if $\mathcal{P}$ uses smart cards) or nonces; level *unknown* to all remaining domain values (including, e.g., the secrets that A will invent during the protocol execution, or other principals' initial secrets).

This procedure defines what we name *initial SCSP for* $\mathcal{P}$, which specifies the principals' security levels when no session of $\mathcal{P}$ has yet started. Notice that the constraint store representing each principal's security levels is computed using the reflexive, transitive, closure of the entailment relation (§10.1.3). So, when a new message is invented, the corresponding constraint is added to the store along with all constraints that can be extracted by entailment.

Considerations on how official protocol specifications often fail to provide a satisfactory policy [21] exceed the scope of this chapter. Nevertheless, having to define the initial SCSP for a protocol may help pinpoint unknown deficiencies or ambiguities in the policy.

10.1.5 The Policy SCSP

The policy for a protocol $\mathcal{P}$ also establishes which messages must be exchanged during a session between a pair of principals while no-one performs malicious activity. The protocol designer typically writes a single step as $A \rightarrow B : m$, meaning that principal A sends message m to principal B. The policy typically allows each principal to participate in a number of protocol sessions inventing a number of fresh messages. Assuming both these numbers to be bounded, a bounded number of *events* may take place [93]. Because no principal is assumed to be acting maliciously, no message is intercepted, so a message that is sent is certain to reach its intended recipient. Therefore, we only formalise the two following events.

1. A principal invents a fresh message (typically a new nonce).
2. A principal sends a message (constructed by some sequence of applications of encryption, concatenation, decryption, and splitting) to another principal, and the message is delivered correctly.

Clearly, additional events can be formalised to capture protocol-specific details, such as principal's annotation of sensitive messages, message broadcast, SSL-secure trasmission, and so on.

We read from the protocol policy each allowed step of the form $A \rightarrow B : m$ and its informal description, which explains whether A invents m or part of it. Then, we build the *policy SCSP for* $\mathcal{P}$ by the algorithm in Figure 10.2.

The algorithm considers the initial SCSP (line 1) and extends it with new constraints induced by each of the events occurring during the protocol execution (line 2). If the current event is a principal A's inventing a message n (line 3), then a unary constraint is added on variable A assigning security level *private* to the domain value n, and *unknown* to all other values (line 4). If that event is a principal A's sending a message m to a principal B (line 5), then the solution of the current SCSP p is computed and projected on the sender variable A (line 6), and extended by entailment (line 7). The last two steps yield A's view of the network traffic. In particular, also A's security level on m is updated by entailment. For example, if m is built as $\{\!|Na, Nb|\!\}$, the security levels of Na and Nb derive from the computed solution, and then the level of m is obtained by the concatenation rule of the entailment relation.

```
BuildPolicySCSP($\mathcal{P}$)
1.   p ← $\textit{initial SCSP for } \mathcal{P}$;
2.   for each event $ev$ allowed by the policy for $\mathcal{P}$ do
3.     if $ev = (A \textit{ invents } n$, for some $A$ and $n)$ then
4.       p ← p extended with unary constraint on $A$ that assigns
             $\textit{private}$ to $n$ and $\textit{unknown}$ to all other messages;
5.     if $ev = (A \textit{ sends } m \textit{ to } B \textit{ not intercepted}$, for some $A$, $m$ and $B)$ then
6.       $c \leftarrow Sol(\mathsf{p}) \Downarrow_{\{A\}}$;
7.       let $\langle def, con\rangle = c^{\vdash}$ in $newlevel \leftarrow$ RiskAssessment($def(m)$);
8.       p ← p extended with binary constraint between $A$ and $B$ that assigns
             $newlevel$ to $\langle \{\!|\,|\!\}, m\rangle$ and $unknown$ to all other tuples;
9.   return p;
```

Fig. 10.2. Algorithm to construct the policy SCSP for a protocol $\mathcal{P}$

At this stage, A's security level on m is updated again by algorithm RiskAssessment (line 7). As explained in the next section, this shall assess the risks that m runs following A's manipulation and the exposure to the network. The current SCSP can be now extended with a binary constraint on the pair of variables A and B (line 8). It assigns the newly computed security level *newlevel* to the tuple $\langle \{\!|\,|\!\}, m\rangle$ and *unknown* to all other tuples. This reasoning is repeated for each of the bounded number of events allowed by the policy. When there are no more events to process, the current SCSP is returned as policy SCSP for $\mathcal{P}$ (step 9), which is our formal model for the idealised protocol. Termination of the algorithm is guaranteed by finiteness of the number of allowed events. Its complexity is clearly linear in the number of allowed events, which is in turn exponential in the length of the exchanged messages [93].

10.1.6 Assessing the Expected Risk

Each network event involves some message. The events that expose their messages to the network, such as to send or receive or broadcast a message, clearly impose some *expected risk* on those messages — ideal message security is never to use that message. The *risk function* ρ expresses how the expected risk affects the security levels of the messages that are involved.

The actual definition of the risk function depends on the protocol policy, which should clearly state the expected risk for each network event when the protocol is executed in its intended environment. But "often protocols are used in environments other than the ones for which they were originally intended" [144], so the definition also depends on the specific environment that is considered.

The risk function should take as parameters the given security level and the network event that is influencing that level. The second parameter can be omitted for simplicity from this presentation because of the limited number of events we are modelling. Indeed, we will only have to compute the function for the network event whereby a message is sent on the network (either intercepted or not), whereas if we modelled, for example, a broadcast event, then the assessment for that particular event would have to yield *public*.

The risk function must enjoy the two following properties.

i. *Extensivity.* This property means that $\rho(l) \leq l$ for any l. It captures the requirement that each manipulation of a message decrease its security level — each manipulation increases the risk of tampering.
ii. *Monotonicity.* This property means that $l_1 \leq l_2$ implies $\rho(l_1) \leq \rho(l_2)$ for any l_1 and l_2. It captures the requirement that the expected risk preserve the $\leq$ relation between security levels.

Notice that we have stated no restrictions on the values of the risk function. Therefore, an initial total order, e.g. $l_1 < l_2$, may at times be preserved, such as $\rho(l_1) < \rho(l_2)$, or at other times be hidden, such as $\rho(l_1) = \rho(l_2)$.

As a simple example of risk function we choose the following variant of the predecessor function. It takes a security level and produces its predecessor in the linear order induced by $+_{sec}$ on the set L of security levels, unless the given level is the lowest, *public*, in which case the function leaves it unchanged. Our algorithm RiskAssessment in general serves to implement the risk function. Figure 10.3 shows the algorithm for our example function.

We remark that all considerations we advance in the sequel of this chapter merely rely on the two properties we have required for a risk function and are therefore independent from the specific example function. However, the protocol analyser may take, depending on his focus, more detailed risk functions, such as for checking originator(s) or recipient(s) of the current event (conventional principals, trusted third principals, proxi principals, etc.), the network where it is being performed (wired or wireless), and so on.

One could think of embedding the risk function at the constraint level rather than at a meta-level as we have done. That would be possible by embedding the appropriate refinements in the entailment rules. For example, let us consider an agent's construction of a message $m = \{\!|m_1, m_2|\!\}$, which is currently $traded_i$, from concatenation of m_1 and m_2, which are $traded_{i_1}$ and $traded_{i_2}$ respectively. The entailment rule should first compute the maximum between the levels of the components (that is the minimum between the indexes), obtaining $traded_{min(i_1,i_2)}$. Then, it should compute the minimum between the level just computed and that of m (that is the maximum between the indexes), obtaining $traded_{max(min(i_1,i_2),i)}$. Finally, the rule should apply the risk function. With our example risk function, it shold yield $traded_{max(min(i_1,i_2),i)+1}$. But the security levels would be decremented every time the entailment relation were applied.

```
RiskAssessment($l$)

1.    **let** $traded_i = l$ **in**
2.      **if** $i = n + 1$ **then** $l' \leftarrow l$
3.                        **else** $l' \leftarrow traded_{i+1}$;
4.    return $l'$;
```

Fig. 10.3. Implementation for a simple risk function

This would violate a general requirement of constraint programming, that is $c^{\vdash} = c^{\vdash^{\vdash}}$. Hence, the decrement at the meta level is preferable.

10.1.7 The Imputable SCSPs

A real-world network history induced by a protocol $\mathcal{P}$ must account for malicious activity by some principals. Each such history can be viewed as a sequence of events of four different forms.

1. A principal invents a fresh message (typically a new nonce).
2. A principal sends a message (constructed by some sequence of applications of encryption, concatenation, decryption, and splitting) to another principal, and the message is delivered correctly.
3. A principal sends a message (constructed as in the previous event) to another principal, but a third principal intercepts it.
4. A principal discovers a message by cryptanalysing another message.

Unlike the first two events, which were formalised also for constructing the policy SCSP, the last two are new, as they are outcome of malicious activity. We remark that the third event signifies that the message reaches some unexpected principal rather than its intended recipient.

We can model any network configuration at a certain point in any real-world network history as an SCSP by modifying the algorithm given in Figure 10.2 as in Figure 10.4 (unmodified fragments are omitted). The new algorithm takes as inputs a protocol $\mathcal{P}$ and a network configuration nc originated from the protocol execution. The third type of event is processed as follows: when a message is sent by A to B and is intercepted by another principal C, the corresponding constraint must be stated on the pair A, C rather than A, B. The fourth type of event is processed by stating a unary constraint that assigns *private* to the cryptanalyser's security level on the discovered message.

The new algorithm outputs what we name an *imputable SCSP* for $\mathcal{P}$. Both the initial SCSP and the policy SCSP may be viewed as imputable SCSPs. Because we have assumed all our objects to be bounded, the number of possible network configurations is bounded and so is the number of imputable SCSPs for $\mathcal{P}$.

10.1.8 Formalising Confidentiality

"*Confidentiality is the protection of information from disclosure to those not intended to receive it*" [156]. This definition is often simplified into one that is easier to formalise within Dolev-Yao's [89] model with a single attacker: a message is confidential if it is not known to the attacker. The latter definition is somewhat weaker: if a principal C who is not the attacker gets to learn a session key for A and B, the latter definition holds but the former does not. To capture the former definition, we adopt the following threat model: *all principals are attackers if they perform,* either deliberately or not*, any operation that is not admitted by the protocol policy.* As we have discussed in the introductory part of

```
BuildImputableSCSP($\mathcal{P}$, $nc$)
      ⋮
2.    for each event $ev$ in $nc$ do
        ⋮
8.1.    if $ev = (A$ sends $m$ to $B$ intercepted by $C$, for some $A, m, B$ and $C)$ then
8.2.      $c \leftarrow Sol(\mathsf{p}) \Downarrow_{\{A\}}$;
8.3.      let $\langle def, con\rangle = c^{\vdash}$ in $newlevel \leftarrow$ RiskAssessment($def(m)$);
8.4.      $\mathsf{p} \leftarrow \mathsf{p}$ extended with binary constraint betweeen $A$ and $C$ that assigns
              $newlevel$ to $\langle\{\!|\,|\!\}, m\rangle$ and $unknown$ to all other tuples;
8.5.    if $ev = (C$ cryptanalyses $n$ from $m$, for some $C$, $m$ and $n)$ then
8.6.      $\mathsf{p} \leftarrow \mathsf{p}$ extended with unary constraint on $C$ that assigns
              $private$ to $n$ and $unknown$ to all other messages;
        ⋮
```

Fig. 10.4. Algorithm to construct an imputable SCSP for $\mathcal{P}$ (fragment)

this chapter, our threat model exceeds the limits of Dolev-Yao's by allowing us to analyse scenarios with an unspecified number of non-colluding attackers.

A formal definition of confidentiality should account for the variety of requirements that can be stated by the protocol policy. For example, a message might be required to remain confidential during the early stages of a protocol but its loss during the late stages might be tolerated, as is the case with SET [21]. That protocol typically uses a fresh session key to transfer some certificate once, so the key loses its importance after the transfer terminates.

Another possible requirement is that certain messages, such as those signed by a *root certification authority* to associate the principals to their public keys [21], be entirely reliable. Hence, at least those messages must be assumed to be safe from cryptanalysis. Also, a protocol may give different guarantees about its goals to different principals [16], so our definition of confidentiality must depend on the specific principal that is considered.

Using the security levels, we develop uniform definitions of confidentiality and of confidentiality attack that account for any policy requirement. Intuitively, if a principal's security level on a message is l, then the message is *l-confidential* for the principal because the security level in fact formalises the principal's trust on the security, meant as confidentiality, of the message (see the beginning of §10.1). Thus, if an imputable SCSP features a principal with a lower security level on a message w.r.t. the corresponding level in the policy SCSP, then that imputable SCSP bears a *confidentiality attack*.

Here, l denotes a generic security level, m a generic message, A a generic principal. Also, P indicates the policy SCSP for a generic security protocol, and p and p' some imputable SCSPs for the same protocol. We define $Sol(\mathsf{P}) \Downarrow_{\{A\}} = \langle Def_A, \{A\}\rangle$, $Sol(\mathsf{p}) \Downarrow_{\{A\}} = \langle def_A, \{A\}\rangle$, and $Sol(\mathsf{p}') \Downarrow_{\{A\}} = \langle def'_A, \{A\}\rangle$.

Definition 10.1.2 (l-confidentiality). *l-confidentiality of m for A in $\mathsf{p} \iff def_A(m) = l$.*

Preliminary Analysis of Confidentiality. The preliminary analysis of the confidentiality goal can be conducted on the policy SCSP for the given protocol.

Let us calculate the solution of the policy SCSP, and project it on some principal A. Let us suppose that two messages m and m' get security levels l and l' respectively, $l' < l$. Thus, even if no principal acts maliciously, m' must be manipulated more than m, so A trusts that m' will be more at risk than m. We can conclude that the protocol achieves a *stronger confidentiality goal on* m *than on* m' even if it is executed in ideal conditions. Also, m may be used to encrypt m', as is the case with Kerberos (§10.4.1) for example. Therefore, losing m to a malicious principal would be more serious than losing m'. We address a principal's loss of m as *confidentiality attack on* m. A more formal definition of confidentiality attack cannot be given within the preliminary analysis because no malicious activity is formalised. So, the following definition concerns potential confidentiality attacks that may occur during the execution

Definition 10.1.3 (Potential, worse confidentiality attack). *Suppose that there is* l*-confidentiality of* m *in* P *for* A*, that there is* l'*-confidentiality of* m' *in* P *for* A*, and that* $l' < l$*; then,* a confidentiality attack on m would be worse than a confidentiality attack on m'.

Empirical Analysis of Confidentiality. By an empirical analysis, we consider a specific real-world scenario arising from the execution of a protocol and build the corresponding imputable SCSP p. If the imputable SCSP achieves a weaker confidentiality goal of some message for some principal than the policy SCSP does, then the principal has mounted, either deliberately or not, a confidentiality attack on the message.

Definition 10.1.4 (Confidentiality attack). *Confidentiality attack by* A *on* m *in* $\mathsf{p} \iff l$*-confidentiality of* m *in* P *for* $A \wedge l'$*-confidentiality of* m *in* p *for* $A \wedge l' < l$.

Therefore, there is a confidentiality attack by A on m in p iff $def_A(m) < Def_A(m)$. The more an attack lowers a security level, the worse that attack, so confidentiality attacks can be variously compared. For example, let us consider two confidentiality attacks by some agent on a message. If the message is l-confidential for the agent in the policy SCSP, but is l'-confidential and l''-confidential respectively in some imputable SCSPs p and p' for the same agent, then $l > l' > l''$ implies that the attack mounted in p' is worse than that in p. Likewise, let us consider two messages m and m' that are both l-confidential for some agent in the policy SCSP. If m is l'-confidential, and m' is l''-confidential in p, then $l > l' > l''$ implies that the attack mounted on m' is worse than that on m.

10.1.9 Formalising Authentication

The authentication goal enforces the principals' presence in the network and possibly their participation in specific protocol sessions. It is achieved by means of messages that "*speak about*" principals. For example, in a symmetric cryptography setting, given a session key Kab relative to the session between principals A and B and known to both, message $\{\!|A, Na|\!\}_{Kab}$ received by B informs him that A is running the session based on nonce Na and key Kab, namely the message authenticates A with B. An equivalent message in an asymmetric setting could be $\{\!|Nb|\!\}_{Ka^{-1}}$, which B can decrypt using A's public key. Also B's mere knowledge of Ka as being A's public key is a form of authentication of A with B. Indeed, A must be a legitimate principal because Ka is typically certified by a certificate of the form $\{\!|A, Ka|\!\}_{K_{ca}}$, K_{ca} being the public key of a certification authority. It follows that security protocols may use a large variety of message forms to achieve the authentication goal — the ISO standard in fact does not state a single form to use [123].

In consequence, we declare a predicate $speaksabout(m, A)$, but do not provide a formal definition for it because this would necessarily have to be restrictive. However, the examples above provide the intuition of its semantics. There is *l-authentication* of B with A if there exists a message such that A's security level on it is l, and the message speaks about B. This signifies that A received a message conveying B's aliveness.

Definition 10.1.5 (l-authentication). *l-authentication of B with A in* p $\iff \exists\ m\ s.t.\ def_A(m) = l < unknown \wedge speaksabout(m, B)\ \wedge\ def_B(m) < unknown.$

The definition says that there is l-authentication of B with A whenever both A and B's security levels on a message that speaks about B are less than unknown, l being A's level on the message. The intuition behind the definition is that messages that B sends A for authentication will produce a strong level of authentication if they reach A without anyone else's tampering. Otherwise the level of authentication gets weaker and weaker. Precisely, the lower A's security level on m, the weaker the authentication of B with A.

Weaker forms of authentication hold when, for example, B sends a message speaking about himself via a trusted third principal, or when a malicious principal overhears the message (recall that each event of sending decreases the security level of the sent message). Our definition applies uniformly to both circumstances by the appropriate security level.

Another observation is that the weakest form, *public*-authentication, holds for example of B with A in an asymmetric-cryptography setting by the certificate for B's public key in any imputable SCSP where A received the certificate. Likewise, the spy could always forge a *public* message that speaks about B, e.g. a message containing B's identity. But in fact *public*-authentication always holds between any pairs of principals because principals' names are known to all.

Preliminary Analysis of Authentication. As done with the confidentiality goal (§10.1.8), the preliminary analysis of the authentication goal can be conducted on the policy SCSP for the given protocol.

Once we calculate the solution of that SCSP, we can apply our definition of l-authentication, and verify what form of authentication is achieved. In particular, if there is l-authentication of B with A, and l'-authentication of D with C,

$l' < l$, then we can conclude that the protocol achieves a *stronger authentication goal of B with A, than of D with C*. We address a principal's masquerading as B with A as *authentication attack on A by means of B*. A more formal definition of authentication attack cannot be given at this stage, since no principal acts maliciously in the policy SCSP, However, we can compare potential authentication attacks in case they happen during the protocol execution.

Definition 10.1.6 (Potential, worse authentication attack). *Suppose that there is l-authentication of B with A by m in* P*, that there is l'-authentication of D with C by m' in* P*, and that $l' < l$; then* an authentication attack on A by means of B would be worse than an authentication attack on C by means of D.

Empirical Analysis of Authentication. If the policy SCSP P achieves l-authentication of B with A by m, and an imputable SCSP p achieves a weaker form of authentication between the same principals by the same message, then the latter SCSP bears an authentication attack.

Definition 10.1.7 (Authentication attack). *Authentication attack on A by means of B in* $\mathsf{p} \iff$ *l-authentication of B with A in* $\mathsf{P} \wedge$ *l'-authentication of B with A in* $\mathsf{p} \wedge l' < l$.

If a malicious principal has intercepted a message m that authenticates B with A, and forwarded m to B in some imputable SCSP p, then, according to the previous definition, there is an authentication attack on A by means of B in p.

10.2 An Empirical Analysis of Needham-Schroeder

Figure 10.5 presents the asymmetric Needham-Schroeder protocol, which is so popular that it requires little comments.

The goal of the protocol is *authentication*: at completion of a session initiated by A with B, A should get evidence to have communicated with B and, likewise, B should get evidence to have communicated with A. Assuming that encryption is perfect and that the nonces are truly random, authentication is achieved here by confidentiality of the nonces. Indeed, upon reception of Na inside message 2, A would conclude that she is interacting with B, the only principal who could retrieve Na from message 1. In the same fashion, when B receives Nb

1. $A \rightarrow B : \{\!|Na, A|\!\}_{Kb}$
2. $B \rightarrow A : \{\!|Na, Nb|\!\}_{Ka}$
3. $A \rightarrow B : \{\!|Nb|\!\}_{Kb}$

Fig. 10.5. The asymmetric Needham-Schroeder protocol

inside message 3, he would conclude that A was at the other end of the network because Nb must have been obtained from message 2, and no-one but A could perform this operation.

Lowe discovers [136] that the protocol suffers the attack in Figure 10.6, whereby a malicious principal C masquerades as a principal A with a principal B, after A initiated a session with C. The attack, which sees C interleave two sessions, indicates failure of the authentication of A with B, which follows from failure of the confidentiality of Nb. The security levels of all other principals on the nonces Na and Nb are *unknown*. So, by Definition 10.1.2, those nonces are *unknown*-confidential for any principal different from A or B.

An empirical analysis. We start off by building the initial SCSP, whose fragment for principals A and B is in Figure 10.7 (the following only features suitable SCSP fragments pertaining to the principals of interest).

Then, we build the policy SCSP for the protocol by BUILD_POLICY_SCSP. Figure 10.8 presents the fragment pertaining to a single session between principals A and B. The figure indicates that, while A's security level on her nonce Na was initially *private*, it is now lowered to $traded_1$ by entailment because of the binary constraint formalising step 2 of the protocol. Similarly, B's security level on Nb is now $traded_2$ though it was originally *private*. The figure omits the messages that are not relevant to the following discussion.

At this stage, we use BUILD_IMPUTABLE_SCSP to build the imputable SCSP given in Figure 10.9. It formalises the network configuration defined by Lowe's attack. The solution of this SCSP projected on variable C is a constraint that associates security level $traded_4$ to the nonce Nb. Following Definition 10.1.2,

1. $A \rightarrow C : \{\!|Na, A|\!\}_{Kc}$

1′. $C \rightarrow B : \{\!|Na, A|\!\}_{Kb}$

2′. $B \rightarrow A : \{\!|Na, Nb|\!\}_{Ka}$

2. $C \rightarrow A : \{\!|Na, Nb|\!\}_{Ka}$

3. $A \rightarrow C : \{\!|Nb|\!\}_{Kc}$

3′. $C \rightarrow B : \{\!|Nb|\!\}_{Kb}$

Fig. 10.6. Lowe's attack to the Needham-Schroeder Protocol

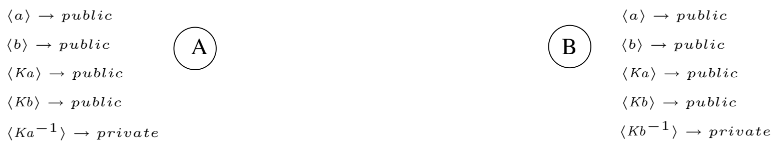

$\langle a \rangle \rightarrow public$
$\langle b \rangle \rightarrow public$
$\langle Ka \rangle \rightarrow public$
$\langle Kb \rangle \rightarrow public$
$\langle Ka^{-1} \rangle \rightarrow private$
A
B
$\langle a \rangle \rightarrow public$
$\langle b \rangle \rightarrow public$
$\langle Ka \rangle \rightarrow public$
$\langle Kb \rangle \rightarrow public$
$\langle Kb^{-1} \rangle \rightarrow private$

Fig. 10.7. Fragment of the initial SCSP for Needham-Schroeder protocol

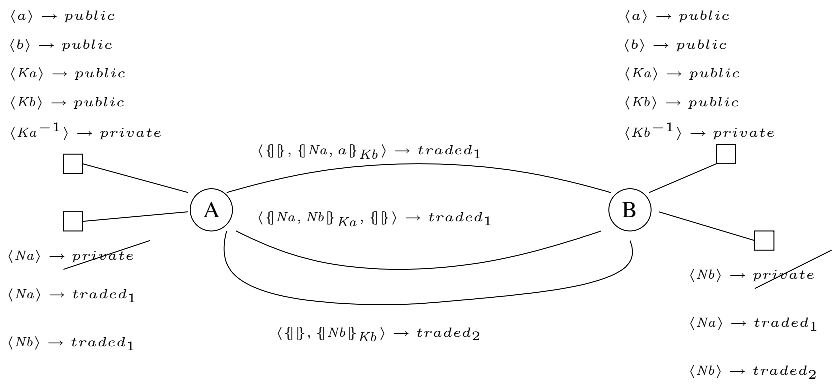


Fig. 10.8. Fragment of the policy SCSP for the Needham-Schroeder protocol

Nb is *traded*$_4$-confidential for C in this SCSP. Hence, there is a deliberate confidentiality attack by C on *Nb* in this problem, because *Nb* got level *unknown* in the policy SCSP. This leads to Lowe's attack.

We discover another attack in the same problem. The problem solution projected on variable B associates security level *traded*$_2$ to the nonce *Na*, which instead got level *unknown* in the policy SCSP. This signifies that B has learnt a nonce that he was not allowed to learn by policy, that there is an indeliberate confidentiality attack by B on *Na*. Notice that the two attacks are uniformly formalised.

As a consequence of the former attack, Lowe reports that, if B is a bank, C can steal money from A's account as follows

$$C \rightarrow B : \{\!|Na, Nb, \text{"Transfer£1000 from } A\text{'s account to } C\text{'s"}|\!\}_{Kb}$$

and the bank B would honour the request believing it came from the account holder A. As a consequence of the attack we have discovered, if A is a bank, B can steal money from C's account as follows

$$B \rightarrow A : \{\!|Na, Nb, \text{"Transfer£1000 from } C\text{'s account to } B\text{'s"}|\!\}_{Ka}$$

and the bank A would honour the request believing it came from the account holder C. In practice, it would be sufficient that B realises what *Na* is for the latter crime to succeed.

There are also less serious attacks. The nonce *Na* is *traded*$_3$-confidential for A in this SCSP, while it was *traded*$_1$-confidential in the policy SCSP. The discrepancy highlights that the nonce has been handled differently from the policy prescription — in fact C reused it with B. Also, *Nb*'s security level for A is *traded*$_3$ instead of *traded*$_1$ as in the policy SCSP. Similar considerations apply to *Nb*, whose security level for B is *traded*$_4$ instead of *traded*$_2$. This formalises C's abusive use of the nonce.

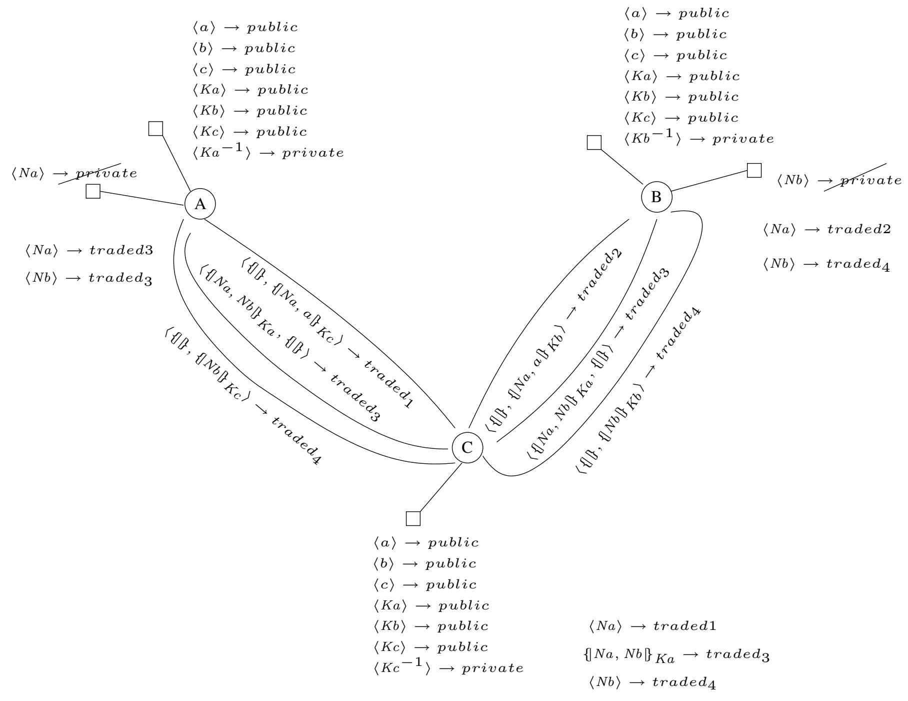

$\langle a \rangle \rightarrow public$
$\langle b \rangle \rightarrow public$
$\langle c \rangle \rightarrow public$
$\langle Ka \rangle \rightarrow public$
$\langle Kb \rangle \rightarrow public$
$\langle Kc \rangle \rightarrow public$
$\langle Ka^{-1} \rangle \rightarrow private$
$\langle Na \rangle \rightarrow$ ~~private~~
A
$\langle Na \rangle \rightarrow traded3$
$\langle Nb \rangle \rightarrow traded_3$
$\langle a \rangle \rightarrow public$
$\langle b \rangle \rightarrow public$
$\langle c \rangle \rightarrow public$
$\langle Ka \rangle \rightarrow public$
$\langle Kb \rangle \rightarrow public$
$\langle Kc \rangle \rightarrow public$
$\langle Kb^{-1} \rangle \rightarrow private$
B
$\langle Nb \rangle \rightarrow$ ~~private~~
$\langle Na \rangle \rightarrow traded2$
$\langle Nb \rangle \rightarrow traded_4$
$\langle \{\!|\!\}, \{\!|Na, a|\!\}_{Kc} \rangle \rightarrow traded_1$
$\langle \{\!|Na, Nb|\!\}_{Ka}, \{\!|\!\} \rangle \rightarrow traded_3$
$\langle \{\!|\!\}, \{\!|Nb|\!\}_{Kc} \rangle \rightarrow traded_4$
$\langle \{\!|\!\}, \{\!|Na, a|\!\}_{Kb} \rangle \rightarrow traded_2$
$\langle \{\!|Na, Nb|\!\}_{Ka}, \{\!|\!\} \rangle \rightarrow traded_3$
$\langle \{\!|\!\}, \{\!|Nb|\!\}_{Kb} \rangle \rightarrow traded_4$
C
$\langle a \rangle \rightarrow public$
$\langle b \rangle \rightarrow public$
$\langle c \rangle \rightarrow public$
$\langle Ka \rangle \rightarrow public$
$\langle Kb \rangle \rightarrow public$
$\langle Kc \rangle \rightarrow public$
$\langle Kc^{-1} \rangle \rightarrow private$
$\langle Na \rangle \rightarrow traded1$
$\{\!|Na, Nb|\!\}_{Ka} \rightarrow traded_3$
$\langle Nb \rangle \rightarrow traded_4$

Fig. 10.9. Fragment of the Imputable SCSP corresponding to Lowe's attack

10.3 The Kerberos Protocol

Kerberos is a protocol based on symmetric cryptography meant to distribute session keys with authentication over local area networks. The protocol has been developed in several variants (e.g. [146]), and also integrated with smart cards [124]. Here, we refer to the version by Bella and Riccobene [23].

The layout in Figure 10.10 shows that Kerberos relies on two servers, the *Kerberos Authentication Server* (Kas in brief), and the *Ticket Granting Server* (Tgs in brief). The two servers are trusted, namely they are assumed to be secure from the spy's tampering. They have access to an internal database containing the long-term keys of all principals. The database is in turn assumed to be secure. Only the first two steps of the protocol are mandatory, corresponding to a principal A's authentication with Kas. The remaining steps are optional as they are executed only when A requires access to a network resource B.

In the authentication phase, the initiator A queries Kas with her identity, Tgs and a timestamp T_1; Kas invents a session key and looks up A's shared key in the database. It replies with a message sealed by A's shared key containing the session key, its timestamp Ta, Tgs and a ticket. The session key and the ticket are the credentials to use in the subsequent authorisation phase, so we address them as *authkey* and *authticket* respectively.

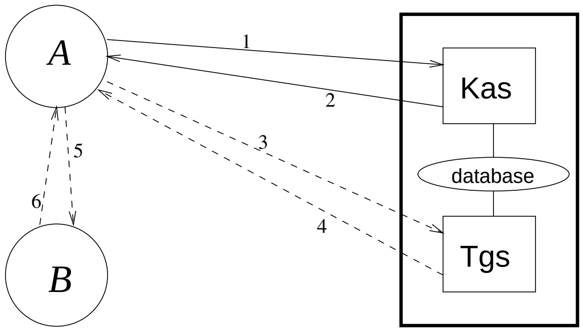


Fig. 10.10. The Kerberos layout

Authentication

1. $A \rightarrow \mathsf{Kas} : A, \mathsf{Tgs}, T_1$
2. $\mathsf{Kas} \rightarrow A : \{\!|authK, \mathsf{Tgs}, Ta, \underbrace{\{\!|A, \mathsf{Tgs}, authK, Ta|\!\}_{Ktgs}}_{authTicket}|\!\}_{Ka}$

Authorisation

3. $A \rightarrow \mathsf{Tgs} : \overbrace{\{\!|A, \mathsf{Tgs}, authK, Ta|\!\}_{Ktgs}}^{authTicket}, \overbrace{\{\!|A, T_2|\!\}_{authK}}^{authenticator1}, B$
4. $\mathsf{Tgs} \rightarrow A : \{\!|servK, B, Ts, \underbrace{\{\!|A, B, servK, Ts|\!\}_{Kb}}_{servTicket}|\!\}_{authK}$

Service

5. $A \rightarrow B : \overbrace{\{\!|A, B, servK, Ts|\!\}_{Kb}}^{servTicket}, \overbrace{\{\!|A, T_3|\!\}_{servK}}^{authenticator2}$
6. $B \rightarrow A : \underbrace{\{\!|T_3 + 1|\!\}_{servK}}_{authenticator3}$

Fig. 10.11. The Kerberos protocol

Now, A may start the authorisation phase. She sends Tgs a three-component message including the authticket, an authenticator sealed by the authkey containing her identity and a new timestamp T_2, and B's identity. The lifetime of an authenticator is a few minutes. Upon reception of the message, Tgs decrypts the authticket, extracts the authkey and checks the validity of its timestamp Ta, namely that Ta is not too old with respect to the lifetime of authkeys. Then, Tgs decrypts the authenticator using the authkey and checks the validity of T_2 with respect to the lifetime of authenticators. Finally, Tgs invents a new session key and looks up B's shared key in the database. It replies with a message sealed by the authkey containing the new session key, its timestamp Ts, B and a ticket. The session key and the ticket are the credentials to use in the subsequent service phase, so we address them as *servkey* and *servticket* respectively. The lifetime of a servkey is a few minutes.

Hence, A may start the service phase. She sends B a two-component message including the servticket and an authenticator sealed by the servkey containing her identity and a new timestamp T_3. Upon reception of the message, B decrypts the servticket, extracts the servkey and checks the validity of its timestamp Ts. Then, B decrypts the authenticator using the servkey and checks the validity of T_3. Finally, B increments T_3, seals it by the servkey and sends it back to A.

10.4 Analysing Kerberos

As a start, we build the initial SCSP for Kerberos. Figure 10.12 shows the fragment pertaining to principals A and B. The assignment *allkeys* $\rightarrow$ *private* signifies that the constraint assigns level *private* to all principals' long-term keys.

Then, we build the policy SCSP for Kerberos using algorithm BuildPolicySCSP (Figure 10.2). Figure 10.13 shows the fragment pertaining to principals A and B. The components that are specific of the session between A and B, such as timestamps and session keys, are not indexed for simplicity. We remark that the security levels of all other principals on the authkey *authK* and on the servkey *servK* are *unknown*.

10.4.1 Confidentiality

The preliminary analysis of confidentiality conducted on the policy SCSP in Figure 10.13 highlights that the late protocol messages get worse security levels than the initial ones do. For example, by definition 10.1.2, there is $traded_3$-confidentiality of *servK* for B. By the same definition, it is crucial to observe that A gets *authK* as $traded_1$-confidential, but gets *servK* as $traded_3$-confidential. So, if we consider a potential confidentiality attack whereby A looses *authK* to some malicious principal other than B, and another potential confidentiality attack

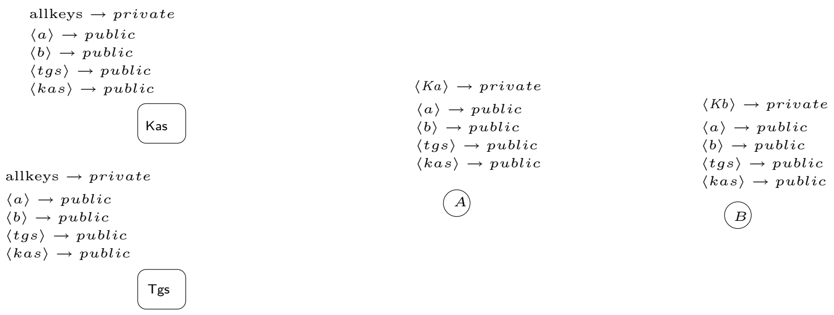


Fig. 10.12. The initial SCSP for Kerberos (fragment)

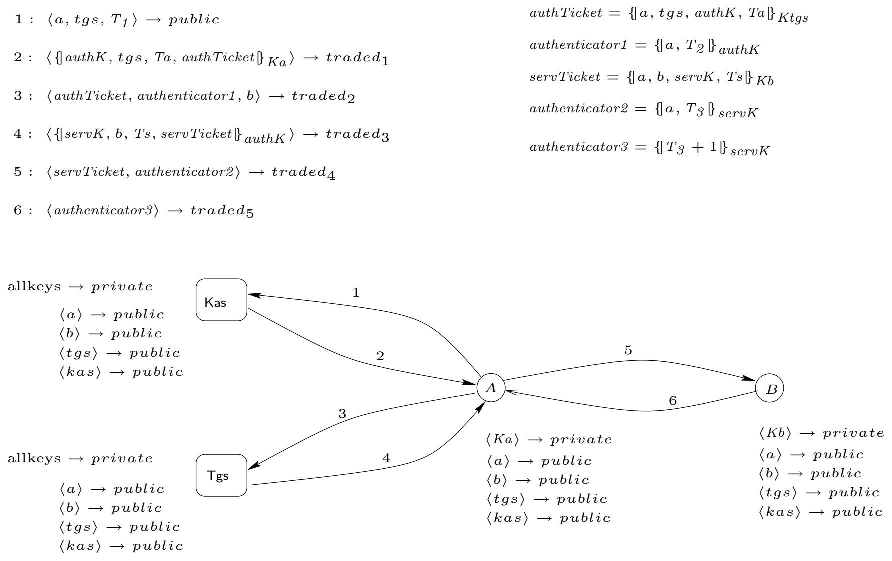

1 : ⟨a, tgs, T1⟩ → public
2 : ⟨{|authK, tgs, Ta, authTicket|}Ka⟩ → traded1
3 : ⟨authTicket, authenticator1, b⟩ → traded2
4 : ⟨{|servK, b, Ts, servTicket|}authK⟩ → traded3
5 : ⟨servTicket, authenticator2⟩ → traded4
6 : ⟨authenticator3⟩ → traded5
authTicket = {|a, tgs, authK, Ta|}Ktgs
authenticator1 = {|a, T2|}authK
servTicket = {|a, b, servK, Ts|}Kb
authenticator2 = {|a, T3|}servK
authenticator3 = {|T3 + 1|}servK
allkeys → private
⟨a⟩ → public
⟨b⟩ → public
⟨tgs⟩ → public
⟨kas⟩ → public
Kas
Tgs
A
B
1
2
3
4
5
6
⟨Ka⟩ → private
⟨Kb⟩ → private

Fig. 10.13. The policy SCSP for Kerberos (fragment)

whereby A or B loose *servK* to some malicious principal, the former would be a worse confidentiality attack than the latter, by definition 10.1.3. Indeed, having *authK* available, one can obtain *servK* from decryption and splitting of message 4.

We also conduct an empirical analysis of confidentiality by considering, as example a *known-ciphertext attack* [185] mounted by some malicious principal C on the authenticator of message 3 to discover the authkey pertaining to a principal A (and Tgs). We briefly remind how such an attack works. Since both principal names and timestamps are public, C knows the body of the authenticator with a good approximation — she should just try out all timestamps of, say, the last day. First, she invents a key, encrypts the known body with it, and checks whether the result matches the encrypted authenticator fetched from the network. If not, C “refines” her key [185] and iterates the procedure until she obtains the same ciphertext as the authenticator. At this stage, she holds the encryption key, alias the authkey, because encryption is injective. The entire tampering took place off line.

Along with the authkey for A, principal C also saves a copy of the corresponding authticket by splitting message 3 into its components. Then, C forwards message 3, unaltered, to Tgs, so A can continue and terminate the session accessing some resource B. A glimpse to Figure 10.11 shows that C is now in a position to conduct, for the lifetime of the authkey, the Authorisation and Service phases while he masquerades as A with some principal D. To do so, C forges an instance $3'$ of message 3 by using the authticket just learnt, by refreshing the timestamp inside the authenticator (which he can do because he knows the authkey), and by mentioning the chosen principal D. As Tgs believes that the message comes from A, Tgs replies to A with a message $4'$ containing some fresh servkey meant for A and D. Having intercepted $4'$, C learns the servkey and therefore can forge an instance $5'$ for D of message 5. Finally, C intercepts $6'$ and the session terminates without A's participation.

Our algorithm BuildImputableSCSP executed on the network configuration just described produces the imputable SCSP in Figure 10.14. The SCSP omits the constraint corresponding to the Authentication phase between A and Kas. Because C intercepts message 3, constraint 3 is stated between A and C. Projecting that constraint on C, we have that C's security level on message $authTicket, authenticator1, b$ is $traded_2$. By splitting this message, C discovers the authticket, so the entailment relation states a unary constraint on C assigning $traded_2$ to $authTicket$. Another unary constraint on C assigns $private$ to $authK$, which is found by cryptanalysis.

Constraint $\bar{3}$ between C and Tgs assigns $traded_3$ to message 3 because of C's rerouting. Projecting that constraint on Tgs, we have by entailment that Tgs's security level on $authK$ goes down to $traded_3$, whereas it was $traded_2$ in the policy SCSP. Constraint 4 formalises Tgs's reply to A, while the constraints for the rest of the session between A and B are omitted. Constraints $3'$, $4'$, $5'$, and $6'$ formalise the session between C, Tgs, and D.

At this stage, we can conduct an empirical analysis of confidentiality for each of the agents involved in this imputable SCSP. By definition 10.1.2, $authTicket$, $authenticator1'$, and $authK$ are each $traded_3$-confidential for Tgs in this problem. Since they were $traded_2$-confidential in the policy SCSP, we conclude by definition 10.1.4 that there is a confidentiality attack by Tgs on each of these messages in the imputable SCSP considered here. The attacks signal C's manipulation of messsage 3.

$3 : \langle authTicket, authenticator1, b\rangle \rightarrow traded_2$ $\qquad authTicket = \{\!|a, tgs, authK, Ta|\!\}_{Ktgs}$

$\bar{3} : \langle authTicket, authenticator1, b\rangle \rightarrow traded_3$ $\qquad authenticator1 = \{\!|a, T_2|\!\}_{authK}$

$4 : \langle \{\!|servK, b, Ts, servTicket|\!\}_{authK}\rangle \rightarrow traded_3$ $\qquad servTicket = \{\!|a, b, servK, Ts|\!\}_{Kb}$

$3' : \langle authTicket, authenticator1', d\rangle \rightarrow traded_3$ $\qquad servTicket' = \{\!|a, d, servK', Ts'|\!\}_{Kd}$

$4' : \langle \{\!|servK', d, Ts', servTicket'|\!\}_{authK}\rangle \rightarrow traded_4$ $\qquad authenticator1' = \{\!|a, T_2'|\!\}_{authK}$

$5' : \langle servTicket', authenticator2'\rangle \rightarrow traded_5$ $\qquad authenticator2' = \{\!|a, T_3'|\!\}_{servK'}$

$6' : \langle authenticator3'\rangle \rightarrow traded_6$ $\qquad authenticator3' = \{\!|T_3' + 1|\!\}_{servK'}$

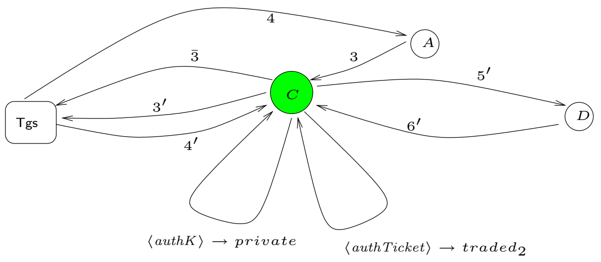

4
$\bar{3}$
3
A
C
$5'$
$3'$
Tgs
D
$6'$
$4'$
$\langle authK\rangle \rightarrow private$
$\langle authTicket\rangle \rightarrow traded_2$

Fig. 10.14. An imputable SCSP for Kerberos (fragment)

The imputable SCSP also achieves *private*-confidentiality of *authK* for C, whereas the policy SCSP achieved *unknown*-confidentiality of *authK* for C. Therefore, there is a confidentiality attack by C on *authK* in this SCSP. Likewise, there is a confidentiality attack by C on *authTicket*. From constraint $4'$ we have by entailment that C's security level on $servTicket'$ and on $servK'$ is $traded_4$ rather than *unknown* as in the policy SCSP, hence we find other confidentiality attacks by C on each of these messages.

There are also confidentiality attacks by D, who gets $servTicket'$, $authenticator2'$, and $servK'$ as $traded_5$, rather than $traded_4$.

10.4.2 Authentication

We now focus on the fragment of policy SCSP for Kerberos given in Figure 10.13 to conduct the preliminary analysis of the authentication goal.

By definition 10.1.5, there is $traded_2$-authentication of A with Tgs in the policy SCSP. The definition holds for message 3, whose first two components speak about A. Also, there is $traded_4$-authentication of A with B thanks to message 5, and $traded_5$-authentication of B with A due to message 6. While it is obvious that message 5 speaks about A, it is less obvious that message 6 speaks about B. This is due to the use of a servkey that is associated to B.

We observe that authentication of B with A is weaker than authentication of A with B even in the ideal conditions formalised by the policy SCSP. Intuitively, this is due to the fact that the servkey has been handled both by A and B rather than just by A. Hence, by definition 10.1.6, a principal C's masquerading as A with B would be a worse authentication attack than a principal D's masquerading as B with A.

An empirical analysis of authentication can be conducted on the imputable SCSP in Figure 10.14. That SCSP achieves $traded_5$-authentication of A with B thanks to message 5, and $traded_6$-authentication of B with A due to message 6. Comparing these properties with the equivalent ones holding in the policy SCSP, which we have seen above, we can conclude by definition 10.1.7 that the imputable SCSP considered hides an authentication attack on B by means of A, and an authentication attack on A by means of B. They are due to C's interception of message 3, which has lowered the legitimate protocol participants' security levels on the subsequent messages.

It is important to emphasize that these authentication attacks could not be captured by an equivalent definition of authentication based on crisp, rather than soft, constraints. The definition in fact holds in the policy SCSP as well as in the imputable SCSP. What differentiates the two SCSPs is merely the security level characterising the goal.

10.5 Conclusions

We have developed a new framework for analysing security protocols, based on a recent kernel [18,19]. Soft constraint programming allows us to conduct a fine analysis of the confidentiality and authentication goals that a protocol attempts to achieve. Using the security levels, we can formally claim that a configuration induced by a protocol achieves a certain level of confidentiality or authentication. That configuration may be ideal if every principal behaves according to the protocol, as formalised by the policy SCSP; or, it may arise from the protocol execution in the real world, where some principal may have acted maliciously, as formalised by an imputable SCSP. We can formally express that different principals participating in the same protocol session obtain different forms of those goals. We might even compare the forms of the same goal as achieved by different protocols.

Our new threat model where each principal is a potential attacker working for his own sake has allowed us to detect a novel attack on the asymmetric Needham-Schroeder protocol. Once C masquerades as A with B, agent B indeliberately gets hold of a nonce that was not meant for him. At this stage, B might decide to exploit this extra knowledge, and begin to act maliciously. Our imputable SCSP modelling the scenario reveals that B's security level on the nonce is lower than that allowed by the policy.

There is some work related to our analysis of Kerberos, such as Mitchell et al.'s analysis by model checking [147]. They consider a version of Kerberos simplified of timestamps and lifetimes — hence authkeys and servkeys cannot be distinguished — and establish that a small system with an initiator, a responder, Kas and Tgs keeps the two session keys secure from the spy. Bella and Paulson [22] verify by theorem proving a version with timestamps of the same protocol. They do prove that using a lost authkey will let the spy obtain a servkey. On top of this, one can informally deduce that the first key is more important than the second in terms of confidentiality. By contrast, our preliminary analysis of the protocol states formally that the authkey is $traded_1$-confidential and the servkey is $traded_3$-confidential (§10.4.1). Another finding is the difference between authentication of initiator with responder and vice versa (§10.4.2).

Some recent research exists that is loosely related to ours. Millen and Shamatikov [145] map the existence of a *strand* representing the attack upon a constraint problem. Comon *et al.* [74], and Amadio and Charatonik [7] solve confidentiality and reachability using *Set-Based Constraint* [157]. By constrast, we build suitable constraint problems for the analysis of a global network configuration where any principals (not just one) can behave maliciously. In doing so, we also analyse the safety of the system in terms of the consequences of a deliberate attack on the environment. The idea of *refinements* [176] is also somewhat related to our use of levels. In that case the original protocol must be specialised in order to be able to map the known/unknown level over the set of levels specified by the policy. The policy have also to specify how the levels have to be changed w.r.t. each operation described in the protocol. Abstract interpretation techniques (much in the spirit of those used by Bistarelli et al. [36]) can be used as a next step to deal with unbounded participants/sessions/messages.

While mechanical analysis was outside our aims, we have implementated a mechanical checker for l-confidentiality on top of the existing *Constraint Handling Rule* (CHR) framework [42]. For example, when we input the policy SCSP for the Needham-Schroeder protocol and the imputable SCSP corresponding to Lowe's attack, the checker outputs

```
checking(agent(a))
checking(agent(b))
   attack(n_a, policy_level(unknown), attack_level(traded_1))
checking(agent(c))
   attack(enk(k(a),pair(n_a,n_b)), policy_level(unknown),
                                   attack_level(traded_1))
   attack(n_b, policy_level(unknown), attack_level(traded1))
```

The syntax seems to be self-explanatory. Line two reveals the new attack we have found on B, who has lowered his security level on Na from *unknown* to $traded_1$. Likewise, line three denounces that not only has C got hold of the nonce Nb but also of the message $\{\!|Na, Nb|\!\}_{Ka}$ (which was meant for A and not for B) that contains it.

11. Conclusions and Directions for Future Work

You'll never know how well you're made until all you had is gone.
Then the good will come out and you'll be free.
. . . a friend of mine . . .

In this book we explored and added softness to the constraint solving and programming framework. We have been involved in several research topics; in the following points we summarize the main results of the book.

11.1 Summary and Main Results

In Chapter 2 we have proposed a general framework for constraint solving where each constraint allows each tuple with a certain level of confidence (or degree, or cost, or other). This allows for a more realistic modelization of real-life problems, but requires a new constraint solving engine that has to take such levels into account. To do this, we used the notion of semiring, which provides both the levels and the new constraint combination operations. Different instantiations of the semiring set and operations give rise to different existing frameworks (Fuzzy CSP, Probabilistic, Classical, VCSP, etc.).

In Chapter 3 we considered the issue of local consistency in such an extended framework, and we provided sufficient conditions that assure these algorithms to work. Then, we considered dynamic programming-like algorithms, and we proved that these algorithms can always be applied to SCSPs (and have a linear time complexity when the given SCSPs can be provided with a parsing tree of bounded size).

We have studied some possible uses of Soft Arc Consistency (SAC) over SCSPs, which could make the search for the best solution faster. In particular, we have analyzed the relationship between AC and SAC. Then we have studied the relationship between the soft constraint formalism and the constraint propagation schema proposed in [11, 12]. What we have discovered is that the GI algorithm of [11, 12] can also be used for soft constraints, since soft constraints provide what is needed for the GI algorithm to work correctly: a partial order with a bottom, and a set of monotone functions.

Moreover, in studying this relationship we have also discovered that, in soft constraints, we do not have to restrict ourselves to local consistency functions which solve a subproblem, but we can use *any* monotone function. Of course, in this more general case, the equivalence of the resulting problem and the initial one is not assured any more, and has to be studied on a case-by-case basis.

By passing from classical constraints to soft constraints, although on finite domains, we have more possible sources of non-termination for the GI algorithm, since the semiring can be infinite. Therefore, we have studied in depth the issue of termination for GI over soft constraints, finding some convenient sufficient conditions. In particular, one of them just requires checking, in a certain special case, that the multiplicative operation of the semiring is idempotent. We show that indeed it is possible to treat in a uniform way hard and soft constraints.

In Chapter 4 we have proposed an abstraction scheme for abstracting soft constraint problems, with the goal of finding an optimal solution in a shorter time. The main idea is to work on the abstract version of the problem and then bring back some useful information to the concrete one, to make it easier to solve. We have also shown how to use it to import propagation rules from the abstract setting to the concrete one.

In Chapter 5 we have described how the semiring framework can be used to both embed the constraint structure and topology in a suitable semiring of functions. The introduction of the functions enriches the expressivity of the framework and gives the possibility to express in a general way dynamic programming and general local consistency techniques.

In Chapter 6 we have introduced a framework for constraint programming over semirings. This allows us to use a CLP-like language for both constraint solving and optimization. In fact, constraint systems based on semirings are able to model both classical constraint solving and also more sophisticated features like uncertainty, probability, fuzziness, and optimization.

We have then given this class of languages three equivalent semantics: model-theoretic, fix-point, and proof-theoretic, in the style of classical CLP programs. Finally, we have obtained interesting decidability results for general SCLP programs and also for those SCLP programs without functions.

In Chapter 7 we have investigated the relationship between shortest path problems and constraint programming, proposing the soft constraint logic programming framework to model and solve many versions of this problem in a declarative way. In particular, both classical, multi-criteria, partially-ordered, and modality-based SP problems are shown to be expressible in this programming formalism. We also have shown that certain algorithms that solve SP problems can be used to efficiently compute the semantics of a certain class of SCLP programs.

In Chapter 8 a concurrent language for programming with soft constraints is discussed. The new language extends the concurrent constraints (cc) language in two ways: first it is able to deal with soft constraints instead of the crisp ones; to do this the notions of “entailment” and of “consistency” have been suitably changed. Second, the syntax of the language is extended with the notion of thresholds; using such a notion we showed how branch of the computation tree can be removed in advance narrowing the total search space.

In Chapter 9 we have discussed the notion of Substitutability and Interchangeability for soft constraints. Degradation and threshold are introduced to increase the number of interchangeabilities in a problem. A study of the complexity of some new algorithms to be used to detect when two domain value are interchangeable is given.

In Chapter 10 we have shown how to use SCSPs for the definition of network of agents executing a security protocol. Some security properties of the network are translated into specific properties of the SCSP solution and in particular the notion of inconsistency is mapped upon the notion of network insecurity.

11.2 Directions for Future Research

The introduction of a general framework to deal with soft constraints could give suggestions for new research topics, only partially studied up now. Here are some research directions in already explored fields and also in new ones:

11.2.1 Abstraction

Future work could concern the comparison between our properties and the one of [169], the generalization of our scheme to also abstract domains and graph topologies, and an experimental phase to assess the practical value of our proposal. Moreover, an experimental phase may be necessary to check the real practical value of our proposal. A possibility could be to perform such a phase within the `clp(fd,S)` system developed at INRIA [112], which can already solve soft constraints in the classical way (branch-and-bound plus propagation via partial arc-consistency).

Interesting would be also to investigate the relationship of our notion of abstraction with several other existing notions currently used in constraint solving. For example, it seems to us that many versions of the intelligent backtracking search could be easily modeled via soft constraints, by associating some information about the variables responsible for the failure to each constraint. Then, it should be possible to define suitable abstractions between the more complex of these frameworks and the simpler ones.

11.2.2 High Order Semirings

Future work could try to study the shapes of SCSPs by starting from their function representation. Moreover, the possibility to also use partially instantiated functions in the SCSP framework could make it possible to represent and solve *parametric problems.*

11.2.3 SCLP

To make SCLP a practical programming paradigm, some efficient techniques to implement their operational semantics are needed. This would require considering variants of branch and bound methods, and developing intelligent ways to recognize and cut useless search branches. In this respect, a possibility could be to study the usability of local consistency techniques [47] for a better approximation of the semiring values to be associated to each value combination in order to obtain better bounds for the search.

Other area of research could be the study of the relationship between semiring-based arc-consistency and classical arc-consistency, since in some cases by using classical arc-consistency one loses some information but gains in speedup.

11.2.4 SCLP for Operational Research Problems

Using structures more complex than semirings, or by considering trees instead of paths, we could also give semantics using OR algorithms to a greater class of SCLP programs. In particular, it could be possible to use SCLP programs to model and solve best tree problems. This methodology could be used to specify and solve planning problems, since in many cases such problems can be cast as best tree problems over the graph describing all possible moves of the planning scenario.

11.2.5 Soft Concurrent Constraints

We see soft cc as a first step towards the possibility of using high level declarative languages for Web programming. Of course there are many more aspects to consider to make the language rich enough to be practically usable. However, soft constraints have already shown their usefulness in describing security protocols (see [18, 20]) and integrity policies (see [40, 41]). We are already considering the introduction of some other features in the language. We are also considering the possibility of adding soft cc primitives inside other concurrent frameworks, such as Klaim [82].

11.2.6 Soft Constraints for Security

At this stage, integrating our framework with model-checking tools appears to be a straightforward exercise. The entailment relation must be extended by a rule per each of the protocol messages in order to compute their security levels. Hence, our constraints would be upgraded much the way multisets are rewritten in the work by Cervesato et al. [64] (though they only focus on a single attacker and their properties are classical yes/no properties). Then, once suitable size limits are stated, the imputable SCSPs could be exhaustively generated and checked against our definitions of confidentiality and authentication. Alternatively, the protocol verifier might use our framework for a finer analysis of network configurations generated using other techniques.

11.2.7 Soft Constraint Databases

The benefits given by the introduction of fuzziness and costs to constraint could also be used in the Constraint Databases framework [129,130]. In this framework the queries and the database objects are represented as constraints; adding levels of preference (or cost, or probability) could add to the framework a way to discriminate between several answers for a given query.

In particular, the probability semiring could be used to represent data that have an intrinsic uncertainty to be (or not to be) in the real problem. A possible application of this idea is represented by the geographic data bases: in this framework each object could have a different probability to be in a specific position.

11.2.8 Soft Web Query

The declarative fashion of constraints together with the levels of preference given by the semiring structure could be used to represent (and execute) query on web documents. In fact, the levels of the semiring could be used to represent the levels of satisfiability of the query, and also the structural properties of the document.

References

1. M. Abadi. Secrecy by typing in security protocols. In *Proc. 3rd International Symposium on Theoretical Aspects of Computer Software (TACS97)*, volume 1281 of *Lecture Notes in Computer Science (LNCS)*, pages 611–638. Springer, 1997.
2. M. Abadi. Secrecy by typing in security protocols. *Journal of the ACM (JACM)*, 46(5):749–786, 1999.
3. M. Abadi and A.D. Gordon. A calculus for cryptographic protocols: The spi calculus. In *Proc. 4th ACM Conference on Computer and Communications Security (CCC'97)*, pages 36–47. ACM, 1997.
4. M. Abadi and R.M. Needham. Prudent engineering practice for cryptographic protocols. *IEEE Transactions on Software Engineering (T-SE)*, 22(1):6–15, 1996.
5. A. Aggoun and N. Beldiceanu. Overview of the chip compiler system. In *Constraint Logic Programming: Selected Research.* MIT Press, 1993.
6. M.F.C. Allais. *Traité d'économie pure.* Clément-Juglar, 1971. orig. 1943: À La Recherche d'une Discipline Économique.
7. R.M. Amadio and W. Charatonik. On name generation and set-based analysis in dolev-yao model. Research Report 4379, INRIA, 2002.
8. R. Anderson. Why cryptosystems fail. In *Proc. 1st ACM Conference on Computer and Communications Security (CCC'93)*, pages 217–227. ACM, 1993.
9. K.R. Apt. The essence of constraint propagation. *CWI Quarterly*, 11(2–3):215–248, 1998.
10. K.R. Apt. The essence of Constraint Propagation. *Theoretical Computer Science*, 221(1–2):179–210, 1999.
11. K.R. Apt. The Rough Guide to Constraint Propagation (corrected version). In *Proc. 5th International Conference on Principles and Practice of Constraint Programming (CP99)*, volume 1713 of *Lecture Notes in Computer Science (LNCS)*, pages 1–23. Springer, 1999. invited lecture.
12. K.R. Apt. The role of commutativity in constraint propagation algorithms. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 22(6):1002–1036, 2000.
13. D. Awduche, J. Malcolm, J. Agogbua, M. O'Dell, and J. McManus. Rfc2702: Requirements for traffic engineering over mpls. Technical report, Network Working Group, 1999.
14. G. Baues, P. Kay, and P. Charlier. Constraint based resource allocation for airline crew management. In *Proc. Aviation, Transport and Travel Informations Systems Conference/Exhibition (ATTIS'94)*, 1994.
15. C. Beierle, E. Börger, I. Durdanivich, U. Glässer, and E. Riccobene. Refining abstract state machine specifications of the steam boiler control to well documented executable code. In *Formal Methods for Industrial Applications: The Steam-Boiler Case Study*, volume 1165 of *Lecture Notes in Computer Science (LNCS)*, pages 52–78. Springer, 1995.

16. G. Bella. *Inductive Verification of Cryptographic Protocols.* PhD thesis, Research Report 493, Computer Laboratory, University of Cambridge, 2000.
17. G. Bella and S. Bistarelli. SCSPs for modelling attacks to security protocols. In *Proc. CP2000 Post-Conference Workshop on SOFT CONSTRAINTS: THEORY AND PRACTICE*, 2000.
18. G. Bella and S. Bistarelli. Soft constraints for security protocol analysis: Confidentiality. In *Proc. 3nd International Symposium on Practical Aspects of Declarative Languages (PADL'01)*, volume 1990 of *Lecture Notes in Computer Science (LNCS)*, pages 108–122. Springer, 2001.
19. G. Bella and S. Bistarelli. Confidentiality levels and deliberate/indeliberate protocol attacks. In *Proc. 10th International Cambridge Security Protocol Workshop*, Lecture Notes in Computer Science (LNCS). Springer, 2002. To appear.
20. G. Bella and S. Bistarelli. Confidentiality levels and deliberate/indeliberate protocol attacks. In *Proc. 10th Cambridge International Security Protocol Workshop (CISPW2002)*, Lecture Notes in Computer Science (LNCS). Springer, 2002. Forthcoming.
21. G. Bella, F. Massacci, and L.C. Paulson. Verifying the SET Registration Protocols. *IEEE Journal of Selected Areas in Communications*, 1(21):77–87, 2003.
22. G. Bella and L.C. Paulson. Kerberos version iv: Inductive analysis of the secrecy goals. In J.-J Quisquater, Y. Desware, C. Meadows, and D. Gollmann, editors, *Proc. European Symposium on Research in Computer Security (ESORICS98)*, volume 1485 of *Lecture Notes in Computer Science (LNCS)*, pages 361–375. SV, 1998.
23. G. Bella and E. Riccobene. Formal analysis of the kerberos authentication system. *Journal of Universal Computer Science (J.UCS)*, 3(12):1337–1381, 1997.
24. R.E. Bellman. *Dynamic Programming.* Princeton University Press, 1957.
25. R.E. Bellman and S.E. Dreyfus. *Applied Dynamic Programming.* Princeton University Press, 1962.
26. J. Bellone, A. Chamard, and C. Pradelles. Plane - an evolutive planning system for aircraft production. In *Proc. 1st Interantional Conference on Practical Applications of Prolog (PAP92)*, 1992.
27. B.W. Benson and E. Freuder. Interchangeability preprocessing can improve forward checking search. In *Proc. 10th European Conference on Artificial Intelligence (ECAI-92)*, pages 28–30. Pitman Publishing, 1992.
28. U. Bertelè and F. Brioschi. *Nonserial Dynamic Programming.* Academic Press, 1972.
29. C. Bessière. Arc-consistency and arc-consistency again. *Artificial Intelligence*, 65(1):179–190, 1994.
30. G. Birkhoff and S. Mac Lane. *A Survey of Modern Algebra.* The Macmillan Company, 1965.
31. S. Bistarelli. Programmazione con vincoli pesati e ottimizzazione. Master's thesis, Dipartimento di Informatica, Università di Pisa, Italy, 1994. In Italian.
32. S. Bistarelli. *Soft Constraint Solving and programming: a general framework.* PhD thesis, Dipartimento di Informatica, Università di Pisa, Italy, 2001.
33. S. Bistarelli, P. Codognet, Y. Georget, and F. Rossi. Abstracting soft constraints. In *Proc. Cyprus ERCIM/COMPULOG workshop*, volume 1865 of *Lecture Notes in Artificial Intelligence (LNAI)*, pages 108–133. Springer, 2000.
34. S. Bistarelli, P. Codognet, Y. Georget, and F. Rossi. Labeling and partial local consistency for soft constraint programming. In *Proc. 2nd International Workshop on Practical Aspects of Declarative Languages (PADL'00)*, volume 1753 of *Lecture Notes in Computer Science (LNCS)*, pages 230–248. Springer, 2000.
35. S. Bistarelli, P. Codognet, and F. Rossi. An abstraction framework for soft constraint. In *Proc. Symposium on Abstraction, Reformulation and Approximation*

(SARA2000), volume 1864 of *Lecture Notes in Artificial Intelligence (LNAI)*, pages 71–86. Springer, 2000.

36. S. Bistarelli, P. Codognet, and F. Rossi. Abstracting soft constraints: Framework, properties, examples. *Artificial Intelligence*, 139(2):175–211, 2002.
37. S. Bistarelli, B. Faltings, and N. Neagu. Interchangeability in soft csps. In *Proc. 8th International Conference on Principles and Practice of Constraint Programming (CP2002)*, volume 2470 of *Lecture Notes in Computer Science (LNCS)*. Springer, 2002.
38. S. Bistarelli, H. Fargier, U. Montanari, F. Rossi, T. Schiex, and G. Verfaillie. Semiring-based CSPs and valued CSPs: Basic properties and comparison. In *Over-Constrained Systems*, volume 1106 of *Lecture Notes in Computer Science (LNCS)*, pages 111–150. Springer, 1996.
39. S. Bistarelli, H. Fargier, U. Montanari, F. Rossi, T. Schiex, and G. Verfaillie. Semiring-based CSPs and Valued CSPs: Frameworks, properties, and comparison. *CONSTRAINTS*, 4(3):199–240, 1999.
40. S. Bistarelli and S. Foley. Analysis of integrity policies using soft constraints. In *Proc. IEEE 4th international Workshop on Policies for Distributed Systems and Networks (POLICY2003)*, Lake Como, Italy, 2003. IEEE.
41. S. Bistarelli and S. Foley. A constraint framework for the qualitative analysis of dependability goals: Integrity. In *Proc. 22th international Conference on Computer safety, Reliability and Security (SAFECOMP2003)*, Lecture Notes in Computer Science (LNCS). Springer, 2003. Forthcoming.
42. S. Bistarelli, T. Fruewirth, M. Marte, and F. Rossi. Soft constraint propagation and solving in chr. In *Proc. ACM Symposium on Applied Computing (SAC)*, pages 1–5. ACM, 2002.
43. S. Bistarelli, R. Gennari, and F. Rossi. Constraint propagation for soft constraint satisfaction problems: Generalization and termination conditions. In *Proc. 6th International Conference on Principles and Practice of Constraint Programming (CP2000)*, volume 1894 of *Lecture Notes in Computer Science (LNCS)*, pages 83–97. Springer, 2000.
44. S. Bistarelli, R. Gennari, and F. Rossi. General properties and termination conditions for soft constraint propagation. *Constraints*, 8(1):79–97, 2003.
45. S. Bistarelli, U. Montanari, and F. Rossi. Constraint Solving over Semirings. In *Proc. 14th International Joint Conference on Artificial Intelligence (IJCAI95)*, pages 624–630, San Francisco, CA, USA, 1995. Morgan Kaufman.
46. S. Bistarelli, U. Montanari, and F. Rossi. Semiring-based Constraint Logic Programming. In *Proc. 15th International Joint Conference on Artificial Intelligence (IJCAI97)*, pages 352–357, San Francisco, CA, USA, 1997. Morgan Kaufman.
47. S. Bistarelli, U. Montanari, and F. Rossi. Semiring-based Constraint Solving and Optimization. *Journal of the ACM (JACM)*, 44(2):201–236, 1997.
48. S. Bistarelli, U. Montanari, and F. Rossi. SCLP semantics for (multi-criteria) shortest path problems. In *Proc. Workshop on Integration of AI and OR technique in Constraint Programming for Combinatorial Optimization Problems (CP-AI-OR'99)*, Ferrara,Italy, 1999.
49. S. Bistarelli, U. Montanari, and F. Rossi. Higher order semiring-based constraints. In *Proc. CP2000 Post-Conference Workshop on SOFT CONSTRAINTS: THEORY AND PRACTICE*, 2000.
50. S. Bistarelli, U. Montanari, and F. Rossi. Semiring-based Constraint Logic Programming: Syntax and Semantics. *ACM Transactions on Programming Languages and System (TOPLAS)*, 23(1):1–29, 2001.
51. S. Bistarelli, U. Montanari, and F. Rossi. Soft constraint logic programming and generalized shortest path problems. *Journal of Heuristics*, 8(1):25–41, 2001.

52. S. Bistarelli, U. Montanari, and F. Rossi. Soft concurrent constraint programming. In *Proc. 11th European Symposium on Programming (ESOP)*, Lecture Notes in Computer Science (LNCS), pages 53–67. Springer, 2002.
53. S. Bistarelli and E. Riccobene. An operational model for the SCLP language. In *Proc. ILPS97 workshop on Tools and Environments for (Constraint) Logic Programming*, 1997.
54. S. Bistarelli and F. Rossi. About Arc-Cconsistency in Semiring-based Constraint Problems. In *Proc. 5th International Symposium on Artificial Intelligence and Mathematics*, 1998.
55. Stefano Bistarelli, Boi Faltings, and Nicoleta Neagu. Interchangeability in soft csps. In *Recent Advances in Constraints, Proc. Joint ERCIM/CologNet International Workshop on Constraint Solving and Constraint Logic Programming, Cork, Ireland, June 19-21, 2002. Selected Papers*, volume 2627 of *Lecture Notes in Computer Science (LNCS)*. Springer, 2003.
56. C. Bodei, P. Degano, F. Nielson, and H.R. Nielson. Security analysis using flow logics. In *Current Trends in Theoretical Computer Science*, pages 525–542. World Scientific, 2001.
57. C. Bodei, P. Degano, F. Nielson, and H.R. Nielson. Static analysis for secrecy and non-interference in networks of processes. In *Proc. 6th International Conference on Parallel Computing Technologies (PaCT 2001)*, volume 2127 of *Lecture Notes in Computer Science (LNCS)*, pages 27–41. Springer, 2001.
58. F.S. De Boer and C. Palamidessi. A fully abstract model for concurrent constraint programming. In *Proc. Colloquium on Trees in Algebra and Programming (CAAP91)*, volume 493 of *Lecture Notes in Computer Science (LNCS)*, pages 296–319. Springer, 1991.
59. E. Börger. *Specification and Validation Methods.* Oxford University Press, 1994.
60. A. Borning. The programming language aspects of thinglab, a constraint oriented simulation laboratory. *ACM Transaction on Programming Languages and Systems (TOPLAS)*, 3(4):353–387, 1981.
61. A. Borning, B. Freeman-Benson, and M. Wilson. Constraint hierachies. *Lisp and Symbolic Computation*, 5(3):223–270, 1992.
62. A. Borning, M. Maher, A. Martindale, and M. Wilson. Constraint hierarchies and logic programming. In G. Levi and M. Martelli, editors, *Proc. 6th International Conference on Logic Programming (ICLP89)*, pages 149–164. MIT Press, 1989.
63. M. Calisti and B. Faltings. Distributed constrained agents for allocating service demands in multi-provider networks,. *Journal of the Italian Operational Research Society*, XXIX(91), 2000. Special Issue on Constraint-Based Problem Solving.
64. I. Cervesato, N.A. Durgin, P. Lincoln, J.C. Mitchell, and A. Scedrov. A meta-notation for protocol analysis. In *Proc. 12th IEEE Computer Security Foundations Workshop (CSFW99)*, pages 55–69. IEEE Computer Society, 1999.
65. P. Chan, K. Heus, and G. Veil. Nurse scheduling with global constraints in CHIP: Gymnaste. In *Proc. 4th International Conference and Exhibition on The Practical Application of Constraint Technology (PACT98)*, London, UK, 1998.
66. S. Chen and K. Nahrstedt. Distributed qos routing with imprecise state information. In *Proc. 7th IEEE International Conference on Computer, Communications and Networks (ICCCN'98)*, pages 614–621, 1998.
67. Berthe Y. Choueiry. *Abstraction Methods for Resource Allocation.* PhD thesis, EPFL, 1994. n. 1292.
68. Berthe Y. Choueiry and Guevara Noubir. On the computation of local interchangeability in discrete constraint satisfaction problems. In *Proc. 17th National Conference on Artificial Intelligence (AAAI-98)*, pages 326–333, 1998.
69. K.P. Chow and M. Perrett. Airport counter allocation using constraint logic programming. In *Proc. 3rd International Conference on The Practical Application of Constraint Technology (PACT97)*, London, UK, 1997.

70. A.H.W. Chun, S.H.C. Chan, F.M.F. Tsang, and D.W.M. Yeung. Stand allocation with constraint technologies at Chek Lap Kok international airport. In *Proc. 1st International Conference and Exhibition on The Practical Application of Constraint Technologies and Logic Programming (PACLP99)*, London, UK, 1999.
71. D. Clark. Rfc1102: Policy routing in internet protocols. Technical report, Network Working Group, 1989.
72. P. Codognet and D. Diaz. Compiling constraints in `clp(fd)`. *Journal of Logic Programming*, 27(3):185–226, 1996.
73. C. Collignon. Gestion optimisee de ressources humaines pour l'audiovisuel. In *Proc. CHIP users' club 96*, 1996.
74. H. Comon, V. Cortier, and J. Mitchell. Tree automata with one memory, set constraints and ping-pong protocols. In *Proc. 28th Int. Coll. Automata, Languages, and Programming (ICALP'2001)*, volume 2076 of *Lecture Notes in Computer Science (LNCS)*, pages 682–693. Springer, 2001. To appear.
75. T.H. Cormen, C.E. Leiserson, and R.L. Rivest. *Introduction to Algorithms*, chapter 26. MIT Press, 1990.
76. P. Cousot. Asynchronous iterative methods for solving a fixed point system of monotone equations in a complete lattice. Technical Report R.R.88, Institut National Polytechnique de Grenoble, 1977.
77. P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proc. 4th ACM Symposium on Principles of Programming Languages (POPL77)*, pages 238–252, 1977.
78. P. Cousot and R. Cousot. Systematic design of program analyis. In *Proc. 6th ACM Symposium on Principles of Programming Languages (POPL79)*, pages 269–282, 1979.
79. P. Cousot and R. Cousot. Abstract interpretation and application to logic programming. *Journal of Logic Programming*, 13(2–3):103–179, 1992.
80. B.A. Davey and H.A. Priestley. *Introduction to lattices and order.* Cambridge University Press, 1990.
81. F. de Boer and C. Palamidessi. From Concurrent Logic Programming to Concurrent Constraint Programming. In *Advances in Logic Programming Theory*, pages 55–113. Oxford University Press, 1994.
82. Rocco de Nicola, Gian Luigi Ferrari, and R. Pugliese. Klaim: a kernel language for agents interaction and mobility. *IEEE Transactions on Software Engineering (Special Issue on Mobility and Network Aware Computing)*, 1998.
83. R. Dechter and A. Dechter. Ief maintenance in dynamic constraint networks. In *Proc. 7th National Conference on Artificial Intelligence (AAAI-88)*, pages 37–42, 1988.
84. R. Dechter, A. Dechter, and J. Pearl. Optimization in constraint networks. In *Influence Diagrams, Belief Nets and Decision Analysis*, pages 411–425. John Wiley & Sons Ltd., 1990.
85. R. Dechter and J. Pearl. Network-Based Heuristics for Constraint-Satisfaction Problems. In *Search in Artificial Intelligence*, pages 370–425. Springer, 1988.
86. R. Dechter and J. Pearl. Tree-clustering schemes for constraint processing. In *Proc. 7th National Conference on Artificial Intelligence (AAAI-88)*, pages 150–154, 1988.
87. D.E. Denning. A lattice model of secure information flow. *Communications of the ACM (CACM)*, 19(5):236–242, 1976.
88. M. Dincbas and H. Simonis. Apache - a constraint based, automated stand allocation system. In *Proc. of Advanced Software technology in Air Transport (ASTAIR'91)*, London, UK, 1991.

89. D. Dolev and A. Yao. On the security of public-key protocols. *IEEE Transactions on Information Theory*, 2(29):198–208, 1983.
90. S.E. Dreyfus. An appraisal of some shortest-paths algorithms. *Operation Research*, 17:395–412, 1969.
91. D. Dubois, H. Fargier, and H. Prade. The calculus of fuzzy restrictions as a basis for flexible constraint satisfaction. In *Proc. IEEE International Conference on Fuzzy Systems*, pages 1131–1136. IEEE Computer Society, 1993.
92. A. Dubos and A. Du Jeu. Application EPPER planification des ageents roulants. In *Proc. CHIP users' club 96*, 1996.
93. N. Durgin, P. Lincoln, J. Mitchell, and A. Scedrov. Undecidability of bounded security protocols. In *Proc. Workshop on Formal Methods and Security Protocols (FMSP'99)*, 1999.
94. T. Ellman. Synthesis of abstraction hierarchies for constraint satisfaction by clustering approximately equivalent objects. In *Proc. of International Conference on Machine Learning*, pages 104–111, 1993.
95. M.H. Van Emden. Quantitative deduction and its fixpoint theory. *Journal of Logic Programming*, 3(1):37–53, 1986.
96. H. Fargier and J. Lang. Uncertainty in constraint satisfaction problems: a probabilistic approach. In *Proc. European Conference on Symbolic and Qualitative Approaches to Reasoning and Uncertainty (ECSQARU)*, volume 747 of *Lecture Notes in Computer Science (LNCS)*, pages 97–104. Springer, 1993.
97. R.E. Fikes. REF-ARF: A system for solving problems stated as procedures. *Artificial Intelligence*, 1:27–120, 1970.
98. M. Fitting. Bilattices and the semantics of logic programming. *Journal of Logic Programming*, 11(1–2):91–116, 1981.
99. R.W. Floyd. Algorithm 97: Shortest path. *Communication of ACM (CACM)*, 5:345, 1962.
100. F. Focacci, E. Lamma, P. Mello, and M. Milano. Constraint logic programming for the crew rostering problem. In *Proc. 3rd International Conference on The Practical Application of Constraint Technology(PACT97)*, London, UK, 1997.
101. R. Focardi, R. Gorrieri, and F. Martinelli. Information flow analysis in a discrete-time process algebra. In *Proc. 13th IEEE Computer Security Foundations Workshop (CSFW'00)*, pages 170–184. IEEE Computer Society, 2000.
102. R. Focardi, R. Gorrieri, and F. Martinelli. Non interference for the analysis of cryptographic protocols. In *Proc. 27th International Colloquium on Automata, Languages and Programming (ICALP00)*, volume 1853 of *Lecture Notes in Computer Science (LNCS)*, pages 354–372. Springer, 2000.
103. S. Foley. Personal conversations, nov 2002.
104. E.C. Freuder. Synthesizing constraint expressions. *Communication of the ACM (CACM)*, 21(11):958–966, 1978.
105. E.C. Freuder. Backtrack-free and backtrack-bounded search. In *Search in Artificial Intelligence*, pages 343–369. Springer, 1988.
106. E.C. Freuder. Eliminating interchangeable values in constraint satisfaction subproblems. In P. Marcotte and S. Nguyen, editors, *Proc. 10th National Conference on Artificial Intelligence (AAAI-91)*, pages 227–233, 1991.
107. E.C. Freuder and D. Sabin. Interchangeability supports abstraction and reformulation for constraint satisfaction. In *Proc. Symposium on Abstraction, Reformulation and Approximation (SARA'95)*, 1995.
108. E.C. Freuder and R.J. Wallace. Partial constraint satisfaction. *Artificial Intelligence*, 58:21–70, 1992.
109. M. Fromherz, V. Gupta, and V. Saraswat. Model-based computing: constructing constraint-based software for electro-mechanical systems. In *Proc. 1st International Conference on The Practical Application of Constraint Technology(PACT97)*, 1995.

110. T. Frühwirth and P. Brisset. Optimal planning of digital cordless telecommunication systems. In *Proc. 3rd International Conference on The Practical Application of Constraint Technology (PACT97)*, London, UH, 1997.
111. Y. Georget. *Extensions de la Programmation par Contraintes.* PhD thesis, Ecole Polytecnique, Paris, 1999.
112. Y. Georget and P. Codognet. Compiling semiring-based constraints with clp(fd,s). In *Proc. 4th International Conference on Principles and Practice of Constraint Programming (CP98)*, volume 1520 of *Lecture Notes in Computer Science (LNCS)*, pages 205–219. Springer, 1998.
113. C. Gervet, Y. Caseau, and D. Montaut. On refining ill-defined constraint problems: A case study in iterative prototyping. In *Proc. 1st International Conference and Exhibition on The Practical Application of Constraint Technologies and Logic Programming (PACLP99)*, London, UK, 1999.
114. R. Giacobazzi, S.K. Debray, and G. Levi. A Generalized Semantics for Constraint Logic Programs. In *Proc. FGCS92*, pages 581–591. ICOT, 1992.
115. F. Glaisner and L.-M. Richard. FORWARD-C: A refinery scheduling system. In *Proc. 3rd International Conference on The Practical Application of Constraint Technology(PACT97)*, London, UK, 1997.
116. S. Gnesi, A. Martelli, and U. Montanari. Dynamic programming as graph searching: an algebraic approach. *Journal of the ACM (JACM)*, 28(5):737–751, 1981.
117. E. Gray. American national standard T1.523-2001, telecom glossary 2000. published on the Web at http://www.its.bldrdoc.gov/projects/telecomglossary2000, 2001.
118. Y. Gurevich. Evolving algebras 1993: Lipari guide. In *Specification and Validation Methods*, pages 9–36. Oxford University Press, 1995.
119. M. Gyssens, P.G. Jeavons, and D.A. Cohen. Decomposing Constraint Satisfaction Problems Using Database Techniques. *Artificial Intelligence*, 66(1):57–89, 1994.
120. A. Haselbock. Exploiting interchangeabilities in constraint satisfaction problems. In *Proc. 13th International Joint Conference on Artificial Intelligence (IJCAI91)*, pages 282–287. Morgan Kaufman, 1993.
121. P. Van Hentenryck. *Constraint Satisfaction in Logic Programming.* MIT Press, 1989.
122. P. Van Hentenryck, Y. Deville, and C.M. Teng. A generic arc-consistency algorithm and its specializations. *Artificial Intelligence*, 57:291–321, 1992.
123. International Organization for Standardization. *Information processing systems – Open Systems Interconnection – Basic Reference Model – Part 2: Security Architecture (ISO 7498-2)*, 1989.
124. N. Itoi and P. Honeyman. Smartcard Integration with Kerberos V5. In *Proceedings of the USENIX Workshop on Smartcard Technology*, pages 59–62, 1999.
125. J. Jaffar and J.L. Lassez. Constraint logic programming. Technical report, IBM T.J. Watson Research Center, Yorktown Heights, 1986.
126. J. Jaffar and J.L. Lassez. Constraint logic programming. In *Proc. 14th ACM Symposium on Principles of Programming Languages (POPL87)*, pages 111–119. ACM, 1987.
127. J. Jaffar and M.J. Maher. Constraint logic programming: A survey. *Journal of Logic Programming*, 19–20:503–581, 1994.
128. R. Jain and W. Sun. QoS/Policy/Constraint-Based routing. In *Carrier IP Telephony 2000 Comprehensive Report.* International Engineering Consortium, 2000.
129. P.C. Kanellakis and D.Q. Goldin. Constraint programming and database query languages. In *Proc. 2nd International Symposium on Theoretical Aspects of Computer Software (TACS94)*, volume 789 of *Lecture Notes in Computer Science (LNCS)*, pages 66–77. Springer, 1994.
130. P.C. Kanellakis, G.M. Kuper, and P.Z. Revesz. Constraint query languages. *Journal of Computer and System Science*, 51(1):26–52, 1995.

131. F. Kokkoras and S. Gregory. D-WMS: Distributed workforce management using CLP. In *Proc. 4th International Conference and Exhibition on The Practical Application of Constraint Technology (PACT98)*, London, UK, 1998.
132. V. Kumar. Algorithms for constraint satisfaction problems: a survey. *AI Magazine*, 13(1):32–44, 1992.
133. J-L. Lauriere. A language and a program for stating and solving combinatorial problems. *Artifical Intelligence*, 10:29–127, 1978.
134. W. Leler. *Constraint Programming Languages, their specification and generation.* Addison-Wesley, 1988.
135. J.W. Lloyd. *Foundations of Logic Programming.* Springer, 1987.
136. G. Lowe. An attack on the needham-schroeder public-key authentication protocol. *Information Processing Letters*, 56(3):131–133, 1995.
137. G. Lowe. Some new attacks upon security protocols. In *Proc. 9th IEEE Computer Security Foundations Workshop (CSFW96)*, pages 139–146. IEEE Computer Society, 1996.
138. A.K. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8(1):99–118, 1977.
139. A.K. Mackworth. Constraint satisfaction. In *Encyclopedia of AI (second edition)*, pages 285–293. John Wiley & Sons Ltd., 1992.
140. A.K. Mackworth and E.C. Freuder. The complexity of some polynomial network consistency algorithms for constraint satisfaction problems. *Artificial Intelligence*, 25:65–74, 1985.
141. K. Marriott and P. Stuckey. *Programming with Constraints.* MIT Press, 1998.
142. A. Martelli and U. Montanari. Nonserial dynamic programming: on the optimal strategy of variable elimination for the rectangular lattice. *Journal of Mathematical Analysis and Application*, 40:226–242, 1972.
143. W. Maximuck and M. Reviakin. A prototype of train schedule module. In *Proc. 4th International Conference and Exhibition on The Practical Application of Constraint Technology (PACT98)*, London, UK, 1998.
144. C. Meadows. Private conversations, 2001.
145. J. Millen and V. Shmatikov. Constraint solving for bounded-process cryptographic protocol analysis. In *Proc. 8th ACM Conference on Computer and Communication Security*, pages 166–175. ACM, 2001.
146. S.P. Miller, J.I. Neuman, J.I. Schiller, and J.H. Saltzer. Kerberos Authentication and Authorisation System. Technical Plan, MIT - Project Athena, 1989.
147. J. C. Mitchell, M. Mitchell, and U. Stern. Automated Analysis of Cryptographic Protocols Using Murphi. In *Proc. IEEE Symposium on Security and Privacy.* IEEE Press, 1997.
148. B. Mobasher, D. Pigozzi, and G. Slutzki. Multi-valued logic programming semantics: An algebraic approach. *Theoretical Computer Science*, 171(1–2):77–109, 1997.
149. U. Montanari. Fundamental properties and applications to picture processing, information sciences. Technical Report 71-1, Dept. of Computer Science, Carnegie Mellon University, 1971.
150. U. Montanari. Networks of constraints: Fundamental properties and applications to picture processing. *Information Science*, 7:95–132, 1974.
151. U. Montanari and F. Rossi. An efficient algorithm for the solution of hierachical networks of constraints. In *Proc. 3rd International Workshop on Graph Grammars and their Application to Computer Science*, volume 291 of *Lecture Notes in Computer Science (LNCS)*, pages 440–457. Springer, 1986.
152. U. Montanari and F. Rossi. Constraint relaxation may be perfect. *Artificial Intelligence*, 48:143–170, 1991.

153. P. Morris. On the density of solutions in equilibrium points for the queens problem. In *Proc. 11th National Conference on Artificial Intelligence (AAAI-92)*, pages 428–433, 1992.
154. H. Moulin. *Axioms for Cooperative Decision Making.* Cambridge University Press, 1988.
155. N. Neagu and B. Faltings. Exploiting interchangeabilities for case adaptation. In *International Conference on Case-Based Reasoning (ICCBR'02)*, Lecture Notes in Computer Science (LNCS), pages 422–436. Springer, 2001.
156. B. C. Neuman and T. Ts'o. Kerberos: An authentication service for computer networks. In *William Stallings, Practical Cryptography for Data Internetworks.* IEEE Press, 1996.
157. L. Pacholski and A. Podelski. Set constraints: A pearl in research on constraints (invited tutorial). In *Proc. Principles and Practice of Constraint Programming (CP97)*, volume 1330 of *Lecture Notes in Computer Science (LNCS)*, pages 549–562. Springer, 1997.
158. S. Pallottino and M.G. Scutellà. *Equilibrium and Advanced Transportation Modelling*, chapter 11. Shortest Path Algorithms in Transportation Models: Classical and Innovative Aspects, pages 245–281. Kluwer, 1998.
159. V. Pareto. *Manual of Political Economy.* Augustus M. Kelley, 1971. orig. (1906) in Italian.
160. L.C. Paulson. The inductive approach to verifying cryptographic protocols. *Journal of Computer Security*, 6:85–128, 1998.
161. M. Perrett. Using constraint logic programming techniques in container port planning. *ICL Technical Journal*, pages 537–545, 1991.
162. G.D. Plotkin. Post-graduate lecture notes in advanced domain theory (incorporating the pisa lecture notes). Technical report, Dept. of Computer Science, Univ. of Edinburgh, 1981.
163. J-F. Puget. A C++ implementation of CLP. In *Proc. Singapore International Conference on Intelligent Systems (SPICIS'94)*, 1994.
164. L. Purvis and P. Jeavons. Constraint tractability theory and its application to the product development process for a constraint-based scheduler. In *Proc. 1st International Conference and Exhibition on The Practical Application of Constraint Technologies and Logic Programming (PACLP99)*, London, UK, 1999.
165. A. Rosenfeld, R.A. Hummel, and S.W. Zucker. Scene labelling by relaxation operations. *IEEE Transactions on Systems, Man, and Cybernetics (T-SMC)*, 6(6):420–433, 1976.
166. F. Rossi. Redundant hidden variables in finite domain constraint problems. In *Constraint Processing*, volume 923 of *Lecture Notes in Computer Science (LNCS)*, pages 205–223. Springer, 1995.
167. Z. Ruttkay. Fuzzy constraint satisfaction. In *Proc. 3rd IEEE International Conference on Fuzzy Systems*, pages 1263–1268, 1994.
168. Z. Ruttkay. Constraint satisfaction – a survey. *CWI Quarterly*, 11(2–3):163–214, 1998.
169. G. Verfaillie S. de Givry and T. Schiex. Bounding the optimum of constraint optimization problems. In G. Smolka, editor, *Proc. 3rd International Conference on Principles and Practice of Constraint Programming (CP97)*, volume 1330 of *Lecture Notes in Computer Science (LNCS)*, pages 405–419. Springer, 1997.
170. D. Sabin and E.C. Freuder. Understanding and improving the mac algorithm. In *Proc. 3rd International Conference on Principles and Practice of Constraint Programming (CP97)*, volume 1330 of *Lecture Notes in Computer Science (LNCS)*, pages 167–181. Springer, 1997.
171. V.A. Saraswat. *Concurrent Constraint Programming.* MIT Press, 1993.

172. V.A. Saraswat, M. Rinard, and P. Panangaden. Semantic foundations of concurrent constraint programming. In *Proc. 18th ACM Symposium on Principles of Programming Languages (POPL91)*, pages 333–352. ACM, 1991.
173. T. Schiex. Possibilistic constraint satisfaction problems, or "how to handle soft constraints?". In *Proc. 8th Conference of Uncertainty in Artificial Intelligence (UAI92)*, pages 269–275. Morgan Kaufmann, 1992.
174. T. Schiex, H. Fargier, and G. Verfaille. Valued Constraint Satisfaction Problems: Hard and Easy Problems. In *Proc. 14th International Joint Conference on Artificial Intelligence (IJCAI95)*, pages 631–637, San Francisco, CA, USA, 1995. Morgan Kaufmann.
175. R. Schrag and D. Miranker. An evaluation of domain reduction: Abstraction for unstructured CSPs. In *Proc. Symposium on Abstraction, Reformulation, and Approximation (SARA92)*, pages 126–133, 1992.
176. D. De Schreye, M. Leuschel, and B. Martens. Tutorial on program specialisation. In *Proc. International Logic Programming Symposium (ILPS95)*, pages 615–616. MIT Press, dec 1995.
177. D.S. Scott. Domains for denotational semantics. In *Proc. 9th International Colloquium on Automata, Languages and Programming (ICALP82)*, volume 140, pages 577–613. Springer, 1982.
178. P. Shenoy. Valuation-based systems for discrete optimization. In *Uncertainty in Artificial Intelligence*. Elsevier Science, 1991.
179. P. Shenoy. Valuation-based systems: A framework for managing uncertainty in expert systems. In *Fuzzy Logic for the Management of Uncertainty*, pages 83–104. John Wiley & Sons Ltd., 1994.
180. H. Simonis and P. Charlier. Cobra - a system for train crew scheduling. In *Proc. DIMACS workshop on constraint programming and large scale combinatorial optimization*, 1998.
181. H. Simonis and T. Cornelissens. Modelling producer/consumer constraints. In *Proc. 1st International Conference on Principles and Practice of Constraint Programming (CP95)*, volume 976 of *Lecture Notes in Computer Science (LNCS)*, pages 449–462. Springer, 1995.
182. M.B. Smyth. Power domains. *Journal of Computer and System Sciences*, 16(1):23–36, 1978.
183. R. Sosic and J. Gu. 3000000 queens in less than one minute. *SIGART Bulletin*, 2(3):22–24, 1991.
184. G.L. Steel. *The definition and implementation of a computer programming language based on Constraints.* PhD thesis, MIT, 1980.
185. D.R. Stinson. *Cryptography Theory and Practice.* CRC Press, 1995.
186. G.J. Sussman and G.L. Steele. CONSTRAINTS - a language for expressing almost-hierarchical descriptions. *Artificial Intelligence*, 14(1):1–39, 1980.
187. I. Sutherland. Sketchpad: a man-machine graphical communication system. In *Proc. Spring Joint Computer Conference*, volume 23 of *AFIPS*, pages 329–346, 1963.
188. A. Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5:285–309, 1955.
189. E.P.K. Tsang. *Foundations of Constraint Satisfaction.* Academic Press, 1993.
190. M. Wallace. Practical applications of constraint programming. *Constraints*, 1(1–2):139–168, 1996.
191. T. Walsh and F. Giunchiglia. Theories of abstraction: A historical perspective. In *Proc. AAAI-92 Workshop on Approximation and Abstraction of Computational Theories*, 1992.
192. T. Walsh and F. Giunchiglia. A theory of abstraction. *Artificial Intelligence*, 56(2–3):323–390, 1992.

193. D.L. Waltz. Understanding line drawings of scenes with shadows. In *The Psychology of Computer Vision*, pages 19–91. McGraw-Hill, 1975.
194. R. Weigel and B. Faltings. Compiling constraint satisfaction problems. *Artificial Intelligence*, 115:257–289, 1999.
195. Rainer Weigel, Boi Faltings, and Marc Torrens. Interchangeability for case adaptation in configuration problems. In *Workshop on Case-Based Reasoning Integrations (AAAI-98)*, volume WS-98-15, pages 166–171, Madison, Wisconsin, USA, July 1998. AAAI Press.
196. L.A. Zadeh. Fuzzy sets. *Information and Control*, 8(3):338–353, 1965.