Abdoulaye Gamatié

Designing Embedded Systems with the SIGNAL Programming Language

Synchronous, Reactive Specification

Springer

Designing Embedded Systems with the SIGNAL Programming Language

Abdoulaye Gamatié

Designing Embedded Systems with the SIGNAL Programming Language

Synchronous, Reactive Specification

Springer

Abdoulaye Gamatié
CNRS - UMR 8022 (LIFL)
INRIA Lille - Nord Europe
Parc scientifique de la Haute Borne
Park Plaza - Bâtiment A
40 avenue Halley
59650 Villeneuve d'Ascq, France
Abdoulaye.Gamatie@lifl.fr

ISBN 978-1-4419-0940-4 e-ISBN 978-1-4419-0941-1
DOI 10.1007/978-1-4419-0941-1
Springer New York Dordrecht Heidelberg London

Library of Congress Control Number: 2009930637

© Springer Science+Business Media, LLC 2010
All rights reserved. This work may not be translated or copied in whole or in part without the written permission of the publisher (Springer Science+Business Media, LLC, 233 Spring Street, New York, NY 10013, USA), except for brief excerpts in connection with reviews or scholarly analysis. Use in connection with any form of information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed is forbidden.
The use in this publication of trade names, trademarks, service marks, and similar terms, even if they are not identified as such, is not to be taken as an expression of opinion as to whether or not they are subject to proprietary rights.

Printed on acid-free paper

Springer is part of Springer Science+Business Media (www.springer.com)

To Leila, and the memory of Boubakar,
Fanta, and Tantie

Foreword

I am very pleased to play even a small part in the publication of this book on the SIGNAL language and its environment POLYCHRONY. I am sure it will be a significant milestone in the development of the SIGNAL language, of synchronous computing in general, and of the dataflow approach to computation.

In dataflow, the computation takes place in a producer–consumer network of independent processing stations. Data travels in streams and is transformed as these streams pass through the processing stations (often called filters). Dataflow is an attractive model for many reasons, not least because it corresponds to the way production, transportation, and communication are typically organized in the real world (outside cyberspace).

I myself stumbled into dataflow almost against my will. In the mid-1970s, Ed Ashcroft and I set out to design a "super" structured programming language that, we hoped, would radically simplify proving assertions about programs. In the end, we decided that it had to be declarative. However, we also were determined that iterative algorithms could be expressed directly, without circumlocutions such as the use of a tail-recursive function.

The language that resulted, which we named LUCID, was much less traditional then we would have liked. LUCID statements are equations in a kind of executable temporal logic that specify the (time) sequences of variables involved in an iteration.

We had originally planned to translate LUCID programs into imperative code, but that proved very difficult. Several people suggested using a dataflow approach, in which the time sequences are realized as streams in a dataflow network. In fact, Gilles Kahn had anticipated this approach in his famous 1974 paper in which he showed that a fairly straightforward dataflow scheme correctly computed the least fixed point of the corresponding stream transformation equations.

LUCID was fascinating but unfocused – one colleague called it "a solution looking for a problem." What we needed was a "killer" application. We looked at scientific computing, graphics, even text processing. It never occurred to us to consider real-time control.

It did, however, occur to the founders of the French school of "synchronous" programming. Encouraged by Gérard Berry and Albert Benveniste, they ruthlessly and drastically simplified the language. This was unavoidable, because the original, general, LUCID seemed to require unbounded FIFO queues between producers

and consumers, and even dynamically growing dataflow nets. The stripped-down languages could now be compiled into (fast) imperative code, but still allowed the programmer to think in terms of dataflow. And, most important of all, one could still carry out formal, practical, and reliable reasoning about the properties of programs.

Of the resulting languages, the Rennes group's SIGNAL – the subject of this book – is arguably the most faithful to the dataflow principle.

Synchronous systems have turned out to be a killer application area for dataflow; however, a killer application is, from what I have seen, not in itself enough to ensure wide adoption of a technology. You also need a good implementation and, along with it, good documentation. Furthermore, it is easy to underestimate how much documentation is needed: conference papers, journal papers, manuals – at a minimum. To really make an impact you need books – thorough, comprehensive, detailed, exhaustive, like the present volume. Abdoulaye Gamatié's book will, I expect, represent a great step forward for synchronous programming and, I hope, for dataflow in general.

University of Victoria *Bill Wadge*

Preface

This book has been written to present the design of embedded systems in safety-critical domains such as automotive vehicles, avionics, and nuclear power plants, by using the SIGNAL programming language. SIGNAL is part of the *synchronous language* family, which advocates design techniques that strongly promote the use of formal concepts, i.e., those having a mathematically sound basis. This book is the first attempt to provide a wide public (scientists, practitioners, and students) with a pedagogical presentation of the necessary rudiments for a successful and pragmatic usage of SIGNAL.

Before detailing the motivations and organization of the book, some historical notes[1] are briefly mentioned about the synchronous languages, in general, and about the SIGNAL language, in particular.

The Advent of the Synchronous Languages

The birth of synchronous languages and models dates back to the beginning of the 1980s, when a few researchers, from control theory and computer science fields, noticed the need for a more adequate design philosophy for highly critical embedded and real-time systems. An important feature of these systems is that they often maintain a continuous interaction with their environment. This environment can be either some physical devices to be controlled, or a human operator, e.g., an aircraft pilot who must realize some specific task. It is the responsibility of the environment to decide the rhythm at which the interaction takes place with a system. Such

[1] These historical notes rely on the following papers:

- "The synchronous languages 12 years later" by Benveniste, Caspi, Edwards, Halbwachs, Le Guernic, and de Simone, published in the *Proceedings of the IEEE*, 91(1):64–83 in 2003
- "Polychronous design of real-time applications with SIGNAL" by Gautier, Le Guernic, and Talpin, published in *ARTIST Survey of Programming Languages* in 2008
- "A synchronous language at work: the story of Lustre" by Halbwachs, presented at the Memocode'05 conference in 2005

systems are usually referred to as *reactive systems*. The synchronous programming paradigm offers a suitable notion of deterministic concurrency in which time is abstracted by symbolic synchronization relations, facilitating the analysis of reactive embedded and real-time system behaviors.

Among the pioneering works concerning this idea are the following: the earlier Grafcet formalism of the Association Française pour la Cybernétique Économique et Technique (AFCET) agency (in France), the synchronous calculus of communicating systems of Milner (in Edinburgh, UK), the STATECHARTS formalism of Harel and Pnueli (in Israel), the ESTEREL language of Marmorat, Rigault, and Berry (at École des Mines, Sophia-Antipolis, in France), the LUSTRE language of Caspi and Halbwachs (at Verimag, Grenoble, in France), and the SIGNAL language of Benveniste and Le Guernic [at Institut National de Recherche en Informatique et Automatique (INRIA) Rennes in France]. Then, the latter three languages became the main pillars of synchronous programming, before the proposition of further languages from the beginning of the 1990s to now, such as the ARGOS language of Maraninchi (at Verimag, Grenoble, in France) and the SYNCCHARTS formalism of André (at Université de Nice, Sophia-Antipolis, in France) as "fully synchronous" versions of STATECHARTS, the REACTIVE-C language of Boussinot (at INRIA Sophia-Antipolis in France), and the LUCID SYNCHRONE language of Caspi and Pouzet (at Verimag, Grenoble, and Laboratoire de Recherche en Informatique, Paris, in France). All these languages provide designers with various description styles under the form of dataflow declarative languages, e.g., LUSTRE, LUCID SYNCHRONE, and SIGNAL, and imperative languages, e.g., ESTEREL and SYNCCHARTS.

The synchronous languages and their associated technology have reached a sufficient maturity that makes them an excellent candidate for the design of real-world safety-critical systems such as flight control systems in avionics and control systems in nuclear power plants. This explains the high interest for these languages in industry today, and particularly their successful adoption by European industries.

Focus on the SIGNAL Language

In 1981, the French INRIA institute and Centre National d'Études des Télécommunications (CNET) started a joint project on the design of signal processing applications executing on digital signal processors. From the INRIA side, research teams from INRIA Rennes and INRIA Rocquencourt were involved in this project. Among the objectives of the project was the definition of a new domain-specific language, adopting a dataflow and graphical style with array and sliding window operators, required for the design of signal processing applications. This language is called "SIGNAL" in reference to its original target application domain, i.e., signal processing. In 1988, it became a joint trademark of CNET and INRIA.

During the joint CNET–INRIA project, Le Guernic and Benveniste were in charge of SIGNAL language definition, together with Gautier. The first paper on SIGNAL, dealing with an algebraic description of networks of flows, was authored

by Le Guernic in 1982. The first complete description of the SIGNAL language was provided by Gautier in his Ph.D. thesis, in 1984. Then, Le Guernic and Benveniste proposed the algebraic encoding of abstract clocks in the $\mathbb{Z}/3\mathbb{Z}$ domain in 1986. Together with other contributors, they described the semantics of SIGNAL using different models: operational semantics (presented in this book), denotational semantics, trace semantics (which is used in the reference manual for SIGNAL version 4), and the more recent tagged model (also presented in this book), now considered as a reference paper for the polychronous model. In addition to these propositions, Nowak proposed in 1999 a coinductive semantics for modeling SIGNAL in the Coq proof assistant.

During the 1990s, a full compiler, implementing the abstract clock calculus of SIGNAL (with hierarchies of Boolean abstract clocks), was described by Besnard in his Ph.D. thesis, defended in 1992. This abstract clock calculus was improved 3 years later during the Ph.D. study of Amagbegnon, who defined arborescent canonical forms. In the same period, the SIGNAL language was improved and its application domain was extended to embedded and real-time systems in general. In particular, the native relational style of the language naturally enables its application to model systems with multiple physical clock rates.

During the same period, the design and implementation of distributed embedded systems using SIGNAL became a hot research topic at INRIA Rennes. Several Ph.D. studies have been devoted to this topic. These studies were conducted in the context of cooperative projects, including European projects such as Synchron, Syrf, Sacres, and SafeAir. Among them, those which are in phase with the mainstream of the current SIGNAL version are the optimization methods proposed by Chéron, the clustering models for SIGNAL programs defined by Le Goff, the required notions for abstraction and separate compilation formalized by Maffeïs, and the implementation of distributed programs described by Aubry. All these studies were conducted by the authors during their Ph.D. research.

In addition to the aforementioned studies, many other works have concerned extensions of SIGNAL, translations to or from SIGNAL, and specific applications. Among these works is the definition of an affine abstract clock calculus for affine abstract clocks by Smarandache. Belhadj, Kountouris, and Le Lann, in collaboration with Wolinski, used SIGNAL for hardware description and synthesis, and proposed a temporal interpretation of SIGNAL programs for the validation of quantitative real-time properties of embedded systems. Dutertre, Le Borgne, and Marchand developed the theory of polynomial dynamical systems on $\mathbb{Z}/3\mathbb{Z}$, implemented it in the SIGALI model-checking tool, and applied it for verification and controller synthesis on SIGNAL programs.

The first decade of the twenty-first century is the era of significant theoretical and practical studies on the polychronous semantic model of SIGNAL, considered as the reference model for the language nowadays. Le Guernic, Benveniste, and other contributors characterized specific classes of polychronous programs such as endochronous ones. They analyzed the links between synchrony and asynchrony and introduced the property of isochrony in the context of synchronous transition systems. A more constructive version of isochrony, called endo-isochrony,

was proposed later by Le Guernic, Talpin, and Le Lann in the tagged model of polychrony. During my Ph.D. research (defended in 2004), I used the SIGNAL language to define a polychronous modeling of real-time executive services of the Aeronautical Radio Inc. (ARINC) avionic standard. The polychronous model of SIGNAL is well suited for the description of concurrent systems via a modular composition of their constituent elements while preserving the global synchronization relations in these systems. Hence, it is well adapted for the design of multiclocked systems in which each component owns a local activation clock, e.g., distributed real-time systems.

A major part of the aforementioned works was carried out within the academic design environment of SIGNAL, called POLYCHRONY, distributed online for free at http://www.irisa.fr/espresso/Polychrony.

In addition to POLYCHRONY, there is an industrial environment for the SIGNAL programming, originally called Sildex, developed by the French software company Techniques Nouvelles pour l'Informatique (TNI) in 1993. SIGNAL was licensed to TNI in the early 1990s. Today, TNI is part of the French Geensys group (http://www.geensys.com). The Sildex commercial toolset, now called RT-Builder, is supplied by Geensys. Among the industrial users of RT-Builder are Snecma, Airbus, and Thales Airborne Systems. Snecma uses RT-Builder for the modeling of Aircraft engine control. Airbus also uses RT-Builder for similar purposes, e.g., modeling of systems of the Airbus A380. Thales Airborne Systems rather uses the tool for performance evaluation of airborne systems.

SIGNAL Versus the Other Synchronous Languages

As a main difference from the other synchronous languages, SIGNAL naturally considers a mathematical time model, in terms of partial order relations, to describe multiclocked systems without the necessity of a sequential reference (or global) abstract clock that serves for the synchronization of system components. This modeling vision, referred to as "polychrony' in the jargon of SIGNAL, also offers a comfortable expressivity to specify asynchronous system behaviors, in which the interleaving of observed events needs to be addressed at a fine-grain level. For that purpose, the polychronous model abstracts the synchronization and scheduling relations between the events that form a system behavior. The resulting relational descriptions are not always deterministic and executable, e.g., when they describe a system only partially. Thus, SIGNAL is both a specification and a programming (in the sense of executable specifications) language.

This vision adopted by SIGNAL differs from that of the other synchronous languages, e.g., LUSTRE and LUCID SYNCHRONE, which assume a priori the existence of a reference abstract clock in system descriptions. In addition, with a synchronous language such as LUSTRE, a developer always writes actual executable specifications; so he or she is programming. However, even though SIGNAL specifications are sometimes not executable, they can be used for property analysis in the system

under construction so as to obtain some early feedback about the current design decisions. But, in most cases such specifications are given executable semantics. In fact, the SIGNAL specification paradigm is very similar to *constraint programming*, where relations between variables are described in the form of constraints. The ESTEREL language recently proposed a multiclocked extension to permit the description of globally asynchronous, locally synchronous systems.

Motivation for This Book

Lots of papers have been published on different aspects of the synchronous language SIGNAL. However, today, an extended pedagogical document presenting the basic material and the main technical concepts of SIGNAL and its associated programming style is lacking. Such a document would strongly facilitate the adoption of the language by scientists, practitioners, and students for the design of embedded systems. The challenge of this book is to fill this demand.

The content of this book was originally and freely inspired by the real-time programming teaching notes (at master's level) of Bernard Houssais, formerly an associate professor at Université de Rennes 1 in France, who retired in 2004. As a former assistant professor (ATER) at Université de Rennes 1, I replaced Bernard and taught that same course for 1 year before moving to the Laboratoire d'Informatique Fondamentale de Lille (France) by the end of 2005. I still teach the same course a few hours per year to master's level students, at Université des Sciences et Technologies de Lille. The Bernard's original teaching notes are currently available at http://www.irisa.fr/espresso/Polychrony. Some the exercises provided in this book come from these notes. The material presented in the book also relies on the rich literature devoted to SIGNAL. As an important complement, the reader could also refer to the works of the Environnement de Spécification de Programmes Reactifs Synchrones (ESPRESSO) team project (http://www.irisa.fr/espresso), which develops the SIGNAL language. The reference manual for SIGNAL written by some members of ESPRESSO (L. Besnard, T. Gautier, and P. Le Guernic) is the most complete technical document on the language. In my opinion, it is a good companion to this book for those who want to learn more about SIGNAL. It is available at the same Web site as the teaching notes mentioned previously.

An important part of this book also relies on my experience with extensive usage of the SIGNAL language for the design of embedded systems in my research activities. As mentioned in the historical notes about SIGNAL, I contributed to the development of the polychronous model during my Ph.D. research, supervised by Le Guernic and Gautier (two major contributors to the SIGNAL language) within the ESPRESSO team project. Finally, even though most of the examples presented in the book concern design and programming, the notions introduced could also serve for specification.

How To Read This Book

The overall organization of the book consists of four parts and an appendix, described below.

- Part I: **Real-time and synchronous programming.** This part is devoted to the presentation of general notions about programming approaches for real-time embedded systems. *Readers who are not familiar at all with the safety-critical embedded system domain may find in this part some basic elements to understand the important challenges for and issues with system design*. Chapter 1 recalls some elementary definitions of real-time embedded systems and discusses different viewpoints on how to model timing aspects when programming such systems. In particular, it aims to show that according to the way time is modeled in a system, reasoning about nonfunctional properties becomes more or less easy and relevant. Chapter 2 concentrates on synchronous programming. It introduces the foundations of the synchronous approach and gives a panorama of synchronous languages. The overall goal of this chapter is to give the unfamiliar reader a flavor of synchronous programming in general.
- Part II: **Basic concepts and notations of** SIGNAL**.** The presentation of SIGNAL programming, which is the main topic of this book starts from Part II. *This part is very easy to read for readers who are familiar with any general-purpose programming language such as C or Java. After this part, beginners are expected to be able to define their first* SIGNAL *programs.* The very basic concepts of the SIGNAL language are introduced as follows. Chapter 3 presents the notions of SIGNALas well as relations. Then, Chap. 4 presents the programming units, called processes. Chapter 5 describes some useful extended constructs of the language, which are specifically devoted to the expression of pure control (i.e., abstract clock manipulation, which is a particularity of SIGNAL in comparison with the other synchronous languages). Finally, Chap. 6 details the practical design of a simple example: from the SIGNAL specification to the simulation via the code generation. At the end of each chapter, the reader is provided with some training exercises.
- Part III: **Formal properties of** SIGNAL **programs.** The mathematical foundations of the SIGNAL language are presented in this part. This characteristic makes the language suitable for formal reasoning about the properties of defined models. Hence, it favors the trustworthy validation of designed systems. *This part is recommended for readers who want to learn, on the one hand, the formal semantics of the* SIGNAL *language and, on the other hand, the formal properties of* SIGNAL *programs that are considered for their analysis. It gives a good picture of what a formal language enables*. Chapter 7 first describes two kinds of semantics for the SIGNAL language: an operational semantics and a denotational semantics. Then, Chap. 8 presents the encoding of SIGNAL programs in the $\mathbb{Z}/3\mathbb{Z}$ domain, allowing one to reason on programs based on the algebraic properties of this domain. Finally, Chap. 9 illustrates typical program analyses in $\mathbb{Z}/3\mathbb{Z}$, and how the result of such analyses is exploited to automatically generate

executable code. These last two chapters describe what is actually done during the compilation of SIGNAL programs in the compiler of POLYCHRONY. In all chapters in this part, a few training exercises are also given.

- Part IV: **Advanced design in SIGNAL.** This part addresses pragmatic design and programming issues in SIGNAL. *It provides some concepts and examples that can significantly help readers to define nontrivial designs.* Chapter 10 first presents the following notions: modularity for reuse, abstraction, abstract clock refinement or oversampling, and assertion for contract-based specification. Chapter 11 deals with the design of multiclocked systems, and globally asynchronous, locally synchronous systems in particular. This topic has been extensively studied in SIGNAL, mostly from a theoretical point of view. Chapter 12 gives some design patterns that help readers to understand more the design principles in SIGNAL. Finally, Chap. 13 illustrates the complete design steps for the implementation of a solution to the well-known synchronization problem of asynchronous systems: the dining philosophers. Similarly to the previous two parts, in each chapter of this part, some training exercises are provided. *From now on, the reader is supposed to be skilled enough with the* SIGNAL *programming concepts to tackle and debug complex problems!*
- Appendixes: Appendix A indicates the SIGNAL compiler's commands that are most often used in practice. Appendix B gives the grammar of the SIGNAL language. Finally, solution ideas for the exercises given throughout this book are provided at the end of the book.

All the SIGNAL descriptions presented in the book are defined with the Polychrony toolset. For their compilation, version 4.15.10 of the batch compiler (available at http://www.irisa.fr/espresso/Polychrony) was considered.

Acknowledgments

This book is obviously not the product of my sole efforts. It is built upon the numerous results obtained by all contributors to the SIGNAL language since the early 1980s. So, my primary acknowledgments will naturally go to all those who have worked on the language. I am especially grateful to P. Le Guernic, T. Gautier, and L. Besnard, who contributed the most to my understanding of SIGNAL and its programming principles. I am also grateful to B. Houssais, whose teaching notes served as the basic inspiration and material for this book. I would like to thank J.-P. Talpin and S. Shukla for their suggestion and encouragement to submit my earlier tutorial document on SIGNAL for publication. This book is a revised and improved version of this tutorial.

I want to express my gratitude to B. Wadge and S. Shukla for having kindly accepted to write, respectively, the foreword and the back-cover statements for this book. I would also like to greatly thank a number of people for the care with which they made invaluable comments so I could improve the previous draft versions of this book: C. André, K. Arnaud, L. Besnard, P. Boulet, P. Devienne, A. Etien,

T. Gautier, B. Jose, P. Le Guernic, M.-R. Mousavi, É. Piel, S. Shukla, and S. Suhaib. Many thanks go to A.-L. Leroy, I. Quadri, and G. Rouzé for their help in the illustration of the book. Also, thanks to CNRS and Xilinx for the permission to use their images for illustration. I am grateful to C. Glaser and A. Davis from Springer for their good work on this edition of the book.

CNRS *Abdoulaye Gamatié*

Contents

Part I Real-Time and Synchronous Programming

1 Generalities on Real-Time Programming 3
1.1 Embedded, Reactive, and Real-Time Systems 3
1.1.1 Definitions and Examples 4
1.1.2 Some Important Design Issues 8
1.2 Dealing with Time During System Execution 11
1.3 Real-Time Programming Models 12
1.3.1 Asynchronous Vision 13
1.3.2 Preestimated Time Vision 14
1.3.3 Synchronous Vision 15
1.3.4 Summary 16
1.4 Methodological Elements for System Design 17
References 20

2 Synchronous Programming: Overview 21
2.1 Objectives 21
2.2 Foundations 22
2.2.1 The Synchronous Hypothesis 22
2.2.2 Monoclocked Versus Multiclocked System Models 24
2.2.3 Implementation Models 26
2.3 Imperative Languages 27
2.3.1 ESTEREL 27
2.3.2 Graphical Languages: STATECHARTS, STATECHARTS, and ARGOS 30
2.4 Declarative Languages 32
2.4.1 Functional Languages: LUSTRE and LUCID Synchrone 33
2.4.2 The Relational Language SIGNAL 36
2.5 Other Languages 36
2.6 Summary 37
References 38

Part II Elementary Concepts and Notations of SIGNAL

3 Basics: Signals and Relations 43
3.1 Signals and Their Elementary Features 43
3.1.1 Definition of Signals 43
3.1.2 Data Types 44
3.1.3 Identifier of a Signal 50
3.1.4 Declaration of Signals 50
3.1.5 Constant Signals 51
3.2 Abstract Clock of a Signal 51
3.2.1 The Notion of Polychrony 51
3.2.2 Definition 54
3.2.3 The Event Type 54
3.3 Relation Between Signals 55
3.3.1 Equational Specification of Relations 55
3.3.2 Primitive Monoclock Relations 55
3.3.3 Primitive Multiclock Relations 59
3.4 Exercises 60
References 61

4 Programming Units: Processes 63
4.1 Elementary Notions 63
4.1.1 Definition of Processes 63
4.1.2 A Specific Process: Function 64
4.1.3 Programs 64
4.2 Primitive Operations on Processes 64
4.2.1 Composition 65
4.2.2 Local Declaration 65
4.3 Notation 66
4.3.1 Process Frame 66
4.3.2 Example: A Resettable Counter 67
4.3.3 Hierarchy of Processes 69
4.3.4 Label of a Process 69
4.4 Exercises 70
References 71

5 Extended Constructs 73
5.1 Pure Control Specification 73
5.1.1 Extraction of Abstract Clocks 74
5.1.2 Synchronization of Abstract Clocks 74
5.1.3 Set Operations on Abstract Clocks 75
5.1.4 Comparison of Abstract Clocks 77
5.2 Memorization 77
5.3 Sliding Window 78
5.4 Array of Processes 79

5.5 Exercises 79
Reference 81

6 Design in POLYCHRONY: First Steps 83
6.1 The POLYCHRONY Design Environment 83
6.1.1 What Is It Useful for? 83
6.1.2 What Tools Does It Provide? 84
6.2 Design of a Watchdog 84
6.2.1 Definition of a SIGNAL Specification 85
6.2.2 Compilation and Code Generation 86
6.2.3 Behavioral Simulation 88
6.3 Exercises 91
Reference 92

Part III Formal Properties of SIGNAL Programs

7 Formal Semantics 95
7.1 An Operational Semantics 95
7.1.1 Preliminary Definitions 96
7.1.2 Primitive Constructs on Signals 98
7.1.3 Primitive Constructs on Processes 100
7.2 A Denotational Semantics 102
7.2.1 A Multiclocked Semantic Model 102
7.2.2 Primitive Constructs on Signals 105
7.2.3 Primitive Constructs on Processes 107
7.3 Exercises 107
References 108

8 Formal Model for Program Analysis 109
8.1 The Synchronization Space: $\mathcal{F}_3$ 109
8.1.1 Encoding Abstract Clocks and Values 110
8.1.2 Encoding Primitive Constructs 110
8.1.3 Encoding Some Extended Constructs 112
8.1.4 General Form of an Encoded Program 114
8.2 Conditional Dependency Graph 115
8.2.1 Dependencies in Primitive Constructs 116
8.2.2 Example: Checking Dependency Cycles 117
8.3 Exercises 118
References 119

9 Compilation of Programs 121
9.1 Overview 122
9.2 Abstract Clock Calculus: Analysis of Programs 123
9.2.1 Typical Program Analysis Issues 123
9.2.2 Hierarchy Synthesis for Abstract Clocks 130

9.3 Exploiting Hierarchies of Abstract Clocks in Practice 132
9.3.1 Endochronous Programs 132
9.3.2 Exochronous Programs 135
9.3.3 Endochronization of Exochronous Programs 136
9.4 Code Generation 141
9.4.1 An Example of a Generated Code Sketch 142
9.5 Exercises 143
References 144

Part IV Advanced Design in SIGNAL

10 Advanced Design Concepts 149
10.1 Modularity 149
10.2 Abstraction 150
10.2.1 External Processes 150
10.2.2 Black Box Model 152
10.2.3 Gray Box Model 153
10.3 Oversampling 155
10.4 Assertion 156
10.4.1 Assertion on Boolean Signals 156
10.4.2 Assertion on Clock Constraints 157
10.5 Exercises 158
References 158

11 GALS System Design 159
11.1 Motivation in Some Application Domains 159
11.2 Theoretical Foundations 160
11.2.1 Endochrony 161
11.2.2 Endo-isochrony 164
11.3 A Distribution Methodology 167
11.4 Exercises 169
References 170

12 Design Patterns 171
12.1 Refinement-Based Design (Top-Down) 171
12.1.1 A Blackboard Mechanism 172
12.1.2 SIGNAL Modeling of read_blackboard 173
12.2 Incremental Design (Bottom-Up) 176
12.2.1 A FIFO Message Queue 177
12.2.2 SIGNAL Modeling of the FIFO Queue 177
12.3 Control-Related Aspects 180
12.3.1 Finite State Machines 180
12.3.2 Preemption 182

12.4 Oversampling 183
12.4.1 Euclid's Algorithm for Greatest Common Divisor Computation 183
12.4.2 SIGNAL Modeling of the Algorithm 184
12.5 Endo-isochrony 185
12.5.1 A Flight Warning System 185
12.5.2 SIGNAL Modeling of the FWS 185
12.6 Exercises 189
References 190

13 A Synchronization Example Design with POLYCHRONY 191
13.1 The Dining Philosophers Problem 191
13.1.1 Informal Presentation 191
13.1.2 A Solution 192
13.2 Design of the Solution Within POLYCHRONY 192
13.2.1 Modeling of Philosophers 193
13.2.2 Modeling of Forks 203
13.2.3 Coordination Between Philosophers and Forks 205
13.3 Exercises 209
References 209

A Main Commands of the Compiler 211
A.1 Compilation Commands 211
A.1.1 General Synopsis 211
A.1.2 Commonly Used Compiler Options 212
A.1.3 Examples of Command Usage 213
A.2 Automatic Makefile Generation 215
A.2.1 Synopsis 216
A.2.2 Examples of Command Usage 216

B The Grammar of SIGNAL 217

Glossary 231

Solutions to Exercises 235

Index 257

Abbreviations

AFCET	Association Française pour la Cybernétique Économique et Technique
APEX	Application executive
ARINC	Aeronautical Radio Inc.
CMA	Centre de Mathématiques Appliquées
CNET	Centre National d'Études des Télécommunications
CNRS	Centre National de la Recherche Scientifique
ESPRESSO	Environnement de Spécification de Programmes Reactifs Synchrones
FSM	Finite state machine
FWS	Flight warning system
GALS	Globally asynchronous, locally synchronous
GCD	Greatest common divisor
I3S	Informatique Signaux Systèmes de Sophia Antipolis
INRIA	Institut National de Recherche en Informatique et Automatique
IRISA	Institut de Recherche en Informatique et Systèmes Aléatoires
LRI	Laboratoire de Recherche en Informatique
SOC	System-on-chip
TNI	Techniques Nouvelles pour l'Informatique

Chapter 1
Generalities on Real-Time Programming

Abstract This introductory chapter presents general notions about real-time embedded systems. Section 1.1 first defines what embedded, reactive, and real-time systems are. Then, it discusses some important issues that often have to be dealt with during the design of these systems. Then, Sect. 1.2 briefly focuses on the temporal aspects during the execution of a real-time system. It illustrates a typical situation where such a system controls a physical process. This illustration aims to serve as a reasoning basis to address the link between the different perceptions of time from the viewpoints of the physical process and the real-time system. On the basis of various representations of time in real-time systems, Sect. 1.3 describes three main programming models: the asynchronous, preestimated time, and synchronous models. Finally, Sect. 1.4 discusses a few methodological concerns for the design of real-time embedded systems.

1.1 Embedded, Reactive, and Real-Time Systems

Embedded systems in general are ubiquitous and pervasive in the modern technological landscape. Two representative examples from our daily life incorporating such systems are cellular phones and cars. The last-generation cellular phones provide a user with highly sophisticated functionalities, including communication functions, video streaming, music, and Internet access, with a very satisfactory quality of service. This partly became possible thanks to the significant improvement of on-chip integration capacities. The same observation holds for cars, e.g., the Toyota Prius hybrid automobile, in which the electronic part of the whole system is more important in comparison with that in a classic car. One main reason is that the constructors of modern cars want to assist users in driving cars while improving the safety, the comfort, the ecological impact on the environment, etc.

More generally, embedded systems are found in domains such as telecommunications, nuclear power plants, automotive vehicles, avionics, and medical technology. The functions achieved by these systems are often application-domain-specific.

A. Gamatié, *Designing Embedded Systems with the SIGNAL Programming Language: Synchronous, Reactive Specification*, DOI 10.1007/978-1-4419-0941-1_1,
© Springer Science+Business Media, LLC 2010

The current section introduces a few basic definitions of embedded systems together with illustrative examples. Then, a survey of some critical design issues is given to motivate the need for adequate approaches.

1.1.1 Definitions and Examples

1.1.1.1 Embedded Systems

Over the past few decades, there has been a wide and rich literature about embedded systems, which proposes several definitions for such systems. In my opinion, among these definitions, the version proposed by Henzinger and Sifakis [5] is one of the most complete. The vision considered below conforms to theirs.

Definition 1.1 (Embedded system). An embedded system is a special-purpose computer system that consists of a combination of software and hardware components that are subject to physical constraints. Such physical constraints come from the system's environment and its execution platform.

An embedded system is in interaction with its outside world, also referred to as its environment, which may be, for example, a physical process, some technical devices in a larger system, or some human operator. Its function consists in supplying its connected environment with specific services: the observation, the supervision, or the control of an industrial plant. This particular function of an embedded system makes it a special-purpose computer system, in contrast to more general purpose computer systems such as desktop computers or laptops.

For the embedded system to achieve its function, each part of it provides the designer with specific advantages:

- The software part enables one to reprogram the functionality of a system for various purposes. It therefore favors design flexibility.
- The hardware part enables one to obtain better execution performances and significantly increases the system capability to satisfy physical constraints (e.g., deadlines, speed, power energy, memory).

The way the software part is programmed can also have an impact on the performances of the system. This is particularly true for the scheduling policies that define how software tasks gain access to hardware resources to compute the outputs of the system.

Example 1.1 (System-on-chip). Modern embedded systems tend to concentrate the components of a computer and the necessary electronic circuits into a single integrated circuit, called a microchip. Such systems are referred to as *system-on-chip* (SoC). The functions contained in a SoC can be digital, analog, mixed-signal, or radio-frequency. There are consumer electronic devices that are small and complex with very high capacities of processing power and memory (e.g., a digital camera).

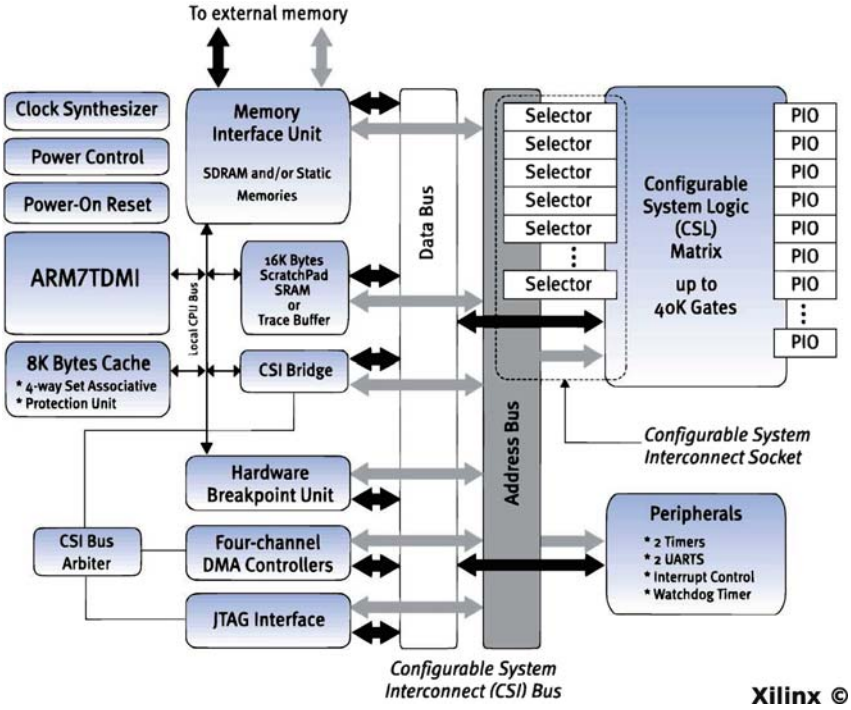


Fig. 1.1 An example of an embedded system: system-on-chip

Figure 1.1 illustrates a simple, yet typical example[1] of a SoC. Such a system integrates a static part (on the left-hand side) and a configurable (dynamic) part (on the right-hand side). Among the components of the static part, one can mention a general-purpose processor, ARM7TDMI, which offers interesting capabilities in terms of execution performances. This processor is associated with an 8KB cache memory. There is also an input/output block, a memory interface unit, which is in charge of exchanging data with the external memory, and a static fast memory (16 KB ScratchPad static random access memory). All these components are connected via a local CPU bus.

The configurable part, composed of the configurable system logic matrix and its associated communication interface, offers flexibility in that it is used to implement some specific system functions requiring high-performance capabilities, which cannot be obtained only with the processor of the static part. Typical functions are fast Fourier transform and discrete cosine transform.

Both the static and the configurable parts are connected via data and address buses. Direct memory access controllers route data directly between external

[1] Figure owned by Xilinx, Inc., courtesy of Xilinx, Inc. © Xilinx, Inc. 1994–2008. All rights reserved.

interfaces and the memory to increase the data throughput of the SoC. In addition to these components, there are also peripherals such as watchdog timers and interrupt control.

1.1.1.2 Reactive Systems

When executed, embedded systems often *continuously* or *repeatedly* interact with their environment, e.g., a human, a physical process. They receive from this environment input events, and compute the corresponding output information, which is finally returned to the environment. They are repeatedly solicited by their environment and they must respond to the events received.

In reactive embedded systems [3], the rhythm according to which the system reacts is imposed by its environment (see Fig. 1.2). Such a system differs from *transformational* systems, which require all their inputs, compute the corresponding outputs, and terminate. A typical example of a transformational system is a language compiler.

Definition 1.2 (Reactive system). An embedded system is said to be reactive when it continuously interacts with its environment that dictates the rhythm at which the reactions take place.

Example 1.2 (Overheating prevention in the core of a nuclear power plant). A reactive embedded system controlling a nuclear power plant must imperatively prevent the core from overheating [2]. For that purpose, it continuously interacts with the plant. According to its inputs provided by sensors, such a system must estimate as accurately as possible the current state of the core. Then, according to the predictable evolutions of the core from its current state, the system may alarm the operators if some overheating is foreseeable, and it may produce adjustment commands to actuators.

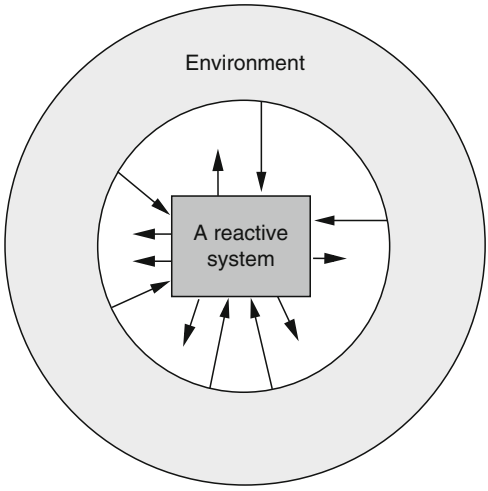


Fig. 1.2 A reactive embedded system

During the continuous interaction with their physical environment, reactive embedded systems must often react fast enough to satisfy environment constraints, typically on deadlines, jitters, and throughputs. The correctness of the system reactivity also depends on its execution platform characteristics, e.g., the speed of the processors or the frequency of the buses used in the platform.

1.1.1.3 Real-Time Systems

The aforementioned characteristics of both the system environment and the platform are used to define the *real-time* requirements that reactive embedded systems have to meet.

Definition 1.3 (Real-time system). A reactive embedded system is said to be a real-time system when its correctness depends not only on the logical results of its associated computations, but also on the delay after which these results are produced.

Let us focus on Fig. 1.3. It illustrates an interaction between a physical process Φ representing, for instance, the critical part of a nuclear power plant, and a logical process Λ representing a real-time embedded system (i.e., a computer system). Λ receives inputs from Φ at the instant τ_1. It computes a specific function that takes a certain delay, then produces an output, which is sent to Φ at the instant τ_2. In addition to the correctness of the output, the quantity $\Delta_\tau = (\tau_2 - \tau_1)$ must not exceed some value r imposed by Φ. This requirement on the execution of Λ represents its associated real-time constraint: $\Delta_\tau \leq r$. The value r is generally a constant even though it may be variable in some contexts (e.g., when the execution of a system tolerates several modes with a variable criticality level).

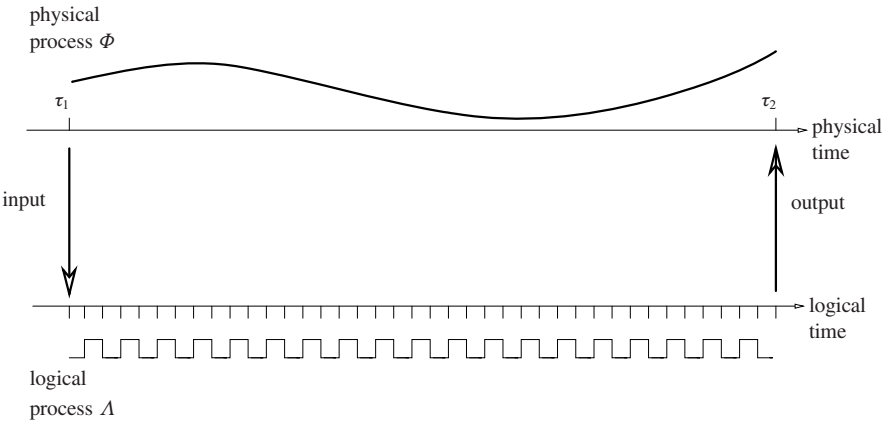


Fig. 1.3 Real-time execution of a single reaction

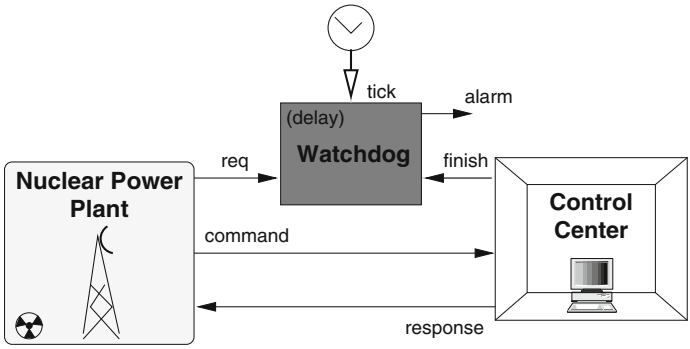


Fig. 1.4 Timely control of a nuclear power plant

In the scenario depicted in Fig. 1.3, the size of the value taken by the constant r depends on the critical nature of the response time in the physical process Φ. For instance, it may vary from microseconds in a system aboard an aircraft to seconds in a basic digital camera.

Example 1.3 (Timely control of a physical process). Let us consider the situation depicted by Fig. 1.4, which deals with an interaction between different processes. A system called "Control Center" is in charge of tracking a nuclear power plant. For a correct implementation, the timing requirements on this system are such that whenever it receives a command from the nuclear power plant, it must perform some computations within a constant duration. Typically, such computations may consist in analyzing received commands, then executing them.

To check whether or not timing requirements are met, a "watchdog" process is used to evaluate the time duration between the output of each command by the nuclear power plant (input "req") and the end of the corresponding treatment by the Control Center (input "finish"). This duration information is calculated on the basis of an external physical clock. When the duration is greater than the imposed constant duration, represented by "delay" as a parameter in the watchdog process, an alarm event is immediately emitted.

A similar context in which physical processes interact with real-time systems is industrial automation, which concerns fields such as processing, manufacturing and power industries, industrial robots, and embedded mechatronic systems (these systems comprise technologies from different engineering disciplines in terms of computer software and hardware, mechanics, and automatic control).

1.1.2 Some Important Design Issues

Real-time embedded systems have specific features that justify the need for well-suited design approaches, enabling them to meet their requirements. Among the important issues to be addressed, the following ones can be mentioned.

1.1.2.1 Hard Versus Soft Real Time

According to the degree of tolerance admitted by a real-time system regarding the nonrespect of its specified temporal constraints during the execution, it can be qualified[2] as *hard real-time* or *soft real-time*.

In hard real-time systems, the violation of any constraint leads to catastrophic consequences. The control systems of a nuclear power plant or the piloting system of an airplane are typical examples of such systems. In soft real-time systems, there is more flexibility in that a violation of their temporal constraints is not necessarily serious. Such systems only lead to low quality of service or poor performances. The developer of these systems is rarely required to rigorously prove that they meet all their real-time requirements. For instance, this is the case for a television, where a time lag of a few milliseconds between the images and the sound or the speech may be reasonably acceptable to a user. Note that most systems combine both hard and soft real-time subparts.

System reliability is essential

Northeast Power Blackout, August 2003. *The Task Force also found that FirstEnergy did not take remedial action or warn other control centers until it was too late, because of a computer software bug in General Electric Energy's Unix-based XA/21 energy management system that prevented alarms from showing on their control system. This alarm system stalled because of a race condition bug.*

European Ariane 5 rocket, June 1996. *Ariane 5's first test flight (Ariane 5 Flight 501) on 4 June 1996 failed, with the rocket self-destructing 37 s after launch because of a malfunction in the control software, which was arguably one of the most expensive computer bugs in history. A data conversion from 64-bit floating point to 16-bit signed integer value had caused a processor trap (operand error).*

Source: Wikipedia, January 2009
http://www.topless.eu/northeast_blackout_of_2003_en.html;
http://en.wikipedia.org/wiki/Ariane_5.

1.1.2.2 Safety Criticality

Embedded systems are often safety-critical as is the case for a large number of hard real-time systems. Soft real-time systems are not always safety-critical. In safety-critical systems, a failure can lead to loss of life, loss of mission, or serious financial consequences. The DO-178B standard dedicated to certification issues for safety-critical systems defines five levels of safety. These levels describe the consequences of a potential fault in such systems: A – the most critical level, a fault leads to catastrophic consequences, B – leads to severe consequences, C – leads to major consequences, D – leads to minor consequences, and E – leads to

[2] The terms "hard" and "soft" do not refer at all to hardware and software, respectively.

consequences without any effect on the system. The ability to prove the *reliability* of safety-critical embedded systems is a *crucial* task in their development.

Dealing with the next two issues is very interesting because it permits us to indirectly address the previously mentioned issues.

1.1.2.3 Determinism

Determinism is a very important property of embedded systems. It allows the designer to ensure that the system will always behave in the same manner, with respect to its expected functional requirements. This is necessary for the system validation.

1.1.2.4 Predictability

Beyond the functional requirements, one must be able to predict at least the critical nonfunctional behavior of an embedded system. Typically, for hard real-time embedded systems, the issue of guaranteeing that timing constraints are always satisfied, i.e., their predictability, is of high importance. Timing constraints can be quantitatively distinguished from several points of view: (1) *response time* or *latency* constraints, which involve the boundness of the durations between the occurrence of an event and the end of the induced system reaction; (2) *rate* constraints, which rely on the number of events processed by a system during a time period. Response time and rate constraints are, respectively, expressed over chronometrical and chronological representations of time.

1.1.2.5 Distribution and Heterogeneity

For various reasons, real-time embedded systems may be distributed: computation performances, geographical delocalization, fault tolerance, etc. This obviously leads to global design issues within the resulting large systems, such as the reliable communication between the distributed subsystems and the guarantee of real-time execution constraints locally to a subsystem and globally to the whole system. Synchronization of physical clocks (agreement on exactly what time it is) in distributed systems is one of the biggest issues. The frequency of quartz crystal oscillations that is usually considered in each subsystem to define local clocks depends on several criteria: the kind of crystal, the manner in which it is cut, the amount of tension it receives when working. This makes it difficult to deal with clock drifts. The subsystems can be a priori of different natures or heterogeneous: software and hardware components working together, components coming from different equipment suppliers, components having different instruction sets, hardware architectures of components with specific characteristics, e.g., faster clock cycles and higher memory capacity, etc.

1.1.2.6 Complexity and Modularity

Modern real-time embedded systems increasingly manage at the same time several functionalities or applications that are highly sophisticated. Typical examples are the last-generation cars and aircraft, which integrate more and more computer systems to ensure several feature-rich functionalities. In the specific case of advanced automobiles, the control system is already composed of 30–60 electronic control units [7] that are involved, e.g., in the achievement of the anticollision functionality of a car. A similar observation can be made for aircraft, in which physical control is being massively replaced by electronic *fly-by-wire* systems. This is the case for the Airbus A380 airplane. The pilot's commands consist of electronic signals that are processed by flight control computers, which realize the suitable actions with regard to actuators. Fly-by-wire is mainstream enough to be adopted in automotive vehicles (e.g., the Toyota Prius hybrid automobile). On the other hand, recent technological advances have enabled an increase in the number of transistors (up to two billion in 2009, e.g., the new Larrabee chip from Intel) in a single chip. The advantage is that the execution performances can be significantly improved. However, the other side of the coin is that all these improvements lead to a growth of system complexity. As a consequence, their design becomes very difficult. System modularity is highly sought to overcome the design complexity.

To address the aforementioned issues during the design of real-time embedded systems, the choice of a reliable modular approach is very important. More specifically, the question about the choice of an adequate programming model is highly relevant. The next two sections focus on different propositions from the literature for representing timing aspects in real-time programming.

1.2 Dealing with Time During System Execution

A real issue in real-time programming is how to deal with timing information. All languages defined for this purpose integrate time as part of their syntax and semantics. This integration takes different forms according to the abstraction level adopted.

Let us consider again the previous interaction scenario between the physical process Φ and the logical process Λ, now illustrated in detail in Fig. 1.5. The logical process Λ evolves with respect to a discrete time dimension referred to as *logical time*. From the viewpoint of Φ, the logical time coincides with the physical time only at instants τ_1 and τ_2. Between these two instants, the logical time can be considered in different ways with respect to the physical time, from the point of view of the logical process.

In the detailed scenario illustrated in Fig. 1.5, after being executed on processor p_1, the process Λ performs a request on a semaphore controlling the access to a given resource. This request generates a waiting delay, which can be unbounded, leading to the *discontinuousness* of the logical time (dashed line). Once the

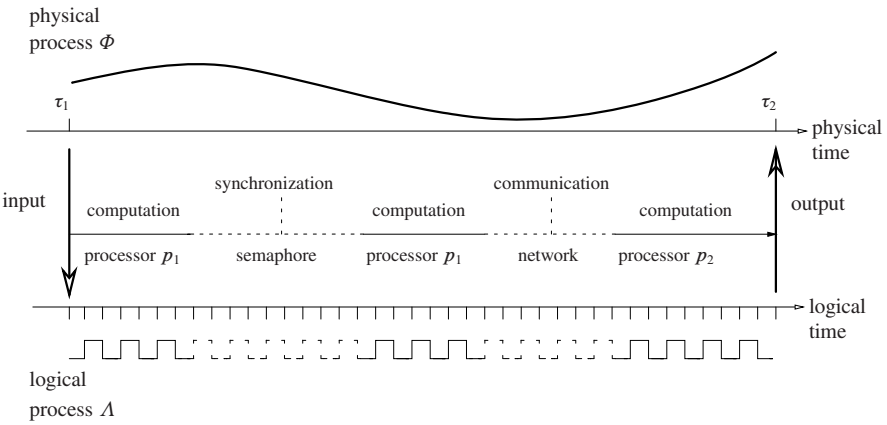


Fig. 1.5 Detailed real-time execution

semaphore becomes available, Λ continues its execution on the same processor. Then, it sends its output to another processor, p_2, through a communication network, which also contributes to making again the logical time discontinuous because of a resulting delay. Finally, the process can finish its execution on p_2 by returning its outputs to Φ.

Through this simple scenario, one observes that the relation between the physical and the logical notions of time depends on several factors, among which are the performance and usage of the execution hardware platform, the scheduling and synchronization strategies, the communication protocols, and possible optimization of programs and compilers.

1.3 Real-Time Programming Models

Depending on the time representation chosen during the design of a real-time system, the analysis of its functional properties as well as of its nonfunctional properties, e.g., temporal properties, can be more or less efficient and facilitated for validation purposes.

The usual time representations can be classified[3] into three models: *asynchronous*, *preestimated time*, and *synchronous*. Roughly speaking, these models mainly differ from each other in their way of characterizing the relation between the physical time and the logical time within a correct system behavior.

To illustrate the main idea of each model, let us consider as a running example the very simple representation of a real-time system, called RTS in Fig. 1.6. The inputs and outputs of RTS are respectively denoted by i and o.

[3] We discuss different time representations according to Kirsch [8]. The reader can find further interesting classifications of time models for embedded system design in [6].

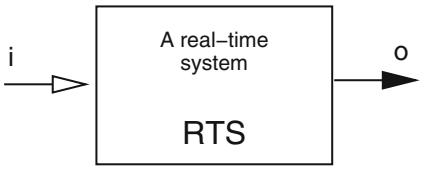


Fig. 1.6 A simple real-time system

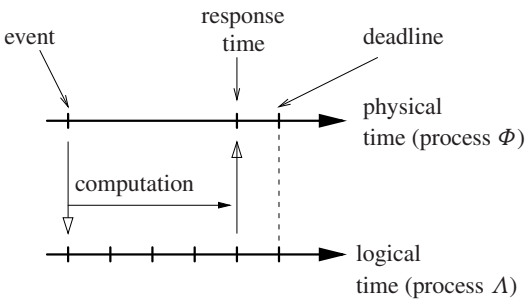


Fig. 1.7 Asynchronous model: logical duration ≤ deadline

1.3.1 Asynchronous Vision

The asynchronous model is the classic one used by developers of real-time systems. A system is represented by a program which consists of a finite number of *tasks*. These tasks execute concurrently to fulfill the system function, under the supervision of specific mechanisms such as *real-time operating systems*. Developers usually consider popular languages such as Ada, C, and Java to define the programs implementing real-time systems.

In the situation depicted in Fig. 1.7, the logical process Λ is represented by a set of tasks whose execution depends on the scheduling policy adopted by the associated real-time operating system. In general, this device relies on low level-mechanisms such as interrupts and physical clocks. From the viewpoint of the logical time, the behavior of the system is strongly influenced by the execution platform: among the factors to be taken into account to determine the execution time of the process are the scheduling policy of the tasks composing the process, the performances, and the utilization ratio of processors. Thus, the asynchronous vision yields *platform-dependent* models.

As a consequence, the logical execution time of process Λ is a priori unknown since the aforementioned factors are variable. This induces a *temporal nondeterminism*, which is an outstanding feature of the asynchronous model compared with the other models presented in the next sections. In practice, to fix the execution bounds within a logical temporal behavior of a system, physical *deadlines* are considered. The nonrespect of these deadline constraints can be tolerated or not, according to the criticality level of the task concerned. In Fig. 1.7, the interval of logical time associated with the execution of the process respects the deadline constraint since the response time is observed before the deadline time.

Let us consider the asynchronous interpretation Λ_a of the behavior of the RTS system (see Fig. 1.6). Whenever inputs *i* are received at some instant τ_1, the duration required to produce outputs *o* at some instant $\tau_2 \geq \tau_1$ depends on the availability of resources in the execution platform chosen . Thus, this duration is variable and cannot be a priori determined. This makes the predictability analysis of the system very complicated.

1.3.2 Preestimated Time Vision

Here, the relevant interactions only occur at τ_1 and τ_2. So, according to the vision adopted in the preestimated time model (also referred to as the *timed model*; see Fig. 1.8) no interaction takes place between processes Φ and Λ within the time interval $]\tau_1, \tau_2[$. The duration observed within $]\tau_1, \tau_2[$ is considered as being a constant and strictly positive amount of time, representing exactly the duration of computation and communications performed by Λ between τ_1 and τ_2. The models resulting from this vision are therefore *platform-dependent* as in the asynchronous vision. The main difference here is that the timing information is a priori fixed, so it can serve for the reasoning about the temporal behavior of a system.

Several formalisms of specification of real-time systems rely on this model. Timed automata are a typical example. A state is annotated with timing information representing the amount of time during which the system represented can potentially remain in such a state. Among programming languages that adopt the timed model, we can mention *Giotto* [4], used in the design of embedded control systems.

On the basis of the convention given above, a program describing the system is annotated with logical duration information, specified by the designer, based on particular criteria. This information indicates an approximation of the real duration of computations and communications. In general, the values chosen are ideal since they are expected to perfectly match physical time durations. A main advantage of the timed model is that the verification of certain properties related to the logical

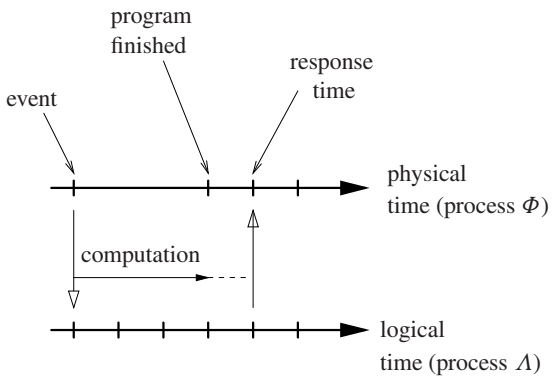


Fig. 1.8 Timed model: logical duration = physical duration

temporal behavior of the system is made possible earlier. For instance, the validity of a scheduling event can be checked at the compilation step since most of the necessary information is known before the execution. This is not the case with the asynchronous model, in which durations are not assumed to be known before execution. The timed model is very suitable for predictability analysis in the design of real-time embedded systems.

Let us consider again the RTS system. In contrast to the asynchronous vision, here, whenever inputs i are received, the duration Δ_τ required to produce the outputs o is assumed to be known. So, two particular situations can arise during the execution of the system: (1) the system may not have finished executing before its preestimated execution duration was already over; (2) it may have completed its execution before its annotated logical execution duration is reached. In the former case, an exception is usually generated, whereas in the latter case, the production of the outputs is delayed until the logical execution duration is reached (see Fig. 1.8). The latter case has a particularly significant impact on the quality of the system implementation. The closer the execution time of a system is to its preestimated execution duration, the better is the associated timed model since the time difference between the end of computations and the corresponding output production is minimized. As a result, the trade-off regarding the choice of suitable preestimated execution times is an important issue in timed models.

1.3.3 Synchronous Vision

In the synchronous model (see Fig. 1.9), the logical temporal reference is completely determined by the successive reactions of the system happening on the occurrences of observed events. ESTEREL, LUSTRE, and SIGNAL are typical examples of synchronous languages [1].

The basic idea of the synchronous model is similar to that of the preestimated timed model, in the sense that process Φ and process Λ do not communicate between instants τ_1 and τ_2. However, the main difference is that in the synchronous

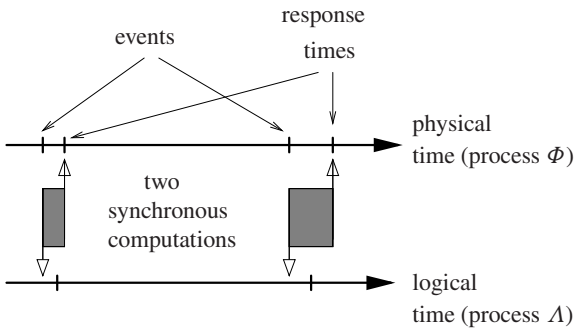


Fig. 1.9 Synchronous model: logical duration = 0

model, on the occurrence of input events, the system is supposed to react fast enough to produce the corresponding output events before the next input events arrive: *synchrony hypothesis*. As a result, the interval $[\tau_1, \tau_2]$ is considered as entirely being part of a single "logical instant" in the synchronous model. The remaining central question is whether or not the execution platform of the system is sufficiently powerful to satisfy the assumption. For this purpose, one should complement the synchronous model with actual execution time information. Thus, the synchronous model is *platform-independent*.

There is a very popular "idealized" picture of the synchrony hypothesis, which states that the communication and computing durations between instants τ_1 and τ_2 are considered as null, i.e., the execution of a system is instantaneous. Of course, this picture is unrealistic and may lead to confusion.

Even though the synchronous model abstracts away the quantitative temporal aspects of a system, it is relevant enough for system validation. Typically, the fact that a program reacts to its environment either every minute or every second (with a different physical synchronization) does not change the inherent functional properties of this program. Moreover, these properties also remain unchanged when the frequency of a processor that executes the program varies. Thus, the synchronous vision offers a deterministic representation of an embedded system, which favors the trustworthy and easier analysis of the functionalities.

Remark 1.1. The synchronous model exhibits a multiform nature of time by considering the behavior of a system through the simultaneousness and the precedence of observable events. For instance, one can equivalently state that a particular event of a given system will occur within 30 min or after 25 km by simply counting 30 or 25 occurrences of some measurement events denoting "minutes" or "kilometers."

The synchronous interpretation of the RTS system behavior assumes that the outputs o will be always produced fast enough to meet the timing requirements of the system. It therefore considers that the outputs are computed instantaneously with the receipt of inputs i. This allows one to focus only on the functional properties of the system.

1.3.4 Summary

As illustrated in Fig. 1.10, the synchronous vision for the programming of real-time systems offers a high abstraction level, whereas the asynchronous vision provides the programmer with a lower abstraction level. Each level enables one to suitably address specific aspects of a system: functional aspects with the synchronous vision and nonfunctional aspects with the asynchronous vision. They are therefore complementary. The preestimated time vision aims to take advantage of the synchronous and asynchronous visions. It could be a good compromise provided that the timing approximations are relevant enough, which is not trivial to guarantee.

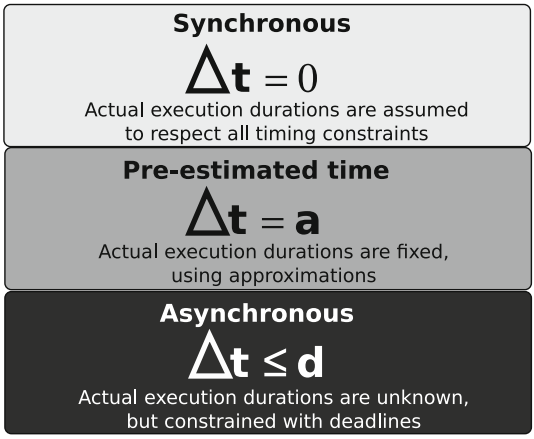


Fig. 1.10 A summary of real-time programming models

1.4 Methodological Elements for System Design

The way time is dealt with during the design of real-time embedded systems leads to the challenging question of the suitable approaches to be adopted. The required approaches must allow the designer to guarantee the requirements of these systems (see Sect. 1.1.2).

A basic solution consists in separating concerns with regard to the behavioral properties of a real-time embedded system:

- On the one hand, its *functional properties*, i.e., properties which state that the results obtained after the system is executed hold the expected values. Typically, these properties characterize *what* function is achieved by the system. Let us consider the Control Center system of Example 1.3. The functional properties of this system describe what values are expected in response to the receipt of a "command" from the nuclear power plant. For instance, if the output "response" of Control Center is of Boolean type, a possible functional property can state that *the value of this output is always either "true" or "false," depending on the value of the system input*.
- On the other hand, its *nonfunctional properties*, i.e., properties which depend on the platforms considered for the execution of the system, typically characterize *when* and *how* the system achieves its corresponding function. Let us consider again the Control Center system. We may require that the value of "response" be valid only when it is computed within a given amount of time after the input is received. So, the correctness of the Control Center behavior also concerns when its output is produced. Such a requirement denotes a nonfunctional property, and more precisely a temporal property. Other examples of functional properties concern memory and energy power.

From the above separation of concerns, we can draw a design methodology for real-time embedded systems in a similar way as a cardiologist establishes part of the diagnosis of heart healthiness with an electrocardiogram (see Example 1.4).

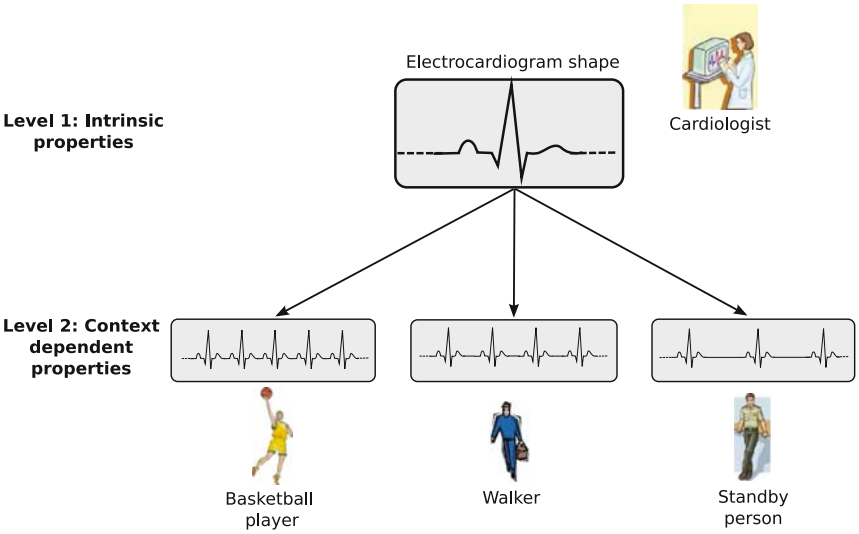


Fig. 1.11 Analysis of human heart

Example 1.4 (The cardiologist metaphor). Let us consider a heart examination scenario by a cardiologist as illustrated in Fig. 1.11[4].

A first relevant aspect in the electrocardiogram is the shape of the curve obtained on a single heart contraction. This shape allows the cardiologist to check whether or not the heart performs all the expected basic actions characterizing a normal cycle during the contraction of the heart: auricular contraction, ventricular contraction, auricular relaxation, and ventricular relaxation. At this stage of the diagnosis, the cardiologist can unambiguously detect anomalies, such as infarct, by just comparing the shapes recorded from different regions of the heart. The absence of anomalies most often[5] guarantees that all regions of the heart are effective. This can be seen as an *intrinsic property* of a healthy heart in contrast with a diseased heart, which can have, e.g., some dead regions. As a result, the shapes observed in an electrocardiogram can be interpreted as the *functional properties* of the analyzed heart.

Afterwards, the cardiologist can consider further aspects that concern the way the "functionally healthy" heart works *depending on different contexts*. As illustrated in Fig. 1.11, it could be interesting to check the cardiac frequency when the person examined is either in a static position (e.g., a standing person) or in a dynamic position (e.g., a basketball player, a walker). All these persons share the fact that the shapes of their hearts are not anomalous, as explained before. However, the frequency may vary from one person to another: it is likely that the pulse, which roughly consists of the number of heart beats per duration, is higher for the basketball player than for

[4] Free clipart images from http://www.bestclipart.com/

[5] The electrocardiogram interpretation of a relaxed person may, however, not be relevant enough. Some anomalies are only diagnosed after physical effort.

the walker. Notice the additional timing information that is now taken into account. At this stage of the diagnosis, the cardiologist is able to observe whether or not the heart suffers from further anomalies that are related to its functioning context on the basis of the cardiac frequency. Depending on standard pulse measurements, the cardiologist can select a convenient context (static or dynamic position) to observe the heart beating. These aspects that are related to the contexts can be seen as the *nonfunctional properties* of the analyzed heart.

By replacing the heart diagnosis by a real-time embedded system, the above examination process will consist of a methodology with: (1) a design step where the functional (i.e., intrinsic) properties of the system can be unambiguously described and analyzed; and (2) another step at which nonfunctional (i.e., context-dependent) properties can be addressed. The major advantage of such a methodology is that the designer can safely concentrate on different validation issues. Whenever the functional properties are proved to be satisfied by the system, they will hold for any execution platform chosen . Then, the designer can focus on the suitable platforms that satisfy the required nonfunctional properties.

The synchronous approach is adequate for the first step of design, whereas the asynchronous approach is more suitable for the second step. These approaches are perfectly complementary. The preestimated time approach rather aims at finding a compromise between both approaches.

Expected Learning Outcomes:

- An **embedded system** is a **special-purpose** computer system **constrained** by both its environment and its execution platform. If it often **continuously interacts** with its connected environment, it is said to be **reactive**. If its reactions are usually **temporally constrained**, it is said to be **real-time**.
- Among the major issues concerning the design of real-time embedded systems are safety criticality, determinism, predictability, scalability, heterogeneity, and complexity.
- When dealing with the design of a real-time system, there are different model representations of the temporal aspects:
  - **Asynchronous model**: the actual execution time is unknown, but is constrained by deadlines.
  - **Preestimated time model**: the actual execution time is assumed to be known (via an estimation).
  - **Synchronous model**: the actual execution time is abstracted away, and replaced by a qualitative representation of time.
- The **separation of concerns** within the design of real-time embedded systems is very important to better address the different properties of such systems. A possible decomposition consists in distinguishing **functional properties** from **nonfunctional properties**.

References

1. Benveniste A, Caspi P, Edwards SA, Halbwachs N, Le Guernic P, de Simone R (2003) The synchronous languages 12 years later. Proceedings of the IEEE 91(1):64–83
2. Gautier T, Le Guernic P, Maffeïs O (1994) For a new real-time methodology. Research report number 2364, INRIA. Available at http://www.inria.fr/rrrt/rr-2364.html
3. Harel D, Pnueli A (1985) On the development of reactive systems. Logics and models of concurrent systems. Springer-Verlag, New York, F-13:477–498
4. Henzinger TA, Horowitz B, Meyer Kirsch C (2001) Embedded control systems development with Giotto. In: Proceedings of The ACM Workshop on Languages, Compilers, and Tools for Embedded Systems (LCTES'2001), The Workshop on Optimization of Middleware and Distributed Systems (OM'2001), Snowbird, Utah, USA
5. Henzinger TA, Sifakis J (2006) The embedded systems design challenge. In: Formal methods (FM'2006), LNCS volume 4085, Springer, Heidelberg, pp 1–15
6. Jantsch A, Sander I (2005) Models of computation and languages for embedded system design. IEE Proceedings on Computers and Digital Techniques 2(152):114–129
7. Martin N (1998) Look who's talking: Motorola's C.D. Tam. Available at: http://findarticles.com/p/articles/mi_m3012/is_1998_Nov
8. Kirsch CM (2002) Principles of real-time programming. In: Sifakis J, Sangiovanni-Vincentelli A (eds) 2002 Conference on Embedded Software, EMSOFT'02, LNCS volume 2491. Springer, Heidelberg

Chapter 2
Synchronous Programming: Overview

Abstract This chapter gives an overview of synchronous programming through the presentation of the main existing languages together with their associated tools. As a complement to what was said in the previous chapter on the synchronous model, the objectives and foundations of the synchronous approach are first recalled in Sects. 2.1 and 2.2, respectively. Then, the synchronous imperative languages are presented in Sect. 2.3. The synchronous declarative languages are introduced in Sect. 2.4. Finally, further synchronous languages are briefly mentioned in Sect. 2.5.

2.1 Objectives

The synchronous languages [3, 13] were introduced in the early 1980s to enable the trusted design of safety-critical embedded systems. They are associated with mathematical models as semantic foundations. These formal models are adequate enough to allow one to unambiguously describe the behaviors of a system and to efficiently analyze and verify its properties for validation. Furthermore, they offer a suitable basis for reasoning about program transformations to permit the automatic construction of implementations depending on the functional requirements of a system.

More precisely, the synchronous languages aim at proposing the useful means to deal with the following design and programming issues:

- *Mathematical specifications.* Such specifications can be confidently considered to reason a priori about the system behaviors. As been mentioned before, it is paramount to ensure the safe and reliable development of such safety-critical systems by being able to demonstrate the correctness of their behaviors.
- *Determinism and predictability.* These, respectively, concern the specification of functional aspects and of timing aspects of a system (see Chap. 1). They play an important role for the trustworthy validation of designs.
- *Concurrency and hierarchy.* An embedded system often evolves concurrently with regard to its environment. On the other hand, it is commonly seen as being formed of several components running in parallel to achieve the task the system

A. Gamatié, *Designing Embedded Systems with the SIGNAL Programming Language: Synchronous, Reactive Specification*, DOI 10.1007/978-1-4419-0941-1_2,
© Springer Science+Business Media, LLC 2010

has been assigned. This is justified by the need for modularity or by geographical constraints. The overall architecture of the system can therefore be seen as some hierarchical structure resulting from the association of basic components.
- *Platform-independent designs.* During the definition of a system, its subparts that are platform-dependent must be reduced and clearly fixed as much as possible. Synchronous languages offer platform-independent descriptions, which are strongly retargetable to different implementation platforms, i.e., favoring *portability*.
- *Automatic code generation.* To avoid error-prone manual programming and tedious debugging tasks, the possibility to automatically generate code from high-level and proven correct specifications is a very interesting solution. All synchronous languages offer this facility.
- *Uniform design frameworks.* The synchronous languages provide the designer with a way to describe at a high level, using the same formalism, the functions or algorithms achieved by the system, and a model of the hardware infrastructure to be considered for implementation.

2.2 Foundations

2.2.1 The Synchronous Hypothesis

2.2.1.1 What Does It Mean?

The synchronous languages rely on a basic assumption which considers that on the occurrence of input events a system reacts fast enough to produce the corresponding output events before the acquisition of the next input events. This assumption is referred to as the *synchrony hypothesis*.

More precisely, a system is viewed through the chronology and simultaneity of the observed events during its execution. This is the main difference from classical approaches, in which the system execution is considered under its chronometric aspect, i.e., duration has a significant role. According to such a picture, a system execution is split into successive and nonoverlapping *synchronized actions* or *reactions*. Such a reaction is typically a system response to the receipt of some events from the environment. It can be also a regular and systematic production of some output events by a system towards its environment (such a system may only have output ports).

2.2.1.2 Illustration and Discussion

Figure 2.1 illustrates on the left-hand side the actual execution trace of a system that has two inputs *i1* and *i2* and one output *o*, and on the right-hand side a corresponding synchronous execution trace.

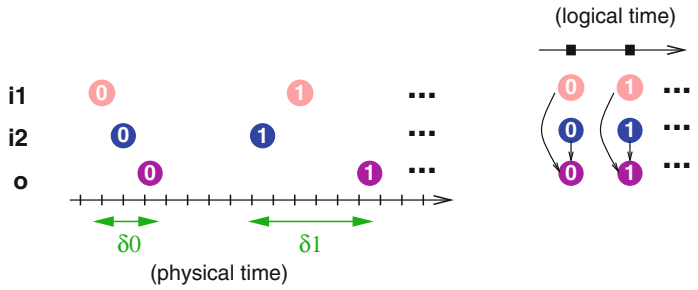


Fig. 2.1 Asynchronous (*left*) versus synchronous (*right*) observations

In the left-hand side trace, which we refer to as asynchronous, the observed events (represented by colored bullets) associated with *i1*, *i2*, and *o* are numbered with the identifier of the reaction in which they are involved. Depending on the reaction, one can observe that both the arrival order the input events and the computation time of the output events vary. For instance, in reaction 0, *i1* is observed before *i2* and inversely in reaction 1; in reaction 0, the computation of *o* requires $\delta 0$ time units, whereas in reaction 1 it requires $\delta 1$ time units. These variations lead to the temporal nondeterminism mentioned in Sect. 1.3.1.

Now, let us consider the corresponding synchronous trace shown on the right-hand side. By ignoring the actual execution time of reactions, we obtain an instantaneous execution. Each reaction denotes a single *logical instant* in the synchronous model, where the data dependencies between the observed events are expressed. For instance, the value of *o* depends on those of *i1* and *i2* as illustrated at each logical instant in the figure.

From this point of view, the execution represented is temporally deterministic. The functional properties of the system behavior can therefore be safely addressed. However, as soon as the system design is guaranteed to be correct using the synchronous model, an implementation phase is a posteriori necessary to validate the synchronous hypothesis on an actual execution platform on which the execution time of reactions is fully taken into account. This validation typically consists in proving that the platform considered enables sufficient execution performances, which ensure satisfactorily bounded delays of reactions with respect to real-time constraints.

Thus, the synchronous hypothesis can be seen as a suitable abstraction of a system execution, allowing one to unambiguously address the design issues by avoiding the temporal nondeterminism inherent in the usual asynchronous vision of real-time systems.

2.2.1.3 A Qualitative Way To Deal with Time

Although the synchronous model does not take into account quantitative aspects of timing information, it allows one to deal with the *ordering of observed events* in

the system as well as the *synchronizability* of these events. These two aspects are strongly related to the timing information.

Typically, logical instants can be seen as "time stamps" that indicate the date at which events occur. Since they are ordered (at least partially) with respect to a given reference set of instants, date comparison becomes possible. Hence, some event can be said to *occur later* than another event.

Furthermore, on the basis of the observed occurrences of events, one can also determine very easily the frequency at which events occur during an execution of the system. This information is captured by the notion of logical or *abstract clock*, used in the rest of this book. Intuitively, such a clock consists of a set of logical instants obtained from the synchronous vision introduced in Chap. 1, page 15. It serves to synchronize the occurrence rates of expected events in the system.

The ordering and synchronizability notions described above enable one to deal with some timing issues without any explicit reference to quantitative durations. This way of working can be qualified as a *qualitative* viewpoint.

2.2.1.4 Methodological Implications

Of course, the synchronous hypothesis is not completely realistic with respect to nonfunctional properties since it does not take into account the actual execution duration of the system. This means that a real-time system, proved to be functionally safe, must be validated on an execution platform to guarantee that the physical timing constraints are satisfied. In some sense, we can see the synchronous vision and the asynchronous vision as being complementary approaches for the development of real-time embedded systems: the former offers a way to guarantee that the system will a priori behave correctly from the functional viewpoint, and the latter enables one to ensure that nonfunctional properties such as execution durations are satisfied.

This complementarity is the aim of the preestimated time vision, introduced in Chap. 1. However, as mentioned previously, the major issue concerns the relevance of the preestimated execution duration with regard to a real-world execution.

2.2.2 Monoclocked Versus Multiclocked System Models

As discussed earlier, real-time embedded systems are sometimes composed of several components or subsystems. According to how the design of such systems is envisaged, one has a choice between different approaches to describe the interaction between the components.

A first approach consists in considering that the whole system holds a *global physical clock*, also referred to as a *master clock*, that indicates when a reaction takes place in the system. The set of reactions initiated by the clock of each component is therefore strictly a subset of the set of reactions initiated by the global clock. We refer to such a system as a *monoclocked* system, illustrated in Fig. 2.2.

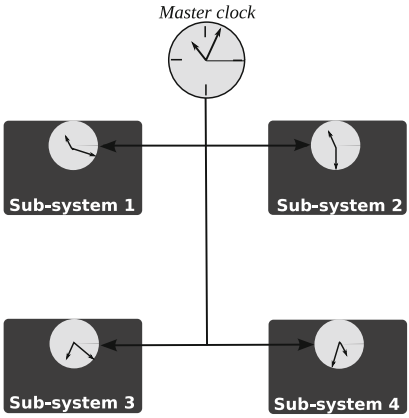


Fig. 2.2 A monoclocked system

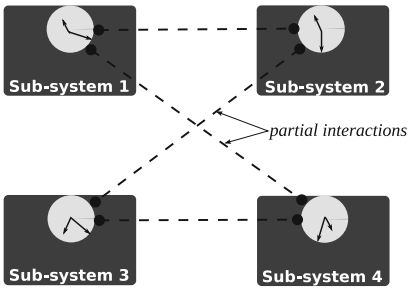


Fig. 2.3 A multiclocked system

There is a tight relation between all clocks of components and the global clock. As a result, whenever a property of a component clock is locally modified, the designer must also take care of what happens globally in the system since the relation between that clock and the global clock could be affected. This modification could even concern the clocks of the other components; typically some resynchronizations may be required. Such a clock organization in a system leads to a *monolithic design approach* because the designer should always keep a global focus on the system.

An alternative approach consists in considering that each component in the system holds its own activation physical clock and there is no longer a global clock. We refer to such a system as a *multiclocked* system, illustrated in Fig. 2.3.

A great advantage is its convenience for component-based design approaches that enable modular development of increasingly complex modern systems. The design of each component of a system can be addressed separately since a local modification of a component clock only concerns the clocks of components with which it interacts.

In accordance with Lamport's observation [21], when the different components of a multiclocked system do not interact, it is not necessary that their clocks be synchronized. The central question is not to agree on exactly what time it is, but rather to agree on the order in which the observed events occur and their possible coincidence.

In the remainder of the book, languages that consider the monoclocked view of a system are referred to as *monoclock languages* and those considering the multiclocked vision are termed *multiclock languages*.

2.2.3 Implementation Models

The most commonly used implementation models for synchronous languages are given in Fig. 2.4: *event-driven* and *clock-driven* executions.

The former expresses the fact that each reaction is initiated on the occurrence of some input event. For instance, in Fig. 1.4, this may consist in seeing the watchdog as only activated whenever it receives a request event *req* from the nuclear power plant. Then, "`compute reaction`" amounts to executing some statements that typically modify the variables manipulated in a program, and to computing the output data depending on the current state and input data. Finally, the next state of the program is updated via its associated memory information. Notice that this memory information is first initialized before it enters the main loop.

The latter implementation model differs from the former in that reactions are only initiated by abstract clock ticks. For instance, in Fig. 1.4, this may consist in seeing the watchdog as only activated whenever it receives an input $tick$ event from the external clock. We can notice that abstract clock ticks can be equivalently represented as *pure* input events, meaning events which only indicate that the system must react, but which do not carry all functional information required to compute the system outputs.

Both event-driven and clock-driven implementation models assume that all actions considered take *bounded memory and time* capacities.

The synchronous languages are associated with powerful compilers that enable automatic code generation from high-level specifications (i.e., synchronous programs). For that purpose, the specifications considered are necessarily required to be deterministic, in other words, the same input values always lead to the same output values when a system specified in these languages is executed. Given a system specification, the determinism property can be effectively checked with compilers.

Observe that nondeterministic synchronous specifications can be meaningful in high-level or partial design as can be conceived with the SIGNAL compiler.

```
initialize memory;
for each input event do
  compute reaction;
  update memory;
end;
```

```
initialize memory;
for each clock tick do
  read inputs;
  compute reaction;
  update memory;
end;
```

Fig. 2.4 Event-driven (*left*) and clock-driven (*right*) execution models

The synchronous languages can be classified into two families according to their programming style: *imperative languages* such as ESTEREL, SYNCCHARTS, and ARGOS use control structures and explicit sequencing of statements, whereas *declarative languages* such as LUSTRE and SIGNAL use equations that express either functional or relational dependencies.

2.3 Imperative Languages

2.3.1 ESTEREL

The ESTEREL language [6, 9] was defined at the Centre de Mathématiques Appliquées[1] (CMA) in Sophia-Antipolis, France, through the collaboration of two organizations: École Nationale Supérieure des Mines de Paris and Institut National de Recherche en Informatique et en Automatique (INRIA).

2.3.1.1 Language Features

The major advantage of a synchronous imperative language such as ESTEREL is its facilitation of the the modular description of reactive systems where control plays a predominant role. ESTEREL relies, on the one hand, on the usual *imperative* control structures such as *sequence* and *iteration* and, on the other hand, on further statements qualified as *reactive* whose semantics is based on the notion of logical instants at which a described system is supposed to react. The basic objects of the language are *signals* and *modules*.

A signal is characterized by its status at each logical instant: present or absent. It can either hold a value or not. In the latter case, it is referred to as a *pure* signal. There is a predefined pure signal in the language called `tick`, which corresponds to a *global abstract clock* that controls activations. This clock is assumed to be faster than all other abstract clocks in a system description. The set of instants at which a signal occurs is commonly termed its associated logical *clock*. The emission of a signal can be of two possible natures: either environmental, and the signal is an input of the program embedded in the emitting environment, or an output of another program with the instruction `emit`.

A module is a construct that defines the structure of an ESTEREL program. From a syntactical viewpoint, a module is composed of an interface, i.e., input and output signals, and a body, i.e., imperative and reactive statements that specify the behaviors encoded by the module. The body is executed instantaneously whenever the module is activated. An ESTEREL program can be seen as a collection for nested, concurrently running threads.

[1] http://www-sop.inria.fr/cma

Among the main characteristics of the ESTEREL language, one can mention the following:

- *Communication* and *synchronization*: They are achieved through an instantaneous broadcasting of signals between entities, which consist of modules.
- *Preemption* and *delay* mechanisms: Preemption is realized via special statements. For instance, in the expression `suspend P when s`, `P` is only executed at instants where the signal `s` is absent. The `pause` statement enables one to suspend an execution until the next reaction. One can that both statements are not instantaneous. Finally, there are some exception mechanisms in ESTEREL.
- Two kinds of *composition*: Given two statements `I1` and `I2`, they can be composed *sequentially*, noted `I1;I2`. In this case, `I2` is only executed after `I1` finishes its execution. The two statements can also be composed using the *parallel composition operator*, noted `I1 || I2`. The execution of the statement resulting from this composition terminates when both `I1` and `I2` terminate.

The main statements of the language [3, 6, 9] are summarized in Table 2.1. All the statements in a program are executed according to the global abstract clock, represented by the special signal called `tick`. However, a recent proposal, called "multiclock Esterel," aims at going beyond this single-clocked vision by extending the language with a clock notion that could allow one to describe systems with multiple clocks [7].

Table 2.1 A summary of basic ESTEREL statements

`emit s`	Makes signal `s` present immediately
`present s then P else Q end`	If signal `s` is present, performs `P` otherwise `Q`
`pause`	Stops the current thread of control until the next reaction
`P ; Q`	Runs `P` then `Q`
`loop P end`	Repeats `P` forever
`loop P each s`	Runs `P`, if `s` occurs while `P` is still active, then `P` is aborted and immediately restarted again; if `P` terminates before `s` occurs, one has to wait for `s` and restart `P` again
`await s`	Pauses until the next reaction in which `s` is present
`P || Q`	Starts `P` and `Q` together; terminates when both have terminated
`abort P when s`	Runs `P` up to: either `P` is finished, or a reaction (not included) in which `s` is present while `P` is not finished yet
`suspend P when s`	Runs `P` except when `s` is present
`sustain s`	Means `loop emit s; pause end`
`run M`	Expands to code for module `M`

```
module ABRO:
  input A, B, R;
  output O;
  loop
     [ await A || await B ];
     emit O;
  each R
end module
```

Fig. 2.5 The ABRO program in ESTEREL

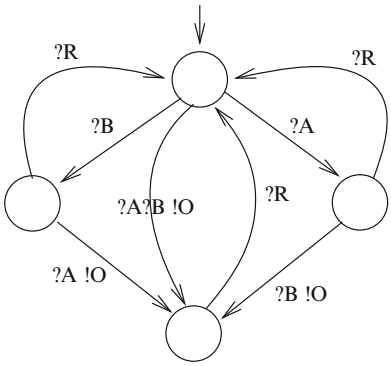


Fig. 2.6 A finite state machine specification of the ABRO process

Example 2.1 (ABRO). Figure 2.5 illustrates a well-known ESTEREL sample program called `ABRO` that specifies preemption. This program expresses in a very simple way a behavior with preemptions. Its inherent control flow is equivalently represented by the finite state machine illustrated in Fig. 2.6. We can observe the concision and the clarity of the ESTEREL program compared with the finite state machine representation.

This program emits the output signal `O` when the input signals `A` and `B` have been received, in any order (even simultaneously). Whenever the input signal `R` (playing the role of a "reset") is received, the behavior of the program is immediately reinitialized. Such a behavior is expressed by the finite state machine depicted in Fig. 2.6. In this figure, ? and !, respectively, denote receipt and emission of an event. The brackets "`[...]`" shown in Fig. 2.5 are simply used to solve syntactic priority conflicts.

2.3.1.2 Compilation of Programs

An interesting feature of synchronous languages is their capacity to enable the analysis of program properties. In particular, the consistency analysis of the set of abstract clock constraints allows one to verify typical behavioral properties such

as the absence of deadlock. Such an analysis is generally part of the compilation during which a program can be synthesized in formats that favor automatic code generation. For instance, an ESTEREL source program can be compiled into either automata [12], electronic circuits [5], or an intermediate format in the form of a control flow graph [25].

2.3.1.3 Application Domains and Tools

Typical application domains where ESTEREL has been used are automotive vehicles, aerospace and defense, rail transportation, semiconductors, and electronics for the design of embedded applications.

Academic versions of ESTEREL-based design platform, which includes compilers and verification tools, have been proposed by the CMA, France Télécom, and the University of Columbia (New York). The industrial version, Esterel Studio, has been commercialized by Esterel Technologies.

2.3.2 *Graphical Languages:* STATECHARTS*,* STATECHARTS*, and* ARGOS

2.3.2.1 STATECHARTS

The STATECHARTS formalism [17] was originally defined at the Weizmann Institute of Science[2] (Israel) to describe avionic systems. It is one of the most popular graphical formalisms used to describe reactive systems. It consists of an extension of the usual state machines with the following aspects:

2.3.2.2 Basic Features

- *Communication*: Transitions between the states are similar to those of Mealy machines. Such a machine is a finite state machine that generates an output depending on its current state and an input. So, the transitions are of the form `event/action`, where the occurrences of the `event` and `action` are simultaneous.
- *Concurrency*: Parallel behaviors can be easily specified.
- *Hierarchy*: A state may contain further states. Two kinds of states are distinguished: *and-state* and *or-state*. The former contains concurrent substates that evolve simultaneously and communicate through event broadcast. In the latter, the contained substates evolve exclusively. In this case, the choice of a substate when entering the global state can be made from two viewpoints: either it is

[2] http://www.weizmann.ac.il

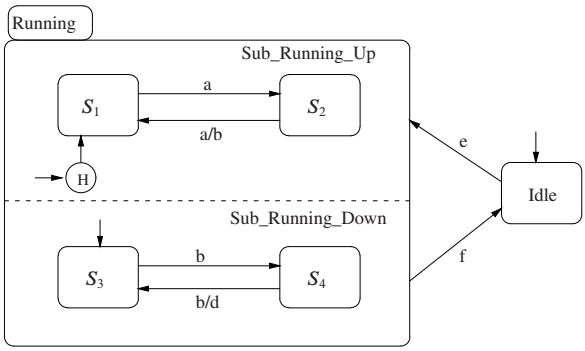


Fig. 2.7 A STATECHARTS-based specification

the substate that is statically indicated by a *default connector* (i.e., a transition without any origin), or it is the substate that was active when the or-state was previously left, indicated by a *history connector*.

The hierarchy also allows one to model preemptive behaviors in the sense that when a state is left, all its substates are systematically left.

Example 2.2 (A hierarchical statechart). Figure 2.7 shows a specification based on STATECHARTS. The automaton represented contains two states in the outer layer: `Idle` and `Running`. The former denotes the initial state. The transitions between these states are fired when events `e` and `f` occur. These events are not associated with any action.

The state `Running` is refined into an and-state, where the concurrent substates are `Sub_Running_Up` and `Sub_Running_Down`. These substates are themselves refined into or-states. For instance, in `Sub_Running_Up`, S_1 and S_2 execute exclusively. We can notice that the transition between S_2 and S_1 is labeled by an event `a` that can trigger an action corresponding, here, to the emission of event `b`. Finally, `H` is a history connector.

2.3.2.3 Semantics of Execution Models

The current *configuration* of a STATECHARTS automaton is defined by the set of its active states. The transition from a configuration to another takes place at each *step*, which represents the unit of reaction. The way occurring actions are taken into account depends on the semantics considered [28].

For instance, in the STATEMATE semantics [18], the actions become effective only at the next step, which induces a temporal delay for the production of outputs. So, the STATEMATE interpretation of STATECHARTS *does not yield a strictly synchronous semantics* such as in ESTEREL. In addition, the specifications obtained are not necessarily deterministic.

STATEMATE defines two models of execution to describe reactions. In the model called **step**, the system receives its inputs from the environment at each step taking part in the current events. In the model called **superstep**, whenever the system receives some inputs at a step, it performs a series of steps, where a step only takes into account the effects of the previous one (i.e., locally generated events), until new events are no longer generated. After this *stable* situation that denotes the end of the superstep, new inputs can be received again from the environment. However, there could be a termination problem in the superstep model if there are infinite loops during the step transitions.

2.3.2.4 ARGOS

The ARGOS language [24] was defined at the Verimag laboratory[3] of the Centre National de la Recherche Scientifique (CNRS), in Grenoble, France. It is a strictly synchronous variant of STATECHARTS: the actions performed within a step become effective in the same step. Moreover, it strongly promotes modularity in specifications. It is compatible with the synchronous viewpoint adopted in ESTEREL. The specifications obtained using this language are always deterministic, in contrast to STATEMATE.

2.3.2.5 SYNCCHARTS

The SYNCCHARTS language [1] was originally defined at the Informatique Signaux Systèmes de Sophia Antipolis[4] (I3S) laboratory, in Sophia-Antipolis, France. It was introduced as a graphical version of the ESTEREL language. Today, a variant of this language, known under the name "Safe State Machines" is being developed Esterel Technologies. SYNCCHARTS is largely inspired by STATECHARTS, but its semantics is fully similar to that of the ESTEREL language.

As an overall observation, a major advantage of graphical languages is that they offer a suitable visual representation, which helps the designer to describe very complex embedded systems.

2.4 Declarative Languages

Synchronous declarative languages such as LUSTRE [14], LUCID SYNCHRONE [10], and SIGNAL [23] belong to the family of dataflow languages whose origin can be historically associated with earlier studies on dataflow models started in the 1970s [11, 20, 29].

[3] http://www.verimag.imag.fr/SYNCRONE

[4] http://www.i3s.unice.fr/~andre/SyncCharts/index.html

2.4.1 *Functional Languages:* LUSTRE *and* LUCID *Synchrone*

The languages introduced below basically use *functions* to describe system behaviors. The resulting descriptions assume a single global abstract clock for the whole system. The status of any component in a program, i.e., presence or absence, is decided according to this global clock.

2.4.1.1 LUSTRE

The LUSTRE language [14] was developed at the Verimag laboratory of CNRS, in Grenoble, France. It is well suited for the description of reactive systems that mainly manipulate dataflows.

2.4.1.2 Language Features

The basic objects are *flows of values* and *nodes*. A flow of values, denoted by a variable x, represents an infinite dataflow $x_1, x_2, ..., x_k, ...$, where x_i is the ith value occurrence of x. A flow is therefore characterized by:

- The set of instants at which it occurs, called its *clock*. This abstract clock can be encoded by a Boolean dataflow in which the occurrences that are equal to *true* denote the presence of the associated flow. On the trace in Fig. 2.8, the Boolean flow b characterizes the clock of the flow y: when b is true (represented by "*t*"), y is present; otherwise y is absent. Two flows are said to be synchronous when they have the same clock.
- The sequence of values carried whenever it occurs. The type of these values can be either simple (e.g., in Fig. 2.8, x and b are, respectively, of integer and Boolean types; *f* and *t*, respectively, denote *false* and *true*) or composite (e.g., array).

A node represents the programming unit and defines a function. So, LUSTRE is a *functional language*. A node is composed of an interface, i.e., input/output and local flows, and a body defined as a set of equations.

Among the operators of the language, we distinguish extensions to flows of usual arithmetic, logical, and conditional operators. For instance, in Fig. 2.8, the usual

v :	0	4	7	2	8	1	5	9	...
b :	*f*	*t*	*f*	*t*	*f*	*f*	*t*	*t*	...
x :	1	2	3	4	5	6	7	8	...
x + v :	1	6	10	6	13	7	12	17	...
pre(x) :	*nil*	1	2	3	4	5	6	7	...
v -> pre(x) :	0	1	2	3	4	5	6	7	...
y = x when b :		2		4			7	8	...
z = current y :	*nil*	2	2	4	4	4	7	8	...

Fig. 2.8 An example of a trace of LUSTRE operators

```
node COUNTER(init : int; reset : bool)
returns (n : int)
let
   n = init -> if reset then init else pre(n) + 1
tel
```

Fig. 2.9 A resettable counter in LUSTRE

Table 2.2 Some useful LUSTRE statements

s_n := f(s_1,..., s_{n-1})	Instantaneous function f on flows s_1,..., s_n
pre(s)	Gives access to the previous value of the flow s
->	Enables to define initial values
s when b	Defines the flow whose sequence of values is that of s when b holds the value *true*
current s	Memorizes the value of s at the basic clock
P ; Q	The nodes P and Q are executed in parallel

pointwise addition is extended to the flows x and v. The pre operator enables the previous values of a flow to be memorized. The initial value is defined using the binary operator ->: the left-hand-side expression (which is a flow) gives the initialization value of the right-hand-side expression (see the example in Fig. 2.9). All these operators make their arguments synchronous. The when operator enables a subsequence of values to be extracted from a flow. Here, only input arguments are required to be synchronous (e.g., in Fig. 2.8, b and x have the same clock). Finally, the current operator enables the values of a flow at a more frequent clock in a node to be memorized. In Fig. 2.8, the flow z is defined with this operator; initially it holds the special undefined value *nil* assumed in LUSTRE. Table 2.2 summarizes the main constructs of the language.

Each node is associated with a global clock, which is the fastest clock, represented by a Boolean dataflow valued *true* at any logical instant. All constant variables of a node are made available at the basic clock. Input flows occur either at the basic clock or at a clock included in the basic clock in terms of sets of instants. This way, whenever a node is invoked, there are at least some flows that are present. Thus, the body of the node can be applied to these flows.

Example 2.3 (A resettable counter in LUSTRE*).* Let us consider a resettable counter that works as follows:

- Whenever the Boolean input reset is *true*, the value of the integer output n is initialized with the value of the integer input init.
- Otherwise, the value of the counter n is incremented by 1 each logical instant.

The LUSTRE program in Fig. 2.9 describes such a counter:

We can see that this node is a monoclocked program. An invocation of such a node is, for instance, val = COUNTER(0, true).

2.4.1.3 Compilation of Programs

LUSTRE programs are deterministic by definition. The compilation of such a program first addresses the causality analysis to schedule the statements and the verification of the consistency of clock constructions [14].

Each expression of a program is associated with a piece of clock information. Such a verification consists in checking that the arguments of specified operators satisfy the expected clock properties. For instance, the arguments of an *instantaneous function* or of a shift register `pre` must have the same clock.

Scheduling constraints are inferred from the instructions of the program considered on the basis of a causality analysis between flow variables. In particular, this enables one to check that every variable is defined only once, that no variable depends on itself, etc.

Finally, when the aforementioned properties have been verified, the compilation of a program produces code in various formats. In the *object code* format, the control structure is defined as extended automata [16]. The *declarative code* format [27] offers interoperability with other synchronous languages: ESTEREL or SIGNAL.

2.4.1.4 Application Domains and Tools

Typical application domains where LUSTRE has been used are automotive vehicles, aerospace and defense, and rail transportation. An academic LUSTRE-based design platform has been proposed by the Verimag laboratory. It includes a compiler, verification (e.g., LESAR [15]), and test tools. The industrial version, called SCADE (Safety Critical Application Development Environment) has been commercialized by Esterel Technologies. It is probably the most successful synchronous technology, with its level A certified code generator in the DO-178B standard.

SCADE: a successful technology

> ***Aerospace domain.*** *"SCADE Suite 4.2 is used by Airbus and its equipment providers for the development of the majority of critical software embedded in the A380 and A 400M aircrafts, as well as the secondary flight controls of the A340-500/600 in operational use since August 2002." – Airbus*

Source: Esterel Technologies. January 2009

2.4.1.5 LUCID Synchrone

The LUCID SYNCHRONE language [10] was developed at Laboratoire de Recherche en Informatique (LRI),[5] in Paris, France. It is a higher-order

[5] http://www.lri.fr/~pouzet/lucid-synchrone

functional language that is built above OCAML (Objective CAML). It shares several characteristics with LUSTRE: it manipulates dataflows, its operators are inspired from the LUSTRE ones, etc.

A specificity of LUCID SYNCHRONE is its abstract clock calculus which is achieved through type calculus as defined in functional languages: the clocks of expressions are inferred by the compiler from a given program. In LUSTRE, a program initially specifies all clocks of expressions.

2.4.2 The Relational Language SIGNAL

The SIGNAL language [4] was developed at the Institut de Recherche en Informatique et Systèmes Aléatoires (IRISA), which has now become the INRIA Rennes–Bretagne Atlantique[6] research center in Rennes, France. In contrast to the other declarative synchronous languages LUSTRE and LUCID SYNCHRONE, it adopts a multiclocked philosophy for modeled embedded systems. The programming style adopted by the SIGNAL language is such that the system behaviors are described using *relations* between, on the one hand, the values of observed events and, on the other hand, the occurrences of these events (i.e., their associated abstract clocks). The so-called *multiclock* or *polychronous semantic model* [23] results from this style. The expressiveness of this semantics allows one to deal with various aspects of the design of large-scale embedded systems.

Notice that while the ESTEREL language has been extended to enable the modeling of multiclocked systems, the SIGNAL language natively proposes concepts that can be used for this purpose.

2.5 Other Languages

Besides the previous languages, there are several others that result from either the extension of an existing general-purpose language with the semantics of synchronous languages, or the specialization of a synchronous language from those already presented. Here, only a few of them are briefly described:

- REACTIVE-C. This language [8] aims at providing a programming style that is close to the C language. The behaviors of a program, also referred to as a "reactive procedure," are expressed in terms of reactions to activations. It was designed in the 1990s at INRIA[7] in Sophia-Antipolis, France.

[6] http://www.inria.fr/inria/organigramme/fiche_ur-ren.en.html

[7] http://www.inria.fr/mimosa/rp

- ECL and JESTER. The ESTEREL-C language (ECL) [22] and the Java-ESTEREL language (JESTER) [2] are other typical examples that, respectively, mix ESTEREL-like constructs with C and Java languages. Both languages are dedicated to embedded system design. ECL was developed at the Cadence Berkeley Labs[8] in Berkeley (CA, USA), whereas JESTER was defined at the PARADES research laboratory[9] in Rome (Italy).
- QUARTZ. The QUARTZ language [26] is a synchronous programming language dedicated to the specification, verification, and implementation of reactive systems. It is based on ESTEREL. It was developed in the Averest[10] project, at the University of Kaiserslautern (Germany).
- *Giotto*. Even though the *Giotto* language [19] adopts a preestimated time vision (see Chap. 1) to program embedded control systems, it also embraces a semantics similar to that of synchronous languages from the functional viewpoint. Thus, it can be put in the same family. *Giotto* was developed at the Center for Electronic Systems Design[11] at the University of California, Berkeley (CA, USA).

2.6 Summary

The synchronous languages advocate a programming discipline that provides developers of real-time embedded systems with a way to suitably deal with the stringent requirements of these systems. They enable one to completely guarantee the correctness of the functional properties of the systems. Owing to their synchrony hypothesis on actual execution durations, nonfunctional properties can be only addressed from a qualitative viewpoint, e.g., the synchronizability of observed events in systems. They have many outstanding features that make them very attractive in comparison with other languages, such as Ada or Real-Time Java:

- Precise mathematical foundations
- Programming styles: imperative, declarative, textual, and graphical
- Formal validation: verification, proof, and test
- Automatic executable code generation

The synchronous languages are associated with adequate integrated development environments that have been adopted in both academia and industry. As the most successful tools, we mention SCADE, RT-BUILDER (http:// www.geensys.com), and ESTEREL STUDIO, which have been extensively used in typical domains such as aerospace, automotive vehicles, and nuclear power plants.

[8] http://www.cadence.com/company/cadence_labs

[9] http://www.parades.rm.cnr.it

[10] http://www.averest.org/overview

[11] http://embedded.eecs.berkeley.edu/giotto

Expected Learning Outcomes:

- The **synchrony hypothesis** is an assumption stating that during the execution of an embedded system, the computations and communications are **instantaneous**.
- The synchronous model enables one to describe:
  - **Monoclocked** systems, i.e., with a global (or master) clock;
  - **Multiclocked** systems, with several independent clocks (without any global clock).
- The temporal aspects are addressed via an abstraction of the physical time by a logical time notion. They are represented by using **abstract clocks** that consist of discrete and ordered pointwise sets of logical instants.
- The synchronous languages have a precise **mathematical semantics that favors formal validation** of described systems. They all support **automatic code generation** towards general-purpose programming languages. The main languages can be distinguished as follows:
  - **Imperative** languages: ESTEREL is textual, whereas STATECHARTS, ARGOS, and SYNCCHARTS are graphical.
  - **Declarative** languages: LUSTRE and LUCID SYNCHRONE are functional, whereas SIGNAL is relational.

References

1. André C (2003) Computing SyncCharts reactions. In: Synchronous Languages, Applications, and Programming (SLAP'03), Electronic Notes in Theoretical Computer Science, Porto, Portugal 88:3–19
2. Antoniotti M, Ferrari A (2000) JESTER, a reactive java extension proposal by ESTEREL hosting. http://www.parades.rm.cnr.it
3. Benveniste A, Caspi P, Edwards SA, Halbwachs N, Le Guernic P, de Simone R (2003) The synchronous languages 12 years later. Proceedings of the IEEE 91(1):64–83
4. Benveniste A, Le Guernic P, Jacquemot C (1991) Synchronous programming with events and relations: the SIGNAL language and its semantics. Science of Computer Programming, Elsevier North-Holland Inc, Amsterdam, The Netherlands 16(2):103–149
5. Berry G (1992) ESTERELon hardware. In: Mechanized reasoning and hardware design, Prentice-Hall Inc, Upper Saddle River, NJ, pp 87–104
6. Berry G (2000) The foundations of ESTEREL. In proof, language and interaction: essays in honour of Robin Milner, MIT Press, Cambridge, pp 425–454
7. Berry G, Sentovich E (2001) Multiclock ESTEREL. In: Proceedings of the 11th IFIP WG 10.5 Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME'01), Springer-Verlag, London, pp 110–125
8. Boussinot F (1991) REACTIVE-C: An extension of C to program reactive systems. Software Practice and Experience 21(4):401–428
9. Boussinot F, de Simone R (1991) The ESTERELlanguage. Another look at real time programming, Proceedings of the IEEE 79:1293–1304

10. Caspi P, Pouzet M (1996) Synchronous Kahn networks. In Proceedings of the first ACM SIGPLAN International Conference on Functional Programming (ICFP'96), Philadelphia, Pennsylvania, pp 226–238
11. Dennis JB (1974) First version of a data flow procedure language. In: Programming Symposium, LNCS 19, Springer, London, UK, pp 362–376
12. Gonthier G (1988) Sémantique et modèles d'exécution des langages réactifs synchrones: application à ESTEREL. PhD thesis, Université d'Orsay, Paris, France (document in French)
13. Halbwachs N (1993) Synchronous programming of reactive systems. Kluwer Academic Publications, Norwell, NA
14. Halbwachs N, Caspi P, Raymond P, Pilaud D (1991) The synchronous dataflow programming language LUSTRE. Proceedings of the IEEE 79(9):1305–1320
15. Halbwachs N, Lagnier F, Ratel C (1992) Programming and verifying real-time systems by means of the synchronous data-flow programming language LUSTRE. In: IEEE Transactions on Software Engineering, Special Issue on the Specification and Analysis of Real-Time Systems 18(9):785–793
16. Halbwachs N, Raymond P, Ratel C (1991) Generating efficient code from data-flow programs. In: Proceedings of the 3rd International Symposium on Programming Language Implementation and Logic Programming, Passau (Germany)
17. Harel D (1987) STATECHARTS : A visual formalism for complex systems. Science of Computer Programming 8(3):231–274
18. Harel D, Naamad A (1996) The STATEMATE semantics of STATECHARTS. ACM Transactions on Software Engineering and Methodology 5(4):293–333
19. Henzinger TA, Horowitz B, Kirsch CM (2001) Embedded control systems development with Giotto. In: Proceedings of The ACM Workshop on Languages, Compilers, and Tools for Embedded Systems (LCTES'2001), The Workshop on Optimization of Middleware and Distributed Systems (OM'2001), Snowbird, Utah, USA
20. Kahn G (1974) The semantics of simple language for parallel programming. In: Proceedings of the IFIP Congress, Information Processing '74, North-Holland Publishing Company, pp 471–475
21. Lamport L (1978) Time, clocks, and the ordering of events in a distributed system. Communications of the ACM, ACM, New York, 21(7):558–565
22. Lavagno L, Sentovich E (1999) ECL: A specification environment for system-level design. In: Proceedings of the 36th Design Automation Conference, New Orleans, Louisana, pp 511–516
23. Le Guernic P, Talpin J-P and Le Lann J-C (2003) Polychrony for system design. Journal for Circuits, Systems and Computers 12(3):261–304
24. Maraninchi F (1991) The ARGOS language: graphical representation of automata and description of reactive systems. In: IEEE Workshop on Visual Languages, Kobe, Japan
25. Potop-Butucaru D, de Simone R (2003) Optimizations for faster execution of ESTEREL programs. In: First ACM and IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE'03), Mont Saint-Michel, France, pp 285–315
26. Schneider K (2009) The synchronous programming language QUARTZ. Internal report, Department of Computer Science, University of Kaiserslautern, Kaiserslautern, Germany
27. Eureka Synchron Project (1995) The common formats of synchronous languages: the declarative code DC. In Deliverable of the Eureka Synchron Project
28. von der Beeck M (1994) A comparison of STATECHARTS Variants. In: Langmaack H, de Roever WP, Vytopil J (eds) Formal Techniques in Real-Time and Fault-Tolerant Systems, Third International Symposium, vol 863 of LNCS, Springer Verlag, Lübeck, Germany, pp 128–148
29. Wadge WW, Ashcroft EA (1985) LUCID, the dataflow programming language. Academic Press Professional Inc, San Diego

Chapter 3
Basics: Signals and Relations

Abstract The very basic concepts of the synchronous language SIGNAL are presented. Section 3.1 first introduces the notion of *signals*. Then, Sect. 3.2 addresses another important aspect of the language, represented by the notion of *abstract clock*. Since SIGNAL is a relational language, it defines a certain number of operators, which enable one to specify *relations* between signals and implicitly between abstract clocks. Section 3.3 describes the primitive operators of SIGNAL. Finally, some exercises are provided in Sect. 3.4. For more details on syntactical aspects of the notions introduced in this chapter, the reader can refer to the grammar given in Appendix B.

3.1 Signals and Their Elementary Features

This section first defines the notion of a signal [1–3], which is very similar to variables in the usual programming languages. Then, its associated characteristics, e.g., type and declaration, are presented.

3.1.1 Definition of Signals

The basic objects manipulated by the SIGNAL language are streams. These streams carry some values whenever they occur and their length can be either finite or infinite. They can be represented as unbounded series of typed values $(s_t)_{t\in\mathbb{N}}$. All the values associated with a stream are of the same nature.

Such streams are referred to as *signals* in SIGNAL. They play the same role as variables in a programming language such as C.

Definition 3.1 (Signal). A signal s is a totally ordered sequence $(s_t)_{t\in I}$ of typed values, where I is $\mathbb{N}$ or an initial segment of $\mathbb{N}$, including the empty segment.

A. Gamatié, *Designing Embedded Systems with the SIGNAL Programming Language: Synchronous, Reactive Specification*, DOI 10.1007/978-1-4419-0941-1_3,
© Springer Science+Business Media, LLC 2010

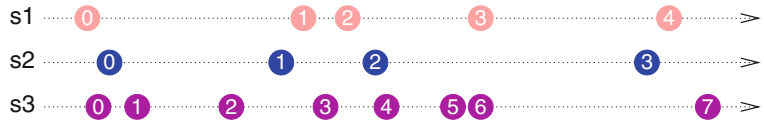


Fig. 3.1 Signals `s1`, `s2`, and `s3` in a trace

Each index t in the sequence corresponding to the signal s denotes a *logical instant*. Hence, the set of logical instants is $\mathbb{N}$.

The trace depicted in Fig. 3.1 shows three signals `s1`, `s2`, and `s3`. The occurrence of each signal is labeled by a rank number in the ordered sequence that defines the logical time scale associated with the signal. The time scales are a priori independent.

3.1.2 Data Types

Similarly to classical languages, SIGNAL defines various *types* of signals. *Numerical* types include `integer`, `real`, and `complex`, whereas *logical* types are mainly represented by the `boolean` type.

3.1.2.1 Type Definition

Types are defined by using the following syntax:

```
type <type_name> = <some_type>.
```

The expression `type second = integer` defines `second` as an `integer` type. It could be used to associate the type `second` with a signal `s` as follows: `second s`.

The most frequently used data types of the SIGNAL language are introduced below.

3.1.2.2 Integer

A signal of `integer` type is a sequence of values whose elements are all of integer type. This is the case of s in the previous example. The integer type is specified using the keyword `integer`.

In the following trace, the signal `s` is of `integer` type:

$$s : 3\ -6\ 0\ 5\ -1\ 8\ 2\ \ldots$$

SIGNAL also proposes *short* and *long representations* for the integer type in addition to the normal representation denoted by `integer`. The short and long integer

representations are, respectively, denoted by the keywords `short` and `long`. If min($type$) and max($type$) express the smallest and the greatest possible values of an integer $type$, respectively, the following must be satisfied:

$$\min(\texttt{long}) \leq \min(\texttt{integer}) \leq \min(\texttt{short}) \leq 0 < \max(\texttt{short}) \leq \max(\texttt{integer}) \leq \max(\texttt{long}).$$

3.1.2.3 Real

A signal of `real` type is also a sequence of values whose elements are all of real type. The main difference from the integer type is that the element notation contains a decimal value (specified with a *dot*). The real type is associated with the same operations as the integer type (see Table 3.1, page 56).

It is specified using the keywords `real` and `dreal` for value representations with, respectively, *simple* and *double* precision. The general syntax for these values is

`<n1>.<n2>e<n3>` (simple precision) and `<n1>.<n2>d<n3>` (double precision),

where `<n1>`, `<n2>`, and `<n3>` are integer constants. The value of these real numbers is defined as $\texttt{n1} + (\texttt{n2} \times 10^{-d}) \times 10^{\texttt{n3}}$, where d is the number of digits in `n2`.

In the following trace, the signal `s` is of `real` type:

s : 3.0e8 −9.5e0 5.8e1 −5e0 1.2e3 −5.8 8.6e0 2.0e7 ...

The signal s in the previous trace is represented with simple precision notation. As it can be observed, some parts of the general notation may be omitted as for $5e0$ and 5.8.

The signal s' in the next trace is represented with double precision notation.

In the following trace, `s` is of `dreal` type:

s' : 3.0d0 9.5d0 5.8d0 5.0d0 1.2d0 5.8d0 8.6d0 ...

3.1.2.4 Complex

SIGNAL also defines a type for *complex* values composed of real and imaginary components. Similarly to signals of real type, those of complex type can be represented either in simple or double precision. The respective type denotations are `complex` and `dcomplex`. The syntax of signals of complex type in the language is

`<real_subpart> @ <imaginary_subpart>`.

Both subparts are represented by values of real type.

The expression `1.0 @ (-1.0)` denotes the value $1 - i$ of complex type.

3.1.2.5 Boolean

The Boolean type is denoted by the keyword `boolean`. Boolean values can be obtained from classical Boolean operators ¬, ∧, and ∨ (with usual priority rules), or comparison operators (`=, /=, <, <=, >, >=`), or constant values *true* and *false*.

In the following trace, the signal `s` is of `boolean` type, where t and f, respectively, denote the values `true` and `false`:

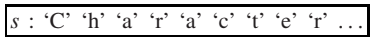
3.1.2.6 Event

This type shares several characteristics with the Boolean type (for more details see Sect. 3.2.3).

3.1.2.7 Character

The character type is denoted by the keyword `char`. The value of a character signal is composed of a usual character (or its ANSI code) represented between single quotation marks.

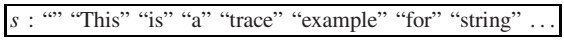
In the following trace, the signal `s` is of `char` type:



3.1.2.8 String

The `string` type includes all finite sequences of admitted characters. The size of a string signal is implementation-dependent. The value of a string signal is represented as a sequence (possibly empty) of characters surrounded by double quotation marks.

In the following trace, the signal `s` is of `string` type:



In addition to the simple types given above, the SIGNAL language also defines composite types such as *enumerate*, *structure*, and *array* types, presented below.

3.1.2.9 Enumeration

The *enumerate* type enables one to represent finite value domains specified with different names. The notation of this type is `enum(v_1, ..., v_n)`, where `v_k` are constant values.

The type `type color = enum(red, green, yellow, blue)` defines an enumerate type for colors.

In the following trace, the signal `s` is of `color` type:

s : #yellow #green #yellow #blue #red #green #blue ...

3.1.2.10 Structure

The *structure* type is a tuple in which elements of different types can be grouped together. It is noted as `struct(type_1 id_1; ...; type_n id_n;)`.

If `s` is a signal of type structure as follows

```
type s = struct(type_1 id_1; ...; type_n id_n;),
```

all its fields `id_1, ..., id_n` are available whenever `s` is present.

Given a signal *s* of `struct` type, all data values associated with the fields `id_k` of *s* are available whenever *s* is assumed to occur. Each field `id_k` is accessed using the usual *dot* notation as follows: `s.id_k`.

The notation `struct(integer i1, i2; boolean b)` defines a structured data type composed of two integer fields `i1`, `i2` and a single Boolean field `b`. If *s* is of this data type, its Boolean component is accessed by writing `s.b`.

3.1.2.11 Bundle

The SIGNAL language proposes a generalization of the structure data type, referred to as `bundle`. The main difference from the `struct` type is that some fields of a signal *s* of `bundle` type may be *unavailable* (i.e., absent) while the signal *s* is assumed to occur.

Explicit constraints can be specified on sets of instants associated with the fields of *s*. Typically, one may specify that some critical fields are available only when specific environment properties are satisfied.

The `bundle` type has the same syntax as the `struct` one where the keyword `struct` is replaced by `bundle`.

3.1.2.12 Arrays

The array type allows one to group together synchronous elements (simultaneously present) of the same type. Its notation is as follows:

```
[dim_1, ..., dim_n] elt_type,
```

meaning an array of dimension `n` where elements are of `elt_type` type. It defines the function

$$([0, \texttt{dim_1} - 1] \times \cdots \times [0, \texttt{dim_n} - 1]) \rightarrow \texttt{elt_type}.$$

The index domain of an array implicitly starts from 0.

For example, the expression `type B = [N] boolean` denotes a Boolean vector `B` of size `N`. An integer matrix with L lines and C columns can be defined as `type Int_Matrix = [L, C] integer`.

Let us consider a one-dimension array `A`. Its elements are extracted using the notation `A[k]`, where `k` is an index signal, synchronous with `A`, taking its values between 0 and the size of `A` minus 1. Consider again the Boolean vector `B` of size `N`. Its first and last elements are, respectively, defined as `B[0]` and `B[N-1]`.

An array can be defined via a static enumeration of its elements. This is expressed using the following notation:

```
A := [elt_0, ..., elt_N-1],
```

where `elt_i` are of the same type as the elements of `A`. For example, if `B` is of size `2`, one can write `B := [false, true]`. Sometimes, one may need to define an array only partially. This is specified as follows:

```
A := (k):elt.
```

Here, the element of `A` at index `k` takes the value `elt`. For instance, if `A` initially holds the value `[true, false]`, then the statement `A := (0):false` yields `A := [false, false]`.

The extraction of a subset of elements from an array is expressed as

```
A1 := A[ind_0, ..., ind_l],
```

where `ind_k` are integer values or arrays of integer values.

For example, if `C` is equivalent to `[4,5,8,9]`, the expressions `C[0]`, `C[1]`, and `C[3]`, respectively, yield `4`, `5`, and `9`. In this case, the indexes are integer values.

3.1.2.13 Comparison and Conversion of Types

Type comparison and conversion are very important when signals of different types are used in the same expression. The expression can be evaluated (i.e., is valid) only if the combined types are compatible in some way.

The previous data types are organized into abstract *domains*, which are used to reason about type compatibility. Let us consider the following domains:

- *Logical type domain*: `boolean` and `event` types
- *Integer type domain*: `short`, `integer`, and `long` types
- *Real type domain*: `real` and `dreal` types
- *Complex type domain*: `complex` and `dcomplex` types
- For each of the remaining types (`char`, `string`, etc.), we consider that the type concerned forms a proper domain

The types belonging to the same domain are *comparable* by using a partial order, defined as follows:

- Logical type domain: `boolean` is greater than `event`

- Integer type domain: `long` is greater than `short` and `integer`, and `integer` is greater than `short`
- Real type domain: `dreal` is greater than `real`
- Complex type domain: `dcomplex` is greater than `complex`

This notion of type comparison is also extended to the arrays and tuple types, i.e., `struct` and `bundle` [2].

Type comparison serves in type conversion. Let us consider a binary SIGNAL expression in which one argument holds a type value `<type1>` and the other argument holds a type value `<type2>`.

Implicit Type Conversion

If both types belong to the same type domain such that `<type2>` is greater than `<type1>`, then the argument with type `<type1>` can be *implicitly* converted to type `<type2>`. In that case, the value of the expression associated with type `<type1>` is unchanged in type `<type2>`. Typically, in a Boolean binary expression containing an `event` signal and a `boolean` signal, the signal of `event` type is implicitly converted into `boolean`; then, the result is of `boolean` type. Similarly, in a binary arithmetic expression, a signal of `integer` type can be safely converted to `long` if the other argument is of `long` type.

Explicit Type Conversion

In addition to implicit type conversion, the SIGNAL language also enables one to *explicitly* define conversions between different types from either the same domain or not from the same domain. The syntax for the explicit type conversion of an expression `<exp>` to a type `<my_type>`, known as *type cast* in the usual programming languages, is as follows:

```
<my_type>(<exp>).
```

Thus, within the same domain, a greater type can be explicitly converted to a smaller type. For instance, in an arithmetic binary expression containing an `integer` signal `s` and a `short` signal, `s` can be converted to `short` so as to have an operator on two signals of `short` type. This is noted `short(s)`. The value v of the converted `integer` signal `s` is unchanged if v is between min(`short`) and max(`short`).

When an expression contains signals with data types from different domains, a similar schema holds. For instance, the conversion of the `boolean` type to `long` type projects the constant values *false* and *true* on 0 and 1, respectively; so, `long(false)` yields the value 0. Conversely, the `long` type can be projected on the `boolean` type by transforming 0 and 1 into *false* and *true*, respectively. The conversion of the `char` type to `long` type transforms the character into the numerical value of its corresponding code.

Remark 3.1. Type conversion between types from different domains is not always possible directly in SIGNAL, e.g., there is no direct conversion between `char` and `boolean`. However, the conversion may be possible via intermediate type domain conversions, e.g., conversion between `char` and `boolean` via the `long` type of the integer type domain. Furthermore, a type conversion between different domains may sometimes be only unidirectional, e.g., from `string` to `event`. For a more complete description of type comparison and conversion, the reader is referred to the SIGNAL reference manual [2].

3.1.3 Identifier of a Signal

Signals are identified by lexical units[1] formed of characters among the set composed of letter and numeral characters and the special character "_" (see the grammar in Appendix B). Examples of signal identifiers are

```
s, S, s_75, mY_IdenTtifier45.
```

Such a lexical unit, referred to as the *name* of a signal (or a signal *variable*), must always start with a letter character. For instance, the identifier `S` is allowed, whereas `8S` is not allowed. The syntax of an identifier is case-sensitive. Typically, `s` and `S` are distinct names. In addition, an identifier cannot be a reserved word of the language.

3.1.4 Declaration of Signals

In SIGNAL, a signal must be declared before being used within any statement. The declaration is given in the following form:

```
<type> <id_1>,...,<id_n>,
```

where `<type>` denotes the type (see Sect. 3.1.2) of the n signals and `<id_k>` is the identifier corresponding to the kth declared signal.

Possible initial values of signals can be specified during the declaration as follows:

```
<type> <id_1>,...,<id_n> init <v_1>,...,<v_n>,
```

where `<v_k>` is the initialization value of the signal identified by `<id_k>`.

[1] More generally, such lexical units are also used to identify other concepts of the language, e.g., directives, parameters, constants, and types. The SIGNAL grammar given on page 217 details this aspect.

3.1.5 Constant Signals

A constant signal c is a sequence that carries the same value whenever the signal is present. This value is the one associated with c.

An example of integer constant signal holding 4 is represented by the following trace:

c : 4 4 4 4 4 4 4 4 4 ...

The declaration of a constant is done using the keyword `constant` as in the following definitions:

```
constant real PI = 3.14,
constant [2,2] integer IDTY = [[1,0],[0,1]].
```

In the language notation, constant values can be represented either directly by their value, e.g., `4`, `3.14`, `true`, or by an identifier, e.g., `x`, `maxvalue`.

An important characteristic of constant signals is that no abstract clock is a priori associated with them. Their presence depends on their usage context. In particular, they are present whenever they are required in the context.

In the following trace, the signal s_2 is the result of the sum of s_1 and 4:

s_1 : 1 3 5 2 0 −4 1 4 −1 ...
constant 4 : 4 4 4 4 4 4 4 4 4 ...
s_2 : 5 7 9 6 4 0 5 8 3 ...

The presence of the constant value 4 depends on its context of usage. In SIGNAL, addition is an operation that imposes the presence of all its operands. So, here all signals are simultaneously present.

3.2 Abstract Clock of a Signal

Owing to the multiclock vision adopted by the SIGNAL language, the ability to *explicitly* deal with the occurrence of signals is clearly an important requirement for specifying system behaviors. The notion of *abstract clock* aims at providing the programmer with an adequate way to express both the temporal property of a signal and the temporal relations, also referred to as *synchronizations*, between several signals. Thus, abstract clocks are fundamentally the main means to express control properties between signals.

3.2.1 The Notion of Polychrony

In Sect. 2.2.2, we saw the difference between the *monoclocked* system model and the *multiclocked* system model. In contrast to the other declarative synchronous

languages (e.g., LUSTRE), SIGNAL assumes a vision of systems based on the latter model: this is referred to as *polychrony*[2] in the jargon of SIGNAL.

3.2.1.1 Need to Specify Absence

Thanks to its polychronous model, a SIGNAL specification does not consider a priori the existence of a reference abstract clock that enables one to decide whether or not a signal is present at any instant. As a result, there is the need to explicitly deal with the *absence* of signals so as to be able to specify temporal relations between signals.

With respect to the physical process (mentioned in Chap. 1), the absence of a signal at a logical instant t means that no event associated with this signal is observed during the interval of physical time corresponding to the instant t. Note that, a priori, when a signal is absent, its value at the next occurrence is not necessarily the same as the one at the very last occurrence (i.e., its value may change). Figure 3.2 shows the same trace as in Fig. 3.1, where the absence of every signal is illustrated with respect to each other. In this partial view of the trace, one can notice that the set of instants at which signal `s2` occurs is strictly included in that of instants at which signal `s1` occurs. On the other hand, even though `s3` seems to occur more often than the other signals, its corresponding set of instants includes none of those corresponding to `s1` and `s2`. For instance, the event tagged "4" in `s1` is not contained in `s3`.

Greek mythology

Chronos. *Painting of a contemporary representation of Chronos, "Father Time," from the Cimitero monumentale di Staglieno in Genoa, Italy.*

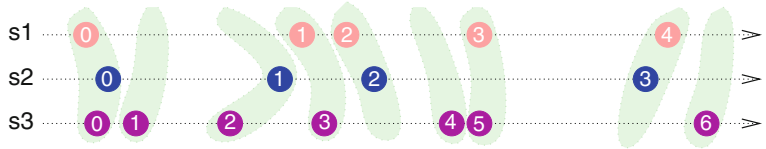


Fig. 3.2 Absence/presence of signals in a trace

[2] From the Greek *poly chronos*, meaning "more than one clock" or "many clocks."

3.2.1.2 Role of Abstract Clocks in Design

Abstract clocks play a very important role in system design, especially when the systems are multiclocked. They offer an adequate way to describe the following aspects:

- *Loosely coupled systems* where components can behave according to their own activation physical clock. In particular, the modeling of the so-called *globally asynchronous, locally synchronous* (GALS) systems is easier (see also Chap. 11). Figure 3.3 illustrates how the activation abstract clocks of the different execution nodes in the GALS system can be related. On the one hand, each node has its own perspective of time. On the other hand, the interaction (e.g., communications) between nodes can be modeled by specifying relations between the abstract clocks of the nodes concerned. In the figure, arrows are used to describe at which instants the interaction takes place with respect to the local perspectives of time. As a consequence, time becomes *partially ordered* from a global point of view in the system.
- During the design of complex systems in an *incremental* way, the addition of a new component C to an already defined subpart S of a system does not necessarily require one to adjust an existing reference clock (i.e., resynchronization) of the system. For instance, let us consider again the system illustrated in Fig. 3.3. If a new node, say, `node 4`, has to be added such that this node only interacts with `node 2`, the only abstract clock relations the designer has to focus on are those implying `node 2` and `node 4`. The other parts of the system would not require any additional modification.

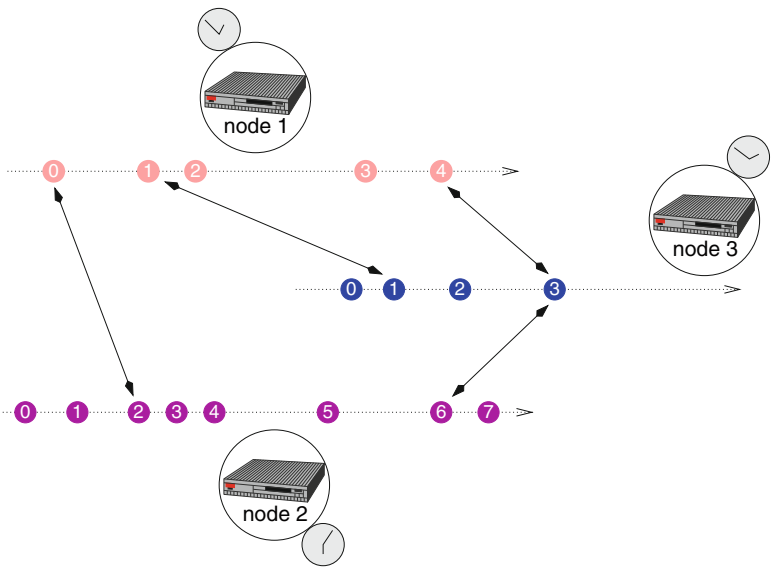


Fig. 3.3 Node interaction within a globally asynchronous, locally synchronous system

3.2.2 Definition

Let us consider the following trace, where the center dot represents the absence of signal value at logical instants:

t :	t_0	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}	t_{11}	t_{12}	...
s_1 :	·	t	f	t	·	t	t	f	·	f	t	·	t	...
s_2 :	·	3	·	6	0	5	·	1	·	8	·	·	2	...
s_3 :	·	3.0	9.5	·	·	5.8	5.0	1.2	5.8	8.6	·	·	2.0	...

The status of signals s_1, s_2, and s_3 (i.e., their presence or absence) can be checked with respect to signal t, which is more frequent than the others. This signal may play the role of a logical time reference, denoted as $(t_k)_{k\in\mathbb{N}}$. In particular, we notice that s_1, s_2, and s_3 are absent at instant t_{11}. This was not observable on the trace given in Fig. 3.2 because there was no reference signal. So the status of a signal is decided relative to another signal.

Definition 3.2 (Abstract clocks). Given a logical time reference, the set of logical instants at which a signal s is *present* is called its abstract clock.

Example 3.1. In the trace given above, the abstract clocks associated with s_1 and s_2, are respectively, defined by the following sets of logical instants:

$$\{t_1, t_2, t_3, t_5, t_6, t_7, t_9, t_{10}, t_{12}\} \text{ and } \{t_1, t_3, t_4, t_5, t_7, t_9, t_{12}\}.$$

Given a signal $s = (s_t)_{t\in\mathbb{N}}$, if s does not occur at a logical instant $t \in \mathbb{N}$, the value s_t of s at this instant is denoted by $\perp$ in the semantic notation.

3.2.3 The Event Type

The SIGNAL language defines a special type, called `event`, that allows one to characterize abstract clocks. It is very similar to the `boolean` type. However, the major difference is that a signal of `event` type always holds the value *true* whenever it is present. So, an abstract clock can be represented as a signal of `event` type.

Example 3.2. The trace given below shows a `boolean` signal s_1 and an `event` signal s_2, where s_2 denotes the abstract clock of s_1.

s_1 :	t	f	t	·	t	t	f	f	·	t	·	t	...
s_2 :	t	t	t	·	t	t	t	t	·	t	·	t	...

Remark 3.2. From now on, for the sake of simplicity, the word **clock** when used alone will denote **abstract clock**. However, whenever necessary, it will be explicitly defined to avoid any confusion between physical and abstract clocks.

3.3 Relation Between Signals

Relations are fundamental notions in SIGNAL programming. They offer the basic means to describe behaviors by allowing one to express properties on sets of signals.

Definition 3.3 (Relation). A relation $R(s_1, \ldots, s_k)$ between different signals $s_1, \ldots, s_k$ consists of a set of constraints that has to be satisfied by

- The values of signals $s_1, \ldots, s_k$: *functional constraint*
- The abstract clocks of signals $s_1, \ldots, s_k$: *temporal constraint*

The next sections introduce the way relations are expressed between signals by using the primitive constructs of SIGNAL.

3.3.1 Equational Specification of Relations

The specification of a relation between different signals is described using the following basic syntax:

```
<id> := <exp>,
```

where `<id>` identifies a signal and `<exp>` denotes an expression over signals.

The operator `:=` is usually interpreted as *assignment* even though it semantically consists of an equality. Indeed, the expression `<id> := <exp>` is an *equation*. This equality states that `<id>` and `<exp>` have *the same clock and the same value*. The next sections present the clock definitions of different kinds of expressions obtained by using the *primitive constructs* or relations of the language.

SIGNAL relies on six primitive constructs. In this chapter, we only introduce four of them through their associated operators. These four operators share the fact that they apply to signals. This is not the case for the remaining two operators, which are presented in Chap. 4.

The primitive operators that apply to signals can be classified into two families:

1. **Monoclock** operators, for which all signals involved have the same clock: *instantaneous relations/functions and delay*
2. **Multiclock** operators, for which the signals involved may have different clocks: *undersampling* and *deterministic merging*

3.3.2 Primitive Monoclock Relations

There are two kinds of monoclock operators: *static* operators, for which equations refer to the same time index of signals, and *dynamic* operators, for which equations refer to different time indexes of signals.

3.3.2.1 Instantaneous Relations/Functions

These operators canonically extend the usual pointwise relations/functions (e.g., addition and disjunction) to sequences.

Syntax 1 s_n := $R(s_1, \ldots, s_{n-1})$ *where each* s_k *denotes a signal and* R *is a n-ary relation/function that extends a pointwise relation/function to signals.*

Definition 3.4 (Instantaneous relations). $\forall \tau \in \mathbb{N}$

$$s_{n\tau} = \begin{cases} \bot & \text{if } s_{1\tau} = \cdots = s_{n-1\tau} = \bot \\ R(s_{1\tau}, \ldots, s_{n-1\tau}) & \text{if } s_{1\tau} \neq \bot \wedge \ldots \wedge s_{n-1\tau} \neq \bot. \end{cases}$$

The instantaneous relations/functions require all signals s_k involved to have the same clock: the signals are either all present or all absent. Let us recall that absence of signals at a given instant is denoted by the special symbol $\bot$.

As we can observe in their semantics, they are static operators since the properties induced on signals refer to the same value of time index τ.

Example 3.3. Let us consider the following equation: s_3 := s_1 * s_2. A possible corresponding trace is as follows:

t :	t_0	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	...
s_1 :	·	·	5	·	·	4	2	0	·	···
s_2 :	·	·	3	·	·	1	9	8	·	···
s_3 :	·	·	15	·	·	4	18	0	·	···

Tables 3.1, 3.2, and 3.3 provide a summary of the most used SIGNAL operators that enable one to describe monoclock or synchronous expressions on signals.

Beyond these operators, the SIGNAL language also proposes an operator that enables one to express a *synchronous condition* as follows:

```
if b-exp then exp1 else exp2,
```

Table 3.1 Basic arithmetic operators

Comments	Operator	Type
Addition (binary)	+	*numerical-type* × *numerical-type* → *numerical-type*
Addition (unary)	+	*numerical-type* → *numerical-type*
Subtraction (binary)	-	*numerical-type* × *numerical-type* → *numerical-type*
Subtraction (unary)	-	*numerical-type* → *numerical-type*
Multiplication	*	*numerical-type* × *numerical-type* → *numerical-type*
Power	**	*numerical-type* × *integer-type* → *numerical-type*
Division	/	*numerical-type* × *numerical-type* → *numerical-type*
Modulo	modulo	*integer-type* × *integer-type* → *integer-type*

The word *numerical-type* denotes one of the following types: long, integer, short, dreal, real, dcomplex, and complex. The word *integer-type* denotes long, integer, or short. For each binary operator, the type of the result is the greater between the types of the arguments, which must have the same type domain. For each unary operator, the type of the result is the same as for the argument.

Table 3.2 Basic comparison operators

Comments	Operator	Type
Equality	=	*scal_enum-type* × *scal_enum-type* → boolean
Difference	/=	*scal_enum-type* × *scal_enum-type* → boolean
"Greater than"	>, >=	*scal_enum-type*^{c} × *scal_enum-type*^{c} → boolean
"Less than"	<, <=	*scal_enum-type*^{c} × *scal_enum-type*^{c} → boolean

The word *scal_enum-type* denotes one of the following types: boolean, event, long, integer, short, dreal, real, dcomplex, complex, char, string, and enum. The word *scal_enum-type*^{c} means one of the *scal_enum-type* types except dcomplex or complex. The arguments of these operators must have the same type domain.

Table 3.3 Basic Boolean operators

Comments	Operator	Type
Negation	not	*logical-type* → boolean
Conjunction	and	*logical-type* × *logical-type* → boolean
Disjunction	or	*logical-type* × *logical-type* → boolean
Disjunction (exclusive)	xor	*logical-type* × *logical-type* → boolean

The type denoted by *logical-type* denotes either boolean or event

where b-exp is a Boolean expression, and the expressions $\texttt{exp}_1$ and $\texttt{exp}_2$ are of the same type. All signals involved in the synchronous condition hold the same clock. The result of the synchronous condition expression is equal to that of $\texttt{exp}_1$ if b-exp is evaluated to be *true*. It is equal to $\texttt{exp}_2$ if b-exp is *false*.

For instance, the equation $\texttt{s}_1$:= if b then $\texttt{s}_2$ else $\texttt{s}_3$ means that $\texttt{s}_1$ has the same value as $\texttt{s}_2$ if b holds the value *true*; otherwise, $\texttt{s}_1$ has the same value as $\texttt{s}_3$. In addition, $\texttt{s}_1$, b, $\texttt{s}_2$, and $\texttt{s}_3$ are synchronous.

3.3.2.2 Delay (or Shift Register)

This operator enables one to shift a sequence so as to move to lower temporal indexes, denoted by t in a signal $(s_t)_{t\in\mathbb{N}}$. In that way, one can gain access to values carried "in the past" by a signal. Depending on which past values one would like to get, the expression is almost the same.

Here, we first introduce the simplest case: given logical instant τ, how to access the *latest* value of a signal, excepting its value at τ.

Syntax 2 $\texttt{s}_2$:= $\texttt{s}_1$ $ 1 init c, *where* $\texttt{s}_1$ *and* $\texttt{s}_2$ *are signals and* c *is an initializing constant (of the same type as* $\texttt{s}_2$*), is defined as follows. Given a logical instant* τ, $\texttt{s}_2$ *takes the most recent value of* $\texttt{s}_1$ *excepting the one at* τ. *Initially,* $\texttt{s}_2$ *takes the value* c.

As a shortcut, the above syntax is also written as $\texttt{s}_2$:= $\texttt{s}_1$ $ init c.

Definition 3.5 (Delay).

$$
\begin{aligned}
&- (\forall \tau \in \mathbb{N})\ s_{1\tau} = \perp \ \Leftrightarrow\ s_{2\tau} = \perp \\
&- (\exists \tau_i \in \mathbb{N})\ s_{1\tau_i} \neq \perp \Rightarrow s_{2\tau_0} = c,\ \ (\forall \tau_i > 0)\ s_{2\tau_{i+1}} = s_{1\tau_i}, \\
&\qquad\qquad \text{where } \tau_0 = \min\{k \mid s_{1k} \neq \perp\} \text{ and} \\
&\qquad\qquad \tau_{i+1} = \min\{k \mid k > \tau_i \wedge s_{1k} \neq \perp\},
\end{aligned}
$$

where $\min(S)$ denotes the minimum of a set S.

Similarly to instantaneous functions/relations, the delay operator also ensures that signals `s1` and `s2` have the same clock. However, it is a dynamic operator since the properties induced on the signals refer to different values of time indexes τ_i and τ_{i+1}.

Example 3.4. Let us consider the equation `s2 := s1 $ 1 init 3.14`. A possible corresponding trace is as follows:

t :	t_0	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}	...
s_1 :	·	·	1.7	2.5	·	·	6.5	·	2.4	1.3	5.7	...
s_2 :	·	·	3.14	1.7	·	·	2.5	·	6.5	2.4	1.3	...

The delay operator also exists in a *generalized form*. This version enables access to the value carried by a signal k logical instants before. It is expressed through the equation `s2 := s1 $ k init tab`, where `tab` is a bounded vector containing constants for initializing and `k` is a signal of integer type carrying a value at least equal to 0 and less than the length of `tab`.

Example 3.5. For instance, let us consider the statement `s2 := s1 $ 3 init [5,7,9]`. A possible trace is given below:

t :	t_0	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}	...
s_1 :	·	·	1	2	·	·	6	·	4	3	8	...
s_2 :	·	·	5	7	·	·	9	·	1	2	6	...

When a value is not specified for initializing in a delay operation, the SIGNAL language assumes a default value, which depends on the type of the signals defined. For instance, if `s1` and `s2` are of integer type in `s2 := s1 $ 1`, then the default value is `0`.

Remark 3.3. When the delay operator is applied to a signal of `event` type, such as in `s2 := s1 $ 1 init true`, where `s1` is of `event` type, the two signals `s1` and `s2` have the same clock and `s2` holds the previous value of `s1`, i.e., *true*. Thus, it is important to notice that applying the delay operator of SIGNAL on a signal of `event` type does not mean at all delaying the occurrence of this signal for some amount of time, as could be defined in languages such as Ada and ESTEREL.

3.3.3 Primitive Multiclock Relations

3.3.3.1 Undersampling

This operator enables one to extract a subpart of a series, under a given condition.

Syntax 3 `s2 := s1 when b`, *where* `s1` *and* `s2` *are two signals of the same type and* `b` *is a Boolean signal.*

Definition 3.6 (Undersampling). $\forall \tau \in \mathbb{N}$

$$s_{2\tau} = \begin{cases} s_{1\tau} \text{ if } b_\tau = true \\ \perp \quad \text{otherwise.} \end{cases}$$

The clock of `s2` is defined as the intersection of the clock of `s1` and the set of instants where `b` carries the value *true*, noted $[b]$.

Example 3.6. The following trace shows a result of `s2 := s1 when b`, where `s1` and `s2` are of integer type:

t :	t_0	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	...
s_1 :	·	·	5	·	4	8	7	3	−1	...
b :	·	·	t	f	·	f	t	·	t	...
s_2 :	·	·	5	·	·	·	7	·	−1	...

3.3.3.2 Deterministic Merging

This operator allows one to define the functional "interlacing" of two series.

Syntax 4 `s3 := s1 default s2`, *where* `s1`, `s2`, *and* `s3` *are signals of the same type.*

Definition 3.7 (Deterministic merging). $\forall \tau \in \mathbb{N}$

$$s_{3\tau} = \begin{cases} s_{1\tau} \text{ if } s_{1\tau} \neq \perp \\ s_{2\tau} \text{ otherwise.} \end{cases}$$

The clock of `s3` is defined as the union of the clock of `s1` and `s2`.

Example 3.7. A possible trace associated with the statement `s3 := s1 default s2` is given below (all signals are of integer type).

t :	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	...
s_1 :	·	5	·	4	8	·	·	3	·	...
s_2 :	·	51	17	·	32	·	20	13	·	...
s_3 :	·	5	17	4	8	·	20	3	·	...

3.4 Exercises

3.1. Indicate whether or not the following affirmations are true and justify your answer:

1. A signal is necessarily present whenever it holds a value different from $\perp$.
2. A constant signal is always present.
3. When a signal becomes absent, it implicitly keeps its previously carried value.
4. A signal of `event` type and a signal of `boolean` type are exactly the same.
5. The abstract clock of a signal defines the set of instants at which the signal occurs.
6. SIGNAL assumes a reference clock that enables one to always decide the presence/absence of any defined signal.
7. In the expression s_n `:=` `R(`s_1`,...,`s_{n-1}`)`, where `R` is an instantaneous relation, if the signal s_1 is absent while all other arguments of `R` are present, s_n is calculated by considering some default value depending on the type of s_1.
8. In the expression s_2 `:=` s_1 `$ 1 init c`, the signal s_2 may occur with the latest value of s_1 while s_1 is absent.
9. In the expression s_3 `:=` s_1 `default` s_2, the signals s_1 and s_2 must have exclusive clocks.
10. In the expression s_3 `:=` s_1 `or` s_2,
    a. s_3 is true when s_1 is true and s_2 is absent.
    b. s_3 is true when s_1 is true and s_2 is false.
    c. s_3 is false when s_1 is absent and s_2 is false.
    d. s_3 is false when s_1 is absent and s_2 is absent.

3.2. Indicate the possible type for each signal in the following expressions and simplify each expression:

1. (E_1) : `(s when s) default s`
2. (E_2) : `(s1 when b) default (s2 when b)`
3. (E_3) : `if s1 > s2 then true else false`

3.3. In the following expressions, where `s1` and `s2` are two signals independent of each other, `f` and `g` are instantaneous functions and `cond` is a Boolean function.

- (E_1) : `(f(s1) when cond(s2)) default g(s1)`
- (E_2) : `(f(s2) when cond(s1)) default g(s1)`

Determine the clock of expressions E_1 and E_2.

3.4. Let us consider the following expression:

`(A when B) default C.`

Its evaluation can lead to different results. For instance, if signals `A` and `B` are absent but signal `C` is not absent, the expression takes the value of `C`. Given that `B` is a Boolean signal and `A` and `C` are of any type, enumerate all possible cases for the evaluation of that expression.

3.5. Let us consider the following expression:

```
b := true when c default false.
```

- When is this expression correct regarding clock constraints?
- When can it be simplified to `b := c`?

3.6. Let `s1` and `s2` be some Boolean signals. They are assumed to have independent clocks. Define a signal of integer type whose value is determined as follows:

- 1 if `s1` is *true*
- 2 if `s2` is *true*
- 3 if `s1` and `s2` are both *true*
- Absent otherwise

3.7. Indicate the possible type for the signal `s` in the following expressions and give the clock of each expression:

1. `((not s) when (not s)) default s`
2. `(s when s) default (not s)`

Expected Learning Outcomes:

- A **signal** is an infinite totally ordered sequence of typed values. In some sense, it plays the same role as variables in the usual programming languages and has similar characteristics: types, values, etc.
- A **clock** is a specific mechanism of the SIGNAL language, which enables one to refer to the presence or absence of events in a specified system. It is mainly used to describe synchronizations between signals, i.e., the control part of a SIGNAL program.
- **Relations** consist of operators over signals as well as clocks. These operators are used to express some behavioral properties on sets of signals. SIGNAL distinguishes four primitive relational operators:
  - **Monoclock** relations: instantaneous relations/functions, delay
  - **Multiclock** relations: undersampling, deterministic merging

References

1. Benveniste A, Le Guernic P, Jacquemot C (1991) Synchronous programming with events and relations: the SIGNAL language and its semantics. Science of Computer Programming, Elsevier North-Holland, Inc, Amsterdam The Netherlands 16(2):103–149
2. Besnard L, Gautier T, Le Guernic P (2008) SIGNAL v4 – INRIA Version: Reference Manual. Available at: http://www.irisa.fr/espresso/Polychrony
3. Le Guernic P, Gautier T, Le Borgne M, Le Maire C (1991) Programming real-time applications with SIGNAL. Proceedings of the IEEE 79(9):1321–1336

Chapter 4
Programming Units: Processes

Abstract This chapter introduces the notion of *process*, which constitutes together with signals the basic objects of the SIGNAL language. First, Sect. 4.1 defines this notion. Then, in a similar way as for signals, the primitive operators of the language that are applied to processes are presented in Sect. 4.2. Finally, Sect. 4.3 addresses the notational aspects with an illustrative example.

4.1 Elementary Notions

This section first defines the SIGNAL processes [1, 2, 3]. Then, it introduces a particularly useful process, termed "function" in the SIGNAL terminology. Finally, the notion of program is presented.

4.1.1 Definition of Processes

The primitive constructs presented in the previous chapter induce constraints, on the one hand, between the values of signals and, on the other hand, on the clocks of the signals.

Each equation that expresses some properties of signals via a primitive construct defines an *elementary process*.

The following equation for signals `s1` and `s2`,

```
s2 := N * s1,
```

where `N` is a constant parameter, defines an elementary process implying that (1) at each logical instant, the value of `s2` is a multiple of that of `s1` by factor `N` and (2) the clock of `s2` is the same as the clock of `s1`.

A. Gamatié, *Designing Embedded Systems with the SIGNAL Programming Language: Synchronous, Reactive Specification*, DOI 10.1007/978-1-4419-0941-1_4,
© Springer Science+Business Media, LLC 2010

More generally, a process consists of a set of equations. Such a set is obtained from elementary processes by applying a few specific operators, presented in Sect. 4.2.

Definition 4.1 (Process). A process is a set of equations for signals specifying relations between values, on the one hand, and clocks, on the other hand, of the signals involved.

4.1.2 A Specific Process: Function

We now introduce a particular type of process, called *function*. Such a process implicitly imposes the following constraints on its interface signals:

- All input and output signals have the same clock.
- At each logical instant, there is a data dependency between input and output signals (i.e., all input signals are assigned values before all output signals are assigned values). Input signals are said to *precede* the output signals.

In addition, a function does not have any state variable representing some memorization.

Definition 4.2 (Function). A function is a stateless process in which (1) input and output signals are synchronous and (2) all input signals precede all output signals.

4.1.3 Programs

In usual programming languages, e.g., C and Java, a program consists of a sequence of instructions defining a given behavior of the computer, which executes this program. It generally adopts a syntax that differs from that of the programming units of these languages, typically functions or procedures.

In SIGNAL, the notion of program is slightly different from the above vision. A program is a process and shares the same syntax (see Sect. 4.3).

Definition 4.3 (Program). A program is a process.

4.2 Primitive Operations on Processes

There are two basic operators: the first one applies to a pair of processes, whereas the other one applies to a process and a set of signals.

4.2.1 Composition

The composition of two processes is the union of the equation systems defined by both processes.

Syntax 5 `P | Q`*, where* `P` *and* `Q` *are processes. Such a composition syntactically consists in merging the equations corresponding to both processes* `P` *and* `Q`*.*

Definition 4.4 (Composition). The process `P | Q` specifies the relations defined by both `P` and `Q` between values, on the one hand, and clocks, on the other hand, of signals.

In other words, the behavior of `P | Q` can be seen as the conjunction of the mutual behaviors of `P` and `Q`. In `P | Q`, the common signal identifiers refer to common signals. Thus, processes `P` and `Q` communicate via their common signals: input signals of `P` can be output signals of `Q` and vice versa. No signal can be defined at the same time in both `P` and `Q`: SIGNAL respects the usual *single definition principle* of dataflow languages, i.e., a signal can only have a unique definition.

The following composite statement defines the Boolean signal `cond` on the basis of the value of another signal, `s2` (of integer type), which is also calculated in the same statement:

```
s2 := N * s1 | cond := s2 > 32.
```

Among the implicit properties of the parallel composition operator, the following ones can be mentioned [4]:

- *Commutativity* : `P | Q = Q | P`
- *Associativity* : `P | (Q | R) = (P | Q) | R`

4.2.2 Local Declaration

The *local definition* operator enables one to declare local signals in a process.

Syntax 6 `P where type_1 s_1; ...; type_n s_n; end`*, where* `P` *is a process and* `s_1, ..., s_n` *are signals.*

The following syntax also means the same: `P/s_1,...,s_n`.

Definition 4.5 (Restriction). The scope of signals `s_1,...,s_n` is restricted to the process `P`, i.e., these signals are invisible outside `P`.

The local definition operator enables one to restrict the scope of a signal to a process. For an illustration, see Example 4.1.

4.3 Notation

Here, the notation of processes in general is first given in Sect. 4.3.1. For more details on this notation, the reader can also refer to the grammar given in Appendix B. A simple illustrative example is shown in Sect. 4.3.2. A few interesting aspects of process specification are also mentioned in Sects. 4.3.3 and 4.3.4: hierarchy and labeling, respectively.

4.3.1 Process Frame

The general form of a process frame model is as follows:

```
 1:  %process interface%
 2:  process PROCESS_MODEL =
 3:     { %parameters% }
 4:     ( ? %inputs%;
 5:       ! %outputs%; )
 6:     (|
 7:        %body of the process%
 8:      |)
 9:    where
10:      %local declarations%;
11: end; %end of PROCESS_MODEL%
```

This model contains:

- An *interface*: It includes the name of the process (here, `PROCESS_MODEL`) indicated after the keyword `process`, a set of static *parameters* (e.g., initializing constants; a parameter is a constant fixed at compile time); a set of *input* signals indicated by the symbol `?`; and a set of *output* signals indicated by the symbol `!`.
- A *body*: It describes, on the one hand, the internal behavior of the process and, on the other hand, the local declarations in the `where` part.

4.3.1.1 Comments in a Process

The special symbol `%` is used as a delimiter for statements that are only considered as *comments* in the following way:

```
%this is a trivial comment%.
```

The statement between the pair of symbols `%` will not be interpreted as a SIGNAL statement.

To indicate that a process is a function, the keyword `process` is replaced by `function` in the notation of the process model. As a result, the properties of its input/output parameters, i.e., synchronization and precedence, implicitly hold.

In other words, the programmer does not need to explicitly specify these properties as is the case in a process.

Remark 4.1. Note that the interface of a process may not contain parameters, inputs, or outputs (see Sect. 4.3.4). This is absolutely allowed in SIGNAL. When a process does not contain any input and output signals, it simply means that such a process does not communicate with the environment through its interface. However, the activation of such a process can be controlled via the notion of "label" (see Sect. 4.3.4).

Example 4.1 (Static parameters in processes). The following process, called `P1`, takes as a static parameter an integer `N`. The respective input and output signals are the integer `s` and the Boolean `cond`.

The local signal `s2` is used to store the product of `N` and `s1`. Then, it is used to define the output `cond`. As a result, the body of `P1` is composed of two equations that specify relations between the different signals.

```
 1:  process P1 =
 2:     { integer N }
 3:     ( ? integer s1;
 4:       ! boolean cond; )
 5:     (| s2 := N * s1
 6:      | cond := s1 > 32
 7:      |)
 8:    where
 9:      integer s2;
10: end; %process P1%
```

Another version of process `P1`, called `P2`, is defined below where there are neither static parameters nor local declarations. Here, `N` is equal to 4.

```
1:  process P2 =
2:     ( ? integer s1;
3:       ! integer s2;
4:         boolean cond; )
5:     (| s2 := 4 * s1
6:      | cond := s1 > 32
7:      |); %process P2%
```

Processes `P1` and `P2` express strictly the same behavior for `N` equal to 4 despite the syntactical differences of their respective definitions.

They can also be specified as functions: on the one hand, their input `s1` and outputs `s2` and `cond` are synchronous; and, on the other hand, `s1` precedes both `s2` and `cond`.

4.3.2 Example: A Resettable Counter

Let us consider a new specification of a resettable counter (see Fig. 4.1):

- Whenever the input signal `reset` holds the value `true`, the value of the counter is initialized with the value of the static parameter `v0`.

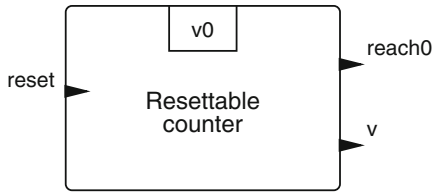


Fig. 4.1 A resettable counter

- The value of the counter is decreased by 1 each logical instant, and the output `v` indicates its current value.
- Whenever the current value becomes zero, the output signal `reach0` occurs with the value `true`. The input signal `reset` is assumed to be received from the environment only at these instants.

The following SIGNAL specification defines a signal `v`, which counts in decreasing order the number of occurrences of the events at which the Boolean signal `reset` holds the value *false*; `v` is reinitialized (with value `v0`) each time `reset` is *true*.

```
1:  process R_COUNTER =
2:   { integer v0; }
3:   ( ? boolean reset;
4:     ! boolean reach0;
5:       integer v; )
6:   (| zv := v $ 1 init 0
7:    | vreset := v0 when reset
8:    | zvdec := zv when (not reset)
9:    | vdec := zvdec - 1
10:   | v := vreset default vdec
11:   | reach0 := true when (zv = 1)
12:   |)
13:  where
14:    integer zv, vreset, zvdec, vdec;
15:  end; %process R_COUNTER%
```

The signal `v` is defined by `v0` whenever `reset` is present and has the value *true*; this is expressed by the equation at line `7` and partly by the equation at line `10`. Otherwise, it takes the value `zvdec − 1`; this is described by the equations at lines `9` and `10`. The signal `zvdec` is defined at line `8` as the previous value of `v`, represented by `zv`, when this value is present and `reset` is present with value *false*. Here, the `not` operator is defined by an instantaneous logical negation function.

The Boolean signal `reach0` is defined at line `11`, as being *true* when the previous value of `v` is `1`. Notice that an alternative definition of the signal `reach0` may use `v` instead of `zv`. In that case, the condition expression `(zv = 1)` becomes `(v = 0)`.

The process `R_COUNTER` cannot be considered as a function since its input and output parameters are not synchronous.

4.3.3 Hierarchy of Processes

The process model proposed in SIGNAL allows one to define subprocesses. These are declared in the `where` part as illustrated in the next example.

Example 4.2. Process `P2` contains a subprocess `Q`.

```
1:  process P2 =
2:     { integer N; }
3:     ( ? integer s1;
4:         boolean ok;
5:       ! integer s2; )
6:     (| tmp := Q{N}(s1)
7:      | s2:= tmp when ok
8:      |)
9:    where
10:     integer tmp;
11:     process Q =
12:        { integer M; }
13:        ( ? integer s1;
14:          ! integer s2; )
15:        (| s2 := s1 + M |);
16:   end; %process P2%
```

Here, note that even though processes `P2` and `Q` have in common signals that have the same identifiers, e.g., `s1` and `s2`, these signals do not necessarily designate the same objects.

4.3.4 Label of a Process

The context clock of a process `P`, which defines the set of instants at which an event at least occurs in `P` (i.e., `P` is active), can be explicitly specified using the notion of *label*. Such a clock is defined as the union of all clocks defined `P`.

In the following SIGNAL notation

```
l::P
```

the variable `l`, of `label` type, denotes the label of a process `P`.

SIGNAL defines a special data type for labels noted as `label`. The use of labels is typically useful when one needs to designate the clock of a process `P` which has no input and output.

Example 4.3 (The Stop_Self service).

Let us consider the following `STOP_SELF` process specifying a system call in an operating system. Whenever it is invoked by a running thread, this system call stops the thread.

```
1:  process Stop_Self =
2:    ( )
3:    (| thread_ID := Get_Active_Thread(true)
4:     | Release_Resources{}(thread_ID)
5:     | Set_State{}(thread_ID,#DORMANT)
6:     | Ask_For_Reschedule(true)
7:     |)
8:    where
9:      process Get_Active_Thread = ...;
10:     process Release_Resources = ...;
11:     process Set_State = ...;
12:     process Ask_For_Reschedule = ...;
13:     TID_type thread_ID;
14: end%process Stop_Self%;
```

In the corresponding SIGNAL model, there are no static parameters, inputs, and outputs in the interface as one can see.

Labels can be typically used to synchronize the invocation of the process, with the occurrence of the events denoting that a running thread completes. A possible specification is as follows:

```
1:  process Running_Thread =
2:    (? event exec;
3:     ! event finished; )
4:    (| % execute statements whenever the input ``exec'' occurs%
5:     | ... when exec
6:     | ...
7:     | % define the output ``finished'' when completed%
8:     | ...
9:     | finished := ...
10:    | stp::Stop_Self()
11:    | % here an equation must be defined to constrain finished
12:     % and stp to have the same clock (see the next chapter on
13:     % Extended Constructs)%
14:    |)
15:   where
16:     process Stop_Self() = ...;
16:     label stp;
17:     ...
18: end%process Running_Thread%;
```

Here, the label `stp` defined at line `10` is synchronized with the `finished` output event that denotes the completion of the running thread. The extended construct that enables one to make two or more signals synchronous is presented in Chap. 5. As a result, the `Stop_Self` system call is effectively executed at that time.

4.4 Exercises

4.1. Define a process `Sum` in which on each occurrence of its unique input `a` of `real` type, the sum of the values of `a` that have occurred until now is computed in the output signal `s`.

4.2. Define a process `Average` in which on each occurrence of its unique input `a` of `real` type, the average of the values of `a` that have occurred until now is computed in the output signal `avg`.

4.3. Given a constant parameter `N` of `integer` type, define a process `AverageN` in which on each occurrence of its unique input `A` of `real` type, the average of the `N` previous values of `A` is computed in the output signal `avg`.
Note: `a$N` may be initialized to 0.0 by using the following expression: `init[to N:0.0]`.

4.4. Let `a` be a positive `integer` signal. Define a signal `max` in a process `Maximum` which takes at the current logical instant the most recent maximum value held by `a`.

4.5. Define a subtraction between `A` and `B` of event types which gives instants where `A` is present but `B is not`.

4.6. Indicate among all the above processes which ones are also functions.

Expected Learning Outcomes:

- A **process** is a set of equations that express relations between signals to describe the behavior of a system.
- A **function** is a particular process in which both inputs and outputs are synchronous and all inputs precede all outputs within a logical instant.
- There are two primitive operators on processes:
  - **Composition**: union of equations
  - **Local declaration**: restriction of a signal visibility in a given process
- A process can be **hierarchical**. In that case, it includes one or more sub-processes that are invoked in the main part of this process.

References

1. Benveniste A, Le Guernic P, Jacquemot C (1991) Synchronous programming with events and relations: the SIGNAL language and its semantics. Science of Computer Programming, Elsevier North-Holland, Inc, Amsterdam, The Netherlands 16(2):103–149
2. Besnard L, Gautier T, Le Guernic P (2008) SIGNAL v4 – INRIA Version: Reference Manual. Available at: http://www.irisa.fr/espresso/Polychrony
3. Le Guernic P, Gautier T, Le Borgne M, Le Maire C (1991) Programming real-time applications with SIGNAL. Proceedings of the IEEE 79(9):1321–1336
4. Le Guernic P, Talpin J-P, Le Lann J-C (2003) Polychrony for system design. Journal for Circuits, Systems and Computers 12(3):261–304

Chapter 5
Extended Constructs

Abstract This chapter presents further constructs of the SIGNAL language. The definition of these constructs is derived from a combination of the previous primitive constructs to provide the user with suitable *macros*. Section 5.1 shows different operators that allow one to express control-related properties by only specifying clock relations. These operators are very useful in practice when describing polychronous programs. Sections 5.2, 5.3, and 5.4, respectively, present three macro constructs that enable one to define a more general memorization mechanism, sliding windows on a sequence of signal values, and an array of processes, which are very helpful when defining data-parallel algorithms.

5.1 Pure Control Specification

© *CNRS Photothèque – Jean-Marc RENNES*

In SIGNAL, clocks offer a suitable means to describe control in a given SIGNAL description of a system. Several derived operators [1] have been identified that enable one to explicitly manipulate clocks. They are introduced below. For all these operators, a corresponding semantics is defined in terms of the SIGNAL primitive constructs, introduced in Chaps. 3 and 4.

In the remainder of this section, the following constructs are presented: clock extraction (Sect. 5.1.1), clock synchronization (Sect. 5.1.2), set operations applied to clocks such as union and intersection (Sect. 5.1.3), and clock comparison operators (Sect. 5.1.4). These clock manipulation operators offer a very convenient means to specify either partial or total relations between the different clocks associated with the components of a multiclocked system.

A. Gamatié, *Designing Embedded Systems with the SIGNAL Programming Language: Synchronous, Reactive Specification*, DOI 10.1007/978-1-4419-0941-1_5,
© Springer Science+Business Media, LLC 2010

5.1.1 Extraction of Abstract Clocks

5.1.1.1 Boolean Signal Value

Syntax 7 *In the equation* `clk := when b`, *the signal* `clk` *of* `event` *type represents the set of instants at which* `b` *holds the value true.*

Definition 5.1 (Clock extraction from Booleans). This equation can be expressed using primitive constructs as

```
clk := b when b.
```

The following statement checks whether or not an integer signal is null: `is_null := when(s = 0)`. An associated trace is as follows:

t :	t_0	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}	...
s :	·	·	1	0	·	0	6	·	4	0	0	...
is_null :	·	·	·	t	·	t	·	·	·	t	t	...

5.1.1.2 Signal Presence

Syntax 8 *In the equation* `clk := ^s`, *the signal* `clk` *of* `event` *type represents the clock of* `s`.

Definition 5.2 (Clock extraction). This equation can be expressed using primitive constructs as

```
clk := (s = s).
```

The following trace corresponds to `clk := ^s`:

t :	t_0	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}	...
clk :	·	·	t	·	·	t	t	t	·	t	t	...
s :	·	·	3	·	·	1	9	8	·	3	5	...

5.1.2 Synchronization of Abstract Clocks

Syntax 9 *The equation* s_1 `^=` s_2 *specifies that signals* s_1 *and* s_2 *are synchronous.*

Definition 5.3 (Clock synchronization). This equation is equivalently defined as

```
(| clk := (^s1 = ^s2) |) where clk.
```

The following trace corresponds to s_1 `^=` s_2:

t :	t_0	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}	...
s_1 :	·	·	3	·	·	1	9	8	·	3	5	...
s_2 :	·	·	2.5	·	·	3.8	0.1	9.2	·	3.3	4.5	...

The synchronization operator (`^=`) is trivially extended to several signals in the following form: s_1 `^=` $\cdots$ `^=` s_n.

5.1.3 Set Operations on Abstract Clocks

SIGNAL defines operations such as clock intersection, union, and difference.

Syntax 10

- *Empty clock:* `^0` *specifies the empty clock (i.e., contains no instant).*
- *Lower bound (or intersection): the equation* `clk := s1 ^* s2` *defines the signal* `clk` *as the intersection of the clocks of signals* s_1 *and* s_2.
- *Upper bound (or union): the equation* `clk := s1 ^+ s2` *defines the signal* `clk` *as the union of the clocks of signals* s_1 *and* s_2.
- *Relative complement: the equation* `clk := s1 ^- s2` *defines the signal* `clk` *as the difference of the clocks of signals* s_1 *and* s_2.

The following trace illustrates the statements $\texttt{clk}_1$ `:= s1 ^* s2`, $\texttt{clk}_2$ `:= s1 ^+ s2` and $\texttt{clk}_3$ `:= s1 ^- s2` :

t :	t_0	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}	…
s_1 :	·	·	3	·	1	·	9	·	3	5	6	…
s_2 :	·	·	2.5	·	·	3.8	0.1	·	3.3	4.5	·	…
$\texttt{clk}_1$:	·	·	*t*	·	·	·	*t*	·	*t*	*t*	·	…
$\texttt{clk}_2$:	·	·	*t*	·	*t*	*t*	*t*	·	*t*	*t*	*t*	…
$\texttt{clk}_3$:	·	·	·	·	*t*	·	·	·	·	·	*t*	…

Definition 5.4 (Set operations on clocks). These definitions are equivalent to:

- Empty clock: equivalent to ∅
- Lower bound: `clk := ^s1 when ^s2`
- Upper bound: `clk := ^s1 default ^s2`
- Relative complement: `clk := when((not ^s2) default ^s1)`

Example 5.1 (Event observation).

Let `start` and `finish` be two distinct `event` signals that occur independently from each other. The questions of interest here are the following:

1. *How can one evaluate the number of logical instants between the occurrences of both signals?*
2. *How can one detect the occurrence of a given signal between the occurrences of both signals?*

Counting the number of instants.

To answer the first question, a clock `clk` is considered. The number of instants between the occurrences of `start` and `finish` will be determined on the basis of this clock, which may be seen as a reference clock. In other words, we are going to count the number of instants of `clk`.

A signal `cnt` plays the role of a counter. It is present whenever any event occurs, but is increased by 1 only when `clk` occurs. It is reset to 0 on `start`, whereas its value indicates the duration on the occurrence of `finish`.

Such a situation is described by the system of equations given below.

```
1: process Counting_Instants =
2:   ( ? event start, finish, clk;
3:     ! integer cnt, duration; )
4:   (| cnt ^= start ^+ finish ^+ clk
5:    | prev_cnt := cnt $ 1 init 0
6:    | cnt := 0 when start default
7:             prev_cnt + 1 when clk default
8:             prev_cnt
9:    | duration := cnt when finish
10:   |)
11: where
12:    integer prev_cnt;
13: end %Counting_Instants%;
```

The following trace corresponds to a possible correct behavior:

t :	t_0	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}	t_{11}	...
clk:	·	*t*	*t*	*t*	·	*t*	*t*	*t*	*t*	*t*	*t*	·	...
start:	·	·	*t*	·	·	*t*	·	·	·	·	*t*	·	...
finish:	·	·	·	·	*t*	·	·	*t*	*t*	·	*t*	*t*	...
cnt:	·	1	0	1	1	0	1	2	3	4	0	0	...
prev_cnt:	·	0	1	0	1	1	0	1	2	3	4	0	...
duration:	·	·	·	·	1	·	·	2	3	·	0	0	...

Checking the presence of a signal.

Now, let us take a look at the second question, according to which one would like to know if a signal `s` is present between the respective occurrences of `start` and `finish`.

For that, at any instant the fact that one is *within the occurrence interval* or *out of this interval* should be memorized. This is achieved via the Boolean signal called `mem`.

```
1: process Checking_Presence =
2:   ( ? event start, finish, s;
3:     ! integer in; )
4:     (| mem ^= start ^+ finish ^+ s
5:      | prev_mem := mem $ 1 init false
6:      | mem := start
7:               default (not finish)
8:               default prev_mem
9:      | in := mem when s
10:     |)
11:   where
12:    event mem, prev_mem;
13: end %Checking_Presence%;
```

The trace shown below illustrates a possible scenario:

t :	t_0	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}	t_{11}	...
s:	·	*t*	·	*t*	·	*t*	·	·	*t*	·	*t*	·	...
start:	·	·	*t*	·	·	*t*	·	·	·	·	*t*	·	...
finish:	·	·	·	·	*t*	·	·	*t*	*t*	·	*t*	*t*	...
mem:	·	*f*	*t*	*t*	*f*	*t*	·	*f*	*f*	·	*t*	*f*	...
prev_mem:	·	*f*	*f*	*t*	*t*	*f*	·	*t*	*f*	·	*f*	*t*	...
in:	·	·	·	*t*	·	*t*	·	·	·	·	*t*	·	...

Following the definition given above, if the signal *s* occurs on *start*, it is in the interval. If it occurs on *finish*, it is out of the interval. What about an occurrence on both *start* and *finish*?

5.1.4 Comparison of Abstract Clocks

SIGNAL also defines predicates that mainly consist of inclusion and intersection emptiness and equality.

Syntax 11

- *Inferiority: the statement* s_1 `^<` s_2 *specifies a set inclusion relation between the clocks of signals* s_1 *and* s_2.
- *Superiority: the statement* s_1 `^>` s_2 *specifies a set containment relation between the clocks of signals* s_1 *and* s_2.
- *Exclusion: the statement* s_1 `^#` s_2 *specifies that the intersection of the clocks of signals* s_1 *and* s_2 *is empty.*
- *Equality:* s_1 `^=` s_2 *(see Sect. 5.1.2).*

Definition 5.5 (Clock comparison). These definitions are equivalent to:

- Inferiority: s_1 `^=` s_1 `^*` s_2
- Superiority: s_1 `^=` s_1 `^+` s_2
- Exclusion: `^0` `^=` s_1 `^*` s_2
- Equality: see Sect. 5.1.2

The next two derived constructs enable one to describe memorization and sliding window on the series of values carried by a signal.

5.2 Memorization

The memorization operator introduced here mainly differs from the *delay* primitive operator in that it does not require all its arguments to have the same clock.

Syntax 12 *The equation* `s2 := s1 cell b init c` *allows one to memorize the values of a signal* `s1` *where* `s2` *is a signal of the same type as* `s1`*,* `b` *is a Boolean signal, and* `c` *is an initializing constant.*

Definition 5.6 (Memorization). This equation is equivalent to

```
(|s2 := s1 default (s2 $ 1 init c) | s2 ^= s1 ^+ (when b) |).
```

Here, the signal s_2 takes the value of s_1 when s_1 is present. Otherwise, if s_1 is absent and the Boolean signal b is present with value *true*, the value of s_2 is the latest one from s_1. If b is present with value *false* before the first occurrence of s_1, s_2 is initialized with constant c.

The clock of s_2 is the union of the clocks of s_1 and the set of instants where b holds the value *true*.

For the process `s2 := s1 cell b init 0` where signals s_1 and s_2 are of integer type, and b is a Boolean signal, a possible execution trace is as follows:

t :	t_0	t_1	t_2	t_3	t_4	t_5	t_6	...
s_1 :	·	1	2	·	·	3	4	...
b :	t	·	f	t	f	f	t	...
s_2 :	0	1	2	2	·	3	4	...

There exists a variant of the memorization operator, which is context-dependent. In the equation `s2 := var s1 init c`, the signal s_2 always carries the latest value of s_1. However, the main difference from the above operator is that here the clock of s_2 is defined by the context in which s_2 is used.

5.3 Sliding Window

The *sliding window* operator [1] enables one to get access to the k most recent values carried by a signal, where k is a finite integer.

Syntax 13 *The equation* `s2 := s1 window k init tab` *defines a sliding window of constant size* `k` *on a signal* `s1`*.*

Definition 5.7 (Sliding window). The associated semantics is as follows:

$$(\forall t \geq 0) \begin{cases} (t + i \geq k) \Rightarrow (Y_t[i] = X_{t-k+i+1}) & \vee \\ (1 \leq t + i < k) \Rightarrow (Y_t[i] = \texttt{tab_init}[t - k + i + 2]). \end{cases}$$

In the above equation, signal s_2 is an array of size $k \geq 1$ whose elements are of the same type as those of signal s_1 and tab is an array of dimension $k' \geq k - 1$ that contains initialization values.

Let us consider the equation: `(| y := x window 3 init [-1,0] |)`, where x is an integer signal and y is a 3-array of integers. Then, a possible execution trace is given below:

t :	t_0	t_1	t_2	t_3	t_4	t_5	t_6	...
s_1 :	·	1	2	·	·	3	4	...
s_2 :	·	$[-1, 0, 1]$	$[0, 1, 2]$	·	·	$[1, 2, 3]$	$[2, 3, 4]$	...

5.4 Array of Processes

Arrays of processes [1] are useful constructs that allow one to describe regular computations achieved by a process. Their syntactic expression is as follows:

Syntax 14 *The expression* `array i to N of <process> with <init_process> end` *defines an array from* 0 *to* `N` *of* `process` *with* `init_process` *as the initial executed process.*

Definition 5.8 (Array of processes). The associated semantics is as follows: `process` is uniformly instantiated `N`+1 times on a regular data structure (e.g., an array). The first instance, corresponding to index 0, is `init_process`.

Example 5.2 (Vector reduction). The following process defines an integer `reduc` resulting from the reduction of a vector `vect` by the sum operator.

```
1: process Reduction =
2:  { integer N; }
3:  ( ? [N] integer vect;
4:    ! integer reduc; )
5:  (| array i to N-1 of
6:       reduc := vect[i] + reduc[?]
7:      with reduc := 0
8:     end
9:   |);
```

The notation `reduc[?]` means the latest value of the signal `reduc` at each logical instant.

5.5 Exercises

5.1. Discuss the difference that may exist between the following two statements:

- `when ((c=1) default (c=2) default (c=3))`
- `when (c=1) default when (c=2) default when (c=3)`

5.2. Simplify the following SIGNAL expressions into equivalent expressions:

1. `when (not ^s) default ^s`
2. `s when (^s)`

5.3. Let `s` be a signal of `event` type.

1. Simplify the expression `true when (^s)`.
2. Is it possible to detect the instants at which `s` is absent by considering the expression `when not (^s)`?

5.4. Let us consider a very simple model of a microwave oven, which has a unique button. This button enables one to start and stop the oven. A signal `button` of `event` type is associated with this button.

Define a process that emits a Boolean output signal `busy`, which has the same clock as `button`. The value of `busy` reflects the state of the oven: *true* for active and *false* for idle.

5.5. Let us consider a positive `integer` signal denoted by `s`.

1. Define a Boolean signal `first` which holds the value *true* only on the first occurrence of `s` and *false* otherwise (try to find a solution without using a counter).
2. Define a signal `up` of `event` type that is present whenever the current value carried by `s` is greater than the previous one. Any two consecutive values of `s` are assumed to be different. For the initial comparison, suppose that the value preceding the initial value of `s` is 0.
3. Define a signal `top` of event type which is present whenever a local maximum is reached among values of `s`. This can be detected on the occurrence of the first decreasing value of `s` after the values had previously been increasing. Any two consecutive values of `s` are still assumed to be different.
4. Let us assume that `s` may hold consecutive identical values. There is a local maximum when the values of `s` decrease after previously increasing and before they become stable (i.e., identical values). Define the local maximal signal `top_bis` in this case.
5. Define a signal `local_max` that only takes the values of `s`, which represents local maxima.

5.6. Let `s1` and `s2` be two signals of any type and independent of each other. Define a signal `present` of `event` type that occurs whenever `s1` and `s2` are present at the same time. Does the expression `^s1 and ^s2` adequately define such a signal?

5.7. Given an input signal `s`, define a process that extracts the values carried by `s` every `N` steps as follows: $s_0, s_{N-1}, s_{2N-1}, \ldots$ (the `modulo` operator could be used).

5.8. Define an `event only_one` that provides notification that either signal `s1` or signal `s2` is present and that both signals are not present at the same time.

5.9. As an addition, `integer` signals `s1` and `s2` must be synchronous. Now, suppose they can arrive at "slightly" different logical instants; we want nevertheless to add them! Define a process `Addition` that describes their addition.

s1	:	·	·	2	·	4		9	· …
s2	:	5	7	·	·	·		6	3 …
sum	:	·	·	7	·	11		15	· …

5.10. Define a vector `v3` denoting the product of two vectors `v1` and `v2` of the same length.

5.11. Define an integer unit matrix `u`.

Expected Learning Outcomes:

- In SIGNAL programming, **clocks** can be directly manipulated by using specific **operators** for

  - The extraction of the clock information corresponding to a signal
  - The synchronization of two or more clocks
  - The combination of two or more clocks: union, intersection, difference
  - The comparison of clocks: equality, inferiority, superiority, etc.

  These operators are derived from the primitive operators. Thus, the former can be fully expressed with the latter.
- There is a more general **memorization operator** that relies on the delay primitive operator. Their main difference is that the general one is a multi-clock operator, whereas the primitive one is a monoclock operator.
- There is a derived operator, called **sliding window**, which enables one to get access to a constant-sized set of elements that precede a given element in the sequence of values corresponding to a signal.
- SIGNAL proposes an **iterative structure** that applies to processes, which is specifically interesting when describing **data-parallel algorithms**.

Reference

1. Besnard L, Gautier T, Le Guernic P 2008 SIGNAL v4 – INRIA Version: Reference Manual. Available at: http://www.irisa.fr/espresso/Polychrony

Chapter 6
Design in POLYCHRONY: First Steps

Abstract The aim of this chapter is to give a flavor of the design activity within the POLYCHRONY environment, which is dedicated to SIGNAL. Several useful details are provided to facilitate the first steps of new practitioners of SIGNAL programming. The chapter presents the specification of an example and how the resulting description is compiled so as to automatically generate an executable code for behavioral simulation. The example chosen is deliberately simple for the sake of clarity. As a complement to this chapter, the reader can also refer to Chap. 13, which also presents the design of a complex example, by addressing more issues. A brief introduction to the POLYCHRONY environment is given in Sect. 6.1. Then, Sect. 6.2 focuses on the SIGNAL design of a watchdog process as an example.

6.1 The POLYCHRONY Design Environment

POLYCHRONY is the academic programming environment dedicated to the SIGNAL language. It was defined at the Institut de Recherche en Informatique et Systèmes Aléatoires (IRISA) laboratory[1] in Rennes, France. Its corresponding commercial version, referred to as RT-BUILDER, was developed by Geensys (http://www.geensys.com). It has been used in several industrial-scale projects by Snecma/Hispano-Suiza and Airbus Industries.

6.1.1 What Is It Useful for?

The POLYCHRONY environment is particularly suitable for the development of safety-critical systems, from abstract specification until deployment on distributed systems. It relies on the application of formal methods, enabled by the representation of a system, at the different abstraction levels during its development, using the SIGNAL multiclock semantic model.

[1] http://www.irisa.fr

A. Gamatié, *Designing Embedded Systems with the SIGNAL Programming Language: Synchronous, Reactive Specification*, DOI 10.1007/978-1-4419-0941-1_6,
© Springer Science+Business Media, LLC 2010

POLYCHRONY offers different facilities within a formal framework that are useful for the:

- High-level formal specifications
- Validation of a design at different levels
- Refinement of descriptions according to a top-down approach
- Abstraction of properties needed for composition
- Assembly of predefined components according to a bottom-up approach, e.g., with commercial off-the-shelf components

6.1.2 What Tools Does It Provide?

The toolset that forms POLYCHRONY is distributed at the following address: http://www.irisa.fr/espresso/Polychrony. It is mainly composed of the following elements:

- A graphical user interface with interactive access to compiling functionalities.
- A SIGNAL batch compiler providing a set of functionalities considered as a set of services, e.g., for program transformations, optimizations, formal verification, abstraction, separate compilation, mapping on distributed architecture, code generation (C, C++, Java), simulation, and temporal profiling. These transformations can be applied according to the needs of a user when using the compiler.
- The SIGALI tool, an associated formal system for formal verification and controller synthesis (for more details on this, the reader is referred to http://www.irisa.fr/vertecs/Softwares/sigali.html).

6.2 Design of a Watchdog

Let us consider the watchdog process mentioned in Sect. 1.1, and depicted in Fig. 6.1. Only its interface is shown, which is composed of a single static parameter, three input parameters, and a single output parameter.

The addition of a SIGNAL specification can be done either textually with the user's favorite editor or with the graphical editor provided in POLYCHRONY. In the sequel, we only consider the former possibility.

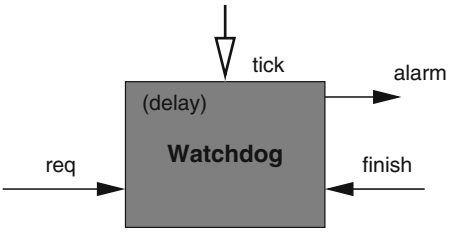


Fig. 6.1 The watchdog component

The solution presented below for the design of the watchdog process relies on [1] (Sect. 6.2.1). The compilation and simulation of this solution in practice are also presented (Sects. 6.2.2, 6.2.3).

6.2.1 *Definition of a* SIGNAL *Specification*

```
1: process Watchdog =
2: { integer delay; }
3: ( ? integer req;
4:     event finish, tick;
5:   ! integer alarm; )
```

The five lines above specify the interface of the `Watchdog` process. Here, the unique parameter is `delay`. There are three input signals (`req, finish, tick`) and one output signal (`alarm`).

```
6: (| hour ^= tick
7:  | hour := (hour$ init 0) + 1
```

Lines `6` and `7` define a signal `hour` that counts how many occurrences of the signal `tick` have been received: these signals are made to be simultaneously present (i.e., they are synchronous) by the statement (line `6`), whereas `hour` is increased by 1 whenever it occurs (line `7`). Note that the signal `hour` is declared locally (see line `17`). In addition, an alternative definition of the equation defined at line `7` is as follows:

```
7.1:  | prev_hour := hour$ init 0
7.2:  | hour := prev_hour + 1
```

which may facilitate the readability of the equation.

```
8:  | cnt ^= tick ^+ req ^+ finish
9:  | prev_cnt := cnt$ init (-1)
10: | cnt := (delay when ^req)
11:     default (-1 when finish)
12:     default ((prev_cnt - 1) when (prev_cnt >= 0))
13:     default  -1
```

The duration between the receipt of `req` and the completion of the associated treatment is counted by a decreasing counter variable, called `cnt` (`count` is a SIGNAL reserved keyword!).

The initial value of `cnt` is set to `delay` when `req` is received (line `10`). Then, it decreases every `tick`, until either the signal `finish` occurs or it reaches 0 while `finish` is still absent.

Line `8` defines the set of instants at which `cnt` is present: this set is the union of the clocks of all input signals (i.e., the set of instants where at least one of these signals is present). At each instant the signal `cnt` is present, `prev_cnt` records its previous values (line `9`).

```
14:   | alarm := hour when (cnt = 0)
```

The alarm signal takes the value of hour when the decreasing counter cnt reaches 0. This means that the signal finish has not arrived in time.

```
15:    |)
```

This line terminates the set of statements. These statements could be written in any order: the compiler analyses their dependencies, and determines the moments where they must be executed. Line 17 defines local signals.

```
16:  where
17:    integer hour, cnt, prev_cnt;
18: end % Watchdog%;
```

To simulate the above SIGNAL specification, one needs to generate a corresponding code in one of the target programming languages in POLYCHRONY: C, C++, or Java. The following steps are therefore considered:

1. Compilation and generation of the target code by using the right command options (see Chap. A of the Appendix, on page 211) and a corresponding executable code
2. Definition of input values in separate data files
3. Execution of the executable code to simulate a given scenario

In the next section, the C language is considered as a target for the generation.

6.2.2 Compilation and Code Generation

The compilation of a SIGNAL program enables one to detect different kinds of specification issues: syntax, type and declaration errors, violation of the single assignment principle (inherent to declarative languages), clock synchronization problems, etc. For more details on these aspects, the reader is referred to Chap. 9.

A code is automatically generated in C from a SIGNAL specification by the compiler only when none of the above issues are encountered. The Watchdog process defined previously is correct regarding these issues. So, one can safely ask for the generation of the corresponding C code via the following command:

```
signal -c Watchdog.SIG -par=Watchdog.PAR,
```

where signal is the basic command, -c means the C language as target, and -par= indicates the location file of the static parameters of the Watchdog process. The files Watchdog.SIG and Watchdog.PAR, respectively, contain the SIGNAL specification of the process and the list of the static parameters of the process.

As a result of the above command invocation, a subdirectory is automatically created, which contains different generated files. The file named Watchdog_body.c is central in that it describes the main behavior of the specified process (here, the static parameter value specified in Watchdog.PAR is 2):

```
/*
  Generated by Polychrony version  V4.15.10
*/
#include "Watchdog_types.h"
#include "Watchdog_externals.h"
#include "Watchdog_body.h"

/*  ==> parameters and indexes */
/*  ==> input signals     */
static int req;
static logical C_req, C_finish, C_tick;
/*  ==> output signals    */
static int alarm;
/*  ==> local signals     */
static int hour, cnt;
static logical C_60, C_prev_cnt, C_78, C_81, C_89, C_alarm;

EXTERN logical Watchdog_initialize()
{
    cnt = -1;
    hour = 0;
    Watchdog_STEP_initialize();
    return TRUE;
}

static void Watchdog_STEP_initialize()
{
    C_78 = FALSE;
}

EXTERN logical Watchdog_iterate()
{
    if (!r_Watchdog_C_req(&C_req)) return FALSE;
    if (!r_Watchdog_C_finish(&C_finish)) return FALSE;
    if (!r_Watchdog_C_tick(&C_tick)) return FALSE;
    C_60 = C_req || C_finish;
    C_prev_cnt = C_60 || C_tick;
    if (C_prev_cnt)
        {
        C_78 = cnt >= 0;
        }
    C_81 = (C_prev_cnt ? C_78 : FALSE);
    if (C_req)
        {
        if (!r_Watchdog_req(&req)) return FALSE;
        }
    if (C_tick)
        {
        hour = hour + 1;
        }
    if (C_prev_cnt)
```

```
        {
        if (C_req) cnt = 2; else if (C_finish) cnt = -1;
              else if (C_78) cnt = cnt - 1; else cnt = -1;
        }
    C_89 = (C_prev_cnt ? (cnt == 0) : FALSE);
    C_alarm = C_tick && C_89;
    if (C_alarm)
        {
        alarm = hour;
        w_Watchdog_alarm(alarm);
        }
    Watchdog_STEP_finalize();
    return TRUE;
}

EXTERN logical Watchdog_STEP_finalize()
{
    Watchdog_STEP_initialize();
    return TRUE;
}
```

In this C code, the input, output, and local variables are first declared. Then, some initialization functions, represented by `Watchdog_initialize()` and `Watchdog_STEP_initialize()`, are defined. They initialize all manipulated state variables before and after each reaction, respectively.

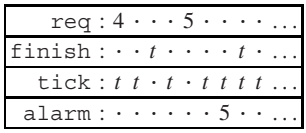
A reaction at each logical instant is defined by the function `Watchdog_iterate()`. In this function, input data are first read from their associated data files. For instance, `r_Watchdog_C_tick` is a function (generated automatically in another file) that enables the detection of the presence of the `tick` input signal at a given logical instant by reading its associated data file.

Once all required inputs have been read, the output values of the current reaction are computed. This is a large part of the function `Watchdog_iterate()`. These values are written in the corresponding output signals. Here, the function `w_Watchdog_alarm()` writes the current `alarm` output value in its corresponding output file.

From the C code shown above, an executable code can be straightforwardly produced. For that purpose, POLYCHRONY allows one to automatically generate a suitable `Makefile` (see Chap. 13, Appendix A).

6.2.3 Behavioral Simulation

Now, let us concentrate on the simulation of the `Watchdog` process. The following trace illustrates an expected scenario where the static parameter `delay` is equal to 2, i.e., the file `Watchdog.PAR` contains the value 2:

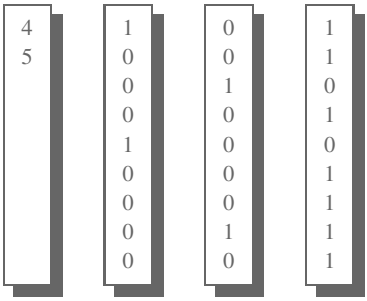


To simulate such a scenario, we have to define the following files for input signals:

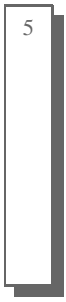
- `Rreq.dat`: The sequence of integer values taken by the signal `tick`.
- `RC_req.dat`: The presence/absence of `req`, i.e., its clock. It consists of a sequence of values from $\{0, 1\}$, where 0 and 1 denote absent and present, respectively.
- `RC_finish.dat`: The presence/absence of `finish`.
- `RC_tick.dat`: The presence/absence of `tick`.

In the above file names, the extension `.dat` means "data." The initial letter `R` is used for "read" and the letter `C` is used for "clock."

The exact contents of the data files are shown below from left to right:



With these input files, the execution of the `Watchdog` executable code produces an output file called `Walarm.dat`. Here, the initial letter `W` means "write." This file contains the computed values for the `alarm` output signal for the entire set of reactions, deduced from the input data files. The content of `Walarm.dat` is as follows:



In this textual form, it is not clear at which reaction the value 5 has been produced. To see that, one solution consists in adding an additional output signal in the `Watchdog` process which represents the clock of the signal `alarm`. Let `C_alarm` denote this signal.

Now, line 5 in the above SIGNAL specification of the process becomes

```
! integer alarm; boolean C_alarm; )
```

An additional equation is specified to define `C_alarm`:

```
| C_alarm := ^alarm default false
```

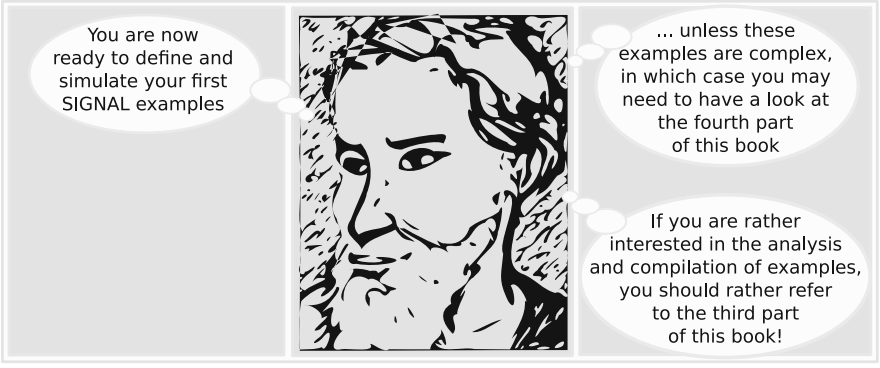
Finally, line 5 is modified to synchronize `C_alarm` and `cnt`, which holds a very frequent clock:

```
| cnt ^= tick ^+ req ^+ finish ^= C_alarm
```

By applying the same code-generation process to the modified process, the execution generates `WC_alarm.dat` as an output file besides `Walarm.dat`. The content of this additional file is as follows:

```
0
0
0
0
0
0
1
0
0
```

This clearly shows that the unique computed value 5 of the `alarm` signal has been produced at the seventh reaction as expected in the initial trace scenario.



6.3 Exercises

6.1. Let us consider the `Counting_Instants` process defined in Chap. 5 (Sect. 5.1.3).

1. Generate the corresponding C code
2. Simulate the trace scenario illustrated in the example

6.2. Define a signal `present` that indicates the number of items stored in a box. The insertion and removal of items are, respectively, denoted by event signals `in` and `out` (both signals may occur at the same instant).

6.3. Let us consider an image processing context. The pixels obtained from a sequence of pictures are represented as Boolean data, received line by line, and for each line, column by column. A picture has NL lines and NC columns. NL and NC are constant parameters. The last pixel of a picture is followed by the first pixel of the next picture.

Define for each pixel the corresponding line and column number. For instance, the first and second pixels are, respectively, associated with the following couples of rank values: $(0, 0)$ and $(0, 1)$.

6.4. A two-track railway crossing is protected by a barrier. Each track is closed or opened depending on the value indicated in a Boolean signal $\texttt{close}_i$ ($i \in \{1, 2\}$): *true* means to close and *false* means to open.

Define the barrier's controller as a Boolean signal `close` that indicates *true* when both tracks are closed and *false* when they are opened, as illustrated in the trace given below:

$\texttt{close}_1$:	t	·	·	t	f	t	f …
$\texttt{close}_2$:	t	·	f	f	f	t	· …
close :	t	·	·	·	f	t	· …

6.5. Resynchronizing, let `s` be a signal, and `clk` be a clock faster than that of `s`. The resynchronization of `s` on `clk` consists in delaying its occurrences (without repetition) at the next occurrences of `clk` whenever `clk` is absent. When `s` occurs simultaneously with `clk`, this resynchronization operation is not necessary. An illustrative trace is as follows:

s :	1	·	·	2	·	·	3 …
clk :	·	·	t	·	t	·	t …
s on clk :	·	·	1	·	2	·	3 …

Define such a behavior.

6.6. Given a matrix M, define the vector `D` that extracts the diagonal elements of M.

Expected Learning Outcomes

- POLYCHRONY is the **programming environment** associated with the SIGNAL language. It includes
  - A graphical user interface
  - A powerful batch compiler
  - A model checker called SIGALI
- Given a system to be designed with POLYCHRONY, the following steps should be taken into account
  1. **Coding** in SIGNAL, either textually or graphically
  2. **Program analysis** with the compiler to address different issues, e.g., syntax, type errors, single assignment, clock synchronization
  3. **Automatic code generation** in a target language, e.g., C, C++, or Java, when the program is correct
  4. **Behavioral simulation** by defining the values of all input parameters via data files

Reference

1. Houssais B (2004) Synchronous Programming Language SIGNAL: A Tutorial. Teaching notes. Available online at the following address: http://www.irisa.fr/espresso/Polychrony

Chapter 7
Formal Semantics

Abstract This chapter addresses the semantical aspects of the SIGNAL language. It presents two kinds of semantics: an operational semantics in Sect. 7.1 and a denotational semantics in Sect. 7.2. In the operational approach to semantics, the meaning of a program is described in terms of its execution steps on an abstract machine. Thus, operational semantics assigns to a program, a computation structure (a graph), representing all possible executions of a program on an abstract machine. In his seminal note, Plotkin (*Journal of Logic and Algebraic Programming* 60–61:3–15, 2004) proposed a structural view on the operational semantics. This view has now become the de facto standard for giving operational semantics to languages. The denotational semantics of programming languages was originally studied by Scott and Strachey (in *Proceedings of the Symposium on Computers and Automata*, Polytechnic Institute of Brooklyn, pp. 19–46, 1971). It aims at describing how the meaning (also referred to as the denotation) of a valid program is interpreted as a function from its inputs to its outputs. The two semantics of SIGNAL presented in this chapter were originally proposed by Benveniste et al. (*Science of Computer Programming* 16:103–149, 1991) and Le Guernic et al. (*Journal for Circuits, Systems and Computers* 12:261–304, 2003). For each semantics, some preliminary notions are first defined. Then, they are used to describe properly the semantics of the basic constructs of the language.

7.1 An Operational Semantics

An operational semantics of SIGNAL primitive constructs is presented by considering a state transition system through an inductive definition of the set of possible transitions. Section 7.1.1 introduces the basic notions that serve to define the semantics of the language primitive constructs in Sects. 7.1.2 and 7.1.3.

A. Gamatié, *Designing Embedded Systems with the SIGNAL Programming Language: Synchronous, Reactive Specification*, DOI 10.1007/978-1-4419-0941-1_7,
© Springer Science+Business Media, LLC 2010

7.1.1 Preliminary Definitions

We denote by $\mathcal{X}$ and $\mathcal{V}$, respectively, the set of signal variables (i.e., identifiers or names of signals, see page 50) and their associated domain of values. The set $\mathcal{V}^{\perp}$ represents $(\mathcal{V} \cup \{\perp\})$, where $\perp$ still means absence of signals at a logical instant. Signal variables may be assigned values according to an execution context or *environment*.

Definition 7.1 (Environment). Let $X \subset \mathcal{X}$ represent a set of signal variables. An environment associated with X is defined as a function

$$\varepsilon : X \to \mathcal{V}^{\perp}$$

that associates values with signal variables at each instant during an execution.

When the signal corresponding to a variable s is present, the environment assigns an effective value $v \in \mathcal{V}$ to s. Otherwise, s is associated with the special value $\perp$.

Figure 7.1 illustrates two different environments $\varepsilon^1 : X \to V_1$ and $\varepsilon^2 : X \to V_2$ that share the same subset of signal variables, but have disjoint subdomains of values V_1 and V_2. Notice that in ε^1, the signal variable s_3 is associated with $\perp$, whereas in ε^2 it holds the value v_{23}.

In the sequel, when no confusion is possible, signals are referred to via their corresponding variables.

Given a set of values V, the set of environments associated with $X \subset \mathcal{X}$ is denoted by ε_X. In addition, a signal $s \in \mathcal{X}$ that is present in the execution environment with the value $v \in \mathcal{V}$ is noted $s(v) \in \varepsilon$ (also noted as $\varepsilon(s) = v$), where v is the value carried by s. A signal that is absent in the execution environment is represented as $s(\perp)$.

The notation $\perp_{\varepsilon}$ designates an environment ε in which all signals are absent, i.e., an environment without any reaction.

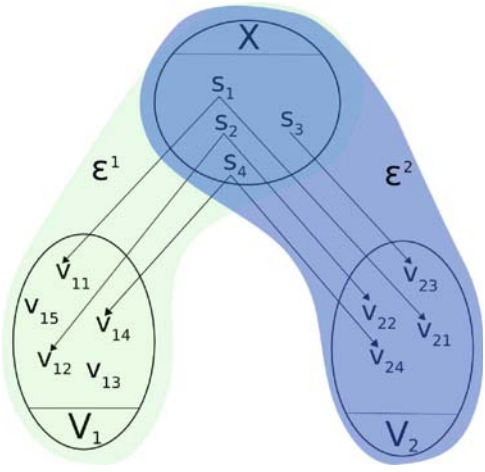


Fig. 7.1 Two environments $\varepsilon^1 : X \to V_1$ and $\varepsilon^2 : X \to V_2$

Definition 7.2 (Environment restriction). Let us consider an environment ε, and a subset of signals $X \subseteq \mathcal{X}$. The restriction noted $\varepsilon_{\bar{X}}$ is equivalent to ε where all signals $s \in X$ are no longer visible in the environment:

$$\varepsilon_{\bar{X}} : (\text{dom}(\varepsilon) \backslash X) \to \mathcal{V}^{\perp}, \text{ and } \forall x \in (\text{dom}(\varepsilon) \backslash X),\ \varepsilon(x) = \varepsilon_{\bar{X}}(x).$$

Figure 7.2 illustrates the restriction $\varepsilon_{\{\bar{s_2}\}}$ of an environment ε. As one can see, the signal variable s_2 is no longer visible in the right-hand-side environment.

Definition 7.3 (Environment composability). Let $\varepsilon^1 \in \varepsilon_{X_1}$ and $\varepsilon^2 \in \varepsilon_{X_2}$ denote two environments. They are composable iff for all $s \in X_1 \cap X_2$, we have $\varepsilon^1(s) = \varepsilon^2(s)$. Their composition, noted $\varepsilon^1 \oplus \varepsilon^2$, is therefore expressed as follows:

$$\begin{array}{rl} \oplus : \varepsilon_{X_1} \times \varepsilon_{X_2} & \to \varepsilon_{X_1 \cup X_2} \\ (\varepsilon^1, \varepsilon^2) & \mapsto \varepsilon^1 \cup \varepsilon^2. \end{array}$$

Figure 7.3 shows the composition of two environments ε_1 and ε_2. Here, the common signal variables hold the same values within both environments.

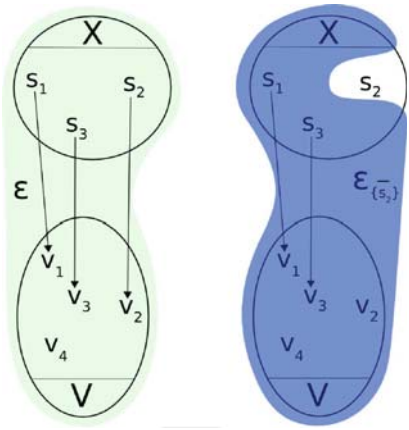


Fig. 7.2 An environment $\varepsilon : X \to V$ (*left*) and its restriction $\varepsilon_{\{\bar{s_2}\}}$ (*right*)

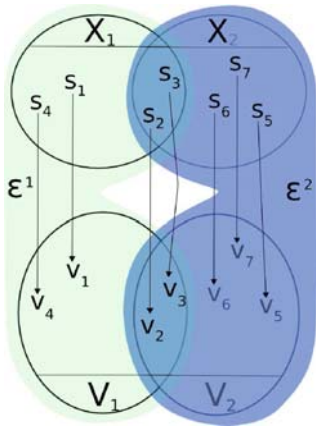


Fig. 7.3 Composition of two environments $\varepsilon^1 : X_1 \to V_1$ and $\varepsilon^2 : X_2 \to V_2$

The operational semantics of the SIGNAL language is specified through a labeled transition system where the states are SIGNAL programs. It is given in a notation à la Plotkin [4] as follows:

$$\frac{C}{p_1 \xrightarrow{\varepsilon} p_2},$$

where p_1 and p_2 represent processes, ε denotes an execution environment of the processes, and C is a *precondition* on p_1, p_2, and ε. ε indicates the status (presence or absence) and the value of the signals involved in p_1 and p_2 during the transition. The precondition C must be satisfied to perform a transition from p_1 to p_2 in an environment ε. The role of environments in such transitions is to define all possible value assignments for the signals involved in processes p_1 and p_2.

7.1.2 Primitive Constructs on Signals

Let p denote a process. It follows the *trivial rule* given below:

$$p \xrightarrow{\perp_\varepsilon} p.$$

The above rule denotes the environment in which all signals involved in a process p are absent. So, the process p never reacts. This rule holds for each primitive operator. It will therefore be voluntarily omitted in the sequel.

7.1.2.1 Instantaneous Relations/Functions

Semantics 1 (Relations/functions) *Let p denote* `s_n := R(s_1, ..., s_n-1)`. *The operational semantics of p is defined as follows:*

$$p \xrightarrow{s_1(v_1),\ldots,s_{n-1}(v_{n-1}),s_n(R(v_1,\ldots,v_{n-1}))} p.$$

The above rule specifies that when the signals $\mathtt{s}_1, \ldots, \mathtt{s}_{n-1}$, arguments of the relation R, are all present with the values $\mathtt{v}_1, \ldots, \mathtt{v}_{n-1}$ respectively, the defined signal $\mathtt{s}_n$ is present and holds the value given by the expression $R(v_1, \ldots, v_{n-1})$. This transition leaves p invariable because p is purely *combinatorial*, meaning that it is stateless.

In addition to this rule, the trivial transition rule holds for p when all signals involved are absent.

7.1.2.2 Delay

Semantics 2 (Delay) *Let p denote* `s_2 := s_1 $ 1 init c`. *The operational semantics of p is defined as follows:*

$$p \xrightarrow{s_1(v),s_2(c)} p',$$

where $p' =$ `s_2 := s_1 $ 1 init v`

In contrast to the instantaneous relation construct, the delay construct expresses a dynamic behavior where the value of the argument signal $\mathtt{s}_1$ is memorized after each transition. This is reason why p evolves towards p', which memorizes the previous value v taken by $\mathtt{s}_1$ during the transition, as the new initialization value for the next transition. The signal $\mathtt{s}_2$ hence carries the former initialization value c.

Besides the previous transition rule, the trivial rule also holds for the delay construct.

An alternative notation used later for the transition rule describing the semantics of the delay construct is as follows:

$$p\{c\} \xrightarrow{\mathtt{s}_1(v),\mathtt{s}_2(c)} p\{v\},$$

where $\{c\}$ and $\{v\}$ represent the values memorized in the different states of the process p. These values are those which are to be written in $\mathtt{s}_2$.

7.1.2.3 Undersampling

Semantics 3 (Undersampling) *Let p denote $\mathtt{s}_2$:= $\mathtt{s}_1$ when b. The operational semantics of p is defined as follows:*

$$\begin{array}{rl} p & \xrightarrow{s_1(v),b(\bot),s_2(\bot)} p, \\ p & \xrightarrow{s_1(\bot),b(v_b),s_2(\bot)} p, \\ p & \xrightarrow{s_1(v),b(true),s_2(v)} p, \\ p & \xrightarrow{s_1(v),b(false),s_2(\bot)} p. \end{array}$$

The semantics of the undersampling construct is straightforwardly expressed via four rules that distinguish the different value assignment scenarios for the signals involved. Here, p remains invariable after each transition for the same reason as the instantaneous relation construct.

The trivial rule holds for the undersampling construct.

7.1.2.4 Deterministic Merging

Semantics 4 (Merging) *Let p denote $\mathtt{s}_3$:= $\mathtt{s}_1$ default $\mathtt{s}_2$. The operational semantics of p is defined as follows:*

$$\begin{array}{rl} p & \xrightarrow{s_1(v_1),s_2(\bot),s_3(v_1)} p, \\ p & \xrightarrow{s_1(v_1),s_2(v_2),s_3(v_1)} p, \\ p & \xrightarrow{s_1(\bot),s_2(v_2),s_3(v_2)} p. \end{array}$$

The semantics of the deterministic merging is defined in a very similar way as that of the undersampling construct.

7.1.3 Primitive Constructs on Processes

7.1.3.1 Composition

Semantics 5 (Composition) *Let us consider four processes p_1, p_2, p_3, and p_4 within the environments ε^1 and ε^2. The operational semantics of the composition operation is expressed as follows:*

$$\frac{p_1 \xrightarrow{\varepsilon_1} p_2 \ ; \ p_3 \xrightarrow{\varepsilon_2} p_4 \ ; \ \varepsilon^1 \text{ and } \varepsilon^2 \text{ are composable}}{(p_1 \mid p_2) \xrightarrow{\varepsilon^1 \oplus \varepsilon^2} (p_3 \mid p_4)} .$$

In the above rule, p_1 is not necessarily different from p_2, e.g., when p_1 is a purely combinatorial process. The same remark is valid for p_3 and p_4.

The upper part of the rule specifies the condition under which the composition of two processes is possible: the transition environments ε^1 and ε^2 (applied to each individual process transition) must be composable. The lower part of the rule therefore describes the composition whenever this condition is satisfied.

7.1.3.2 Local Definition

Semantics 6 (Restriction) *Let us consider two processes p_1 and p_2 within an environment ε. The operational semantics of the restriction operation is expressed as follows:*

$$\frac{p_1 \xrightarrow{\varepsilon} p_2}{(p_1 \text{ where } s) \xrightarrow{\varepsilon_{\setminus\{s\}}} (p_2 \text{ where } s)} .$$

The semantics of the process restriction construct is simply defined by restricting the visibility of the signals concerned in the transition environment.

Example 7.1 (Operational semantics of a process). Let us consider the following process `P`:

```
1: process P {integer c;}
2:  (? integer reset;
3:   ! integer s1, s2, pre_s; )
4:  (| s1 := reset default s2
5:   | s2 := pre_s1 - 1
6:   | pre_s1 := s1 $ 1 init c
7:   |);
```

The previous semantic rules can be used to reason about the behavior of p as shown below.

The (nontrivial) operational interpretation of the different equations in the program `P` is as follows:

- Equation at `line 4`:

(a) `s1 := reset default s2` $\xrightarrow{\text{reset}(v_1),\text{s2}(\bot),\text{s1}(v_1)}$ `s1 := reset default s2`

(b) `s  := reset default s2` $\xrightarrow{\text{reset}(v_1),\text{s2}(v_2),\text{s1}(v_1)}$ `s1 := reset default s2`

(c) `s  := reset default s2` $\xrightarrow{\text{reset}(\bot),\text{s2}(v_2),\text{s1}(v_2)}$ `s1 := reset default s2`

- Equation at `line 5`:

$$\texttt{s2 := pre_s1 - 1} \xrightarrow{\text{pre_s1}(v_3),\text{s2}(v_3-1)} \texttt{s2 := pre_s1 - 1.}$$

- Equation at `line 6`:

$$\texttt{pre_s1 := s1 \$ 1 init c} \xrightarrow{\text{s1}(v_4),\text{pre_s1}(c)} \texttt{pre_s1 := s1 \$ 1 init v}_4\texttt{.}$$

From the transitions of the above statement, there are three possible combinations according to the semantics of composition to define the semantics of the program `P`. For each combination, let us check the associated required precondition.

- By considering interpretation (a) with those of the other statements, and according to the composition semantics, one obtains

$$[\text{reset}(v_1), \text{s2}(\bot), \text{s1}(v_1)] \wedge [\text{pre_s1}(v_3), \text{s2}(v_3 - 1)] \wedge [\text{s1}(v_4), \text{pre_s1}(c)].$$

  This precondition is never satisfied since $v_3 - 1$ carried by s2 is different from $\bot$ by definition! A similar observation is done when considering interpretation (a) with the trivial ones of the other statements:

$$[\text{reset}(v_1), \text{s2}(\bot), \text{s1}(v_1)] \wedge [\text{pre_s1}(\bot), \text{s2}(\bot)] \wedge [\text{s1}(\bot), \text{pre_s1}(\bot)].$$

- By considering interpretation (b) with those of the other statements, and according to the composition semantics, one now obtains

$$[\text{reset}(v_1), \text{s2}(v_2), \text{s1}(v_1)] \wedge [\text{pre_s1}(v_3), \text{s2}(v_3 - 1)] \wedge [\text{s1}(v_4), \text{pre_s1}(c)].$$

  From this precondition, the following equalities are deduced:

$$\begin{cases} v_3 = c, \\ v_2 = v_3 - 1, \\ v_1 = v_4. \end{cases}$$

  Hence, by substitution they lead to the following rule:

$$\texttt{P\{c\}} \xrightarrow{\text{reset}(v_1),\text{s1}(v_1),\text{pre_s1}(c),\text{s2}(c-1)} \texttt{P\{v}_4\texttt{\}.}$$

- By considering interpretation (c) with those of the other statements, and according to the composition semantics, one obtains

$$[\text{reset}(\bot), \text{s2}(v_2), \text{s1}(v_2)] \wedge [\text{pre_s1}(v_3), \text{s2}(v_3 - 1)] \wedge [\text{s1}(v_4), \text{pre_s1}(c)].$$

  In a similar way, this precondition leads to the following rule:

$$\texttt{P\{c\}} \xrightarrow{\text{reset}(\bot),\text{s1}(c-1),\text{pre_s1}(c),\text{s2}(c-1)} \texttt{P\{c-1\}.}$$

Through the above semantic interpretation of the program `P`, it can be concluded that `P` has two possible run scenarios: the one induced by interpretation (b) of the equation at `line 4` and the other induced by interpretation (c) of the equation at `line 4`.

7.2 A Denotational Semantics

The denotational semantics presented here is based on *tags* [3], which are elements of a partially ordered dense set.

There exists another denotational semantics of the language, which relies on infinite series referred to as *traces* [2]. The trace-based semantics mainly differs from the one presented here in that it adopts the following points of view: logical time is represented by a totally ordered set (the set of natural integers $\mathbb{N}$); absence of events is explicitly specified (by the $\perp$ symbol).

In the following, Sect. 7.2.1 defines the basic notions that are used afterwards to describe the semantics of the language primitive constructs in Sects. 7.2.2 and 7.2.3.

7.2.1 A Multiclocked Semantic Model

To define the multiclocked semantic model, the following sets are considered:

- $\mathcal{X}$ is a countable set of variables.
- $\mathcal{B} = \{ff, tt\}$ is a set of Boolean values where *ff* and *tt*, respectively, denote *false* and *true*.
- $\mathcal{V}$ is the domain of operands (at least $\mathcal{V} \supseteq \mathcal{B}$).
- $\mathbb{T}$ is a dense set equipped with a partial order relation $\leq$.

The elements of $\mathbb{T}$ are called *tags*. Each pair of tags $\{t_1, t_2\} \in \mathbb{T}^2$ admits a greatest lower bound noted $lwb\{t_1, t_2\}$ in $\mathbb{T}$.

We now introduce the notion of *observation point* (see Fig. 7.4).

Definition 7.4 (Observation points). A set of observation points is a set of tags $\mathcal{T}$ such that:

- $\mathcal{T} \subset \mathbb{T}$.
- $\mathcal{T}$ is countable.
- Each pair of tags admits a greatest lower bound in $\mathcal{T}$.

The set $\mathcal{T}$ provides a discrete time dimension that corresponds to logical instants at which the presence and absence of events can be observed during executions of a system. The set $\mathbb{T}$ provides a continuous time dimension (or physical time scale). So, the mapping of $\mathcal{T}$ on $\mathbb{T}$ allows one to move from "abstract" descriptions to "concrete" descriptions in the semantic model.

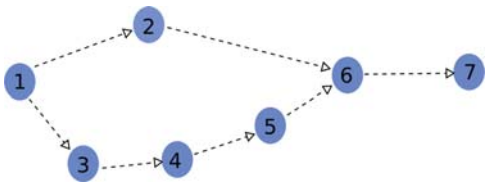


Fig. 7.4 A set of partially ordered observation points

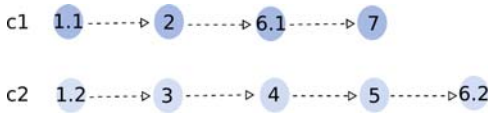


Fig. 7.5 Two chains resulting from the set shown in Fig. 7.4

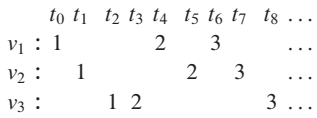


Fig. 7.6 Event, signal, and behavior

A *chain* $C \in \mathcal{T}$ is a totally ordered set which admits a greatest lower bound. The set of chains is denoted by $\mathcal{C}$. Figure 7.5 illustrates two chains.

For a set of observation points $\mathcal{T}$, we denote by $\mathcal{C}_\mathcal{T}$ the set of all chains in $\mathcal{T}$. The notation $\min(C)$ and $\text{pred}_C(t)$ means the minimum and the predecessor of a tag t in a chain C, respectively.

Definition 7.5 (Event). An **event** e on a given set of observation points $\mathcal{T}$ is a couple $(t, v) \in \mathcal{T} \times \mathcal{V}$.

Definition 7.6 (Signal). A **signal** on a given set of observation points $\mathcal{T}$ is a partial function $s \in C \rightharpoonup \mathcal{V}$ which associates values with observation points that belong to a chain $C \in \mathcal{C}_\mathcal{T}$.

The set of signals on $\mathcal{T}$ is noted $\mathcal{S}_\mathcal{T}$. The domain of s is denoted by $\text{tags}(s)$.

Definition 7.7 (Behavior). For a given set of observation points $\mathcal{T}$, a **behavior** b on $X \subseteq \mathcal{X}$ is a function $b \in X \to \mathcal{S}_\mathcal{T}$ that associates each variable $x \in X$ with a signal s on $\mathcal{T}$.

We denote by $\mathcal{B}_{\mathcal{T},X}$ the set of behaviors of domain $X \subseteq \mathcal{X}$ on $\mathcal{T}$. The set $\mathcal{B}_\mathcal{T}$ represents the set that contains all the behaviors defined on the union of all the sets of variables on $\mathcal{T}$. Finally, we write $\text{vars}(b)$ and $\text{tags}(b) = \bigcup_{x \in \text{vars}(b)} \text{tags}(b(x))$ to denote the domain of b and its associated set of tags, respectively.

Example 7.2. In the trace depicted in Fig. 7.6, we distinguish the following objects:

- Variables: $\{v_1, v_2, v_3\}$
- Values: $\{1, 2, 3\}$
- Tags: $\{t_0, t_1, t_2, t_3, t_4, t_5, t_6, t_7, t_8, \ldots\}$
- Events: $\{e_0 = (t_0, 1), e_1 = (t_1, 1), e_2 = (t_2, 1), e_3 = (t_3, 2), e_4 = (t_4, 2), e_5 = (t_5, 2), e_6 = (t_6, 3), e_7 = (t_7, 3), e_8 = (t_8, 3), \ldots\}$
- Signals: $\{s_1 = \{e_0, e_4, e_6, \ldots\}, s_2 = \{e_1, e_5, e_7, \ldots\}, s_3 = \{e_2, e_5, e_8, \ldots\}\}$
- Behavior: $\{(v_1, s_1), (v_2, s_2), (v_3, s_3)\}$

For any behavior b defined on $X \subset \mathcal{X}$, we denote by $b_{|X'}$ its projection on a set of variables $X' \subset X$, i.e., $\text{vars}(b_{|X'}) = X'$ and $\forall x \in X', b_{|X'}(x) = b(x)$. The projection of b on the complementary of X' in X is denoted by $b_{/X'}$. The empty signal is denoted by λ. Then, the particular behavior $0_{|X} = \{(x, \lambda) \mid x \in X\}$ expresses the association of $X \subset \mathcal{X}$ with the empty signal.

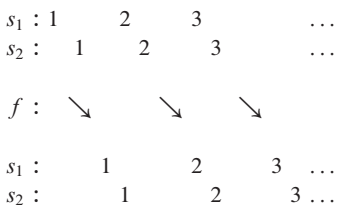


Fig. 7.7 Stretching of a behavior composed of two signals following f

In the polychronous model, the behaviors of a real-time system S are first specified on $\mathcal{T}$. Each instant $t \in \mathcal{T}$ denotes an occurrence of some events of S. Such a description fully takes into account the concurrency and precedence of events. At this stage, the functional properties of S can be easily checked (e.g., schedulability, safety).

On the other hand, one needs to deal with physical time, typically to be able to guarantee a correct deployment of polychronous descriptions on a target platform. This is obtained through a mapping from $\mathcal{T}$ to $\mathbb{T}$ based on the characteristics of the platform (e.g., processor speed, bus frequency).

After this observation, a central notion is now introduced to achieve the above mapping. The intuition is based on viewing a signal as an elastic with ordered marks on it (the tags). If this elastic is stretched (see Fig. 7.7), the marks will remain in the same order, but we may now add more marks between stretched marks. If the elastic is contracted, the marks will be closer to one another, but they will still remain in the same order. The same holds for a set of elastics: a behavior.

Definition 7.8 (Stretching). For a given set of observation points $\mathcal{T}$, a behavior b_1 is less stretched than another behavior b_2, noted $b_1 \leq_{\mathcal{B}_\mathcal{T}} b_2$, iff there exists a bijection $f : \text{tags}(b_1) \to \text{tags}(b_2)$ following b_1 and b_2 are isomorphic:

- $\forall x \in \text{vars}(b_1),\ \ f(\text{tags}(b_1(x))) = \text{tags}(b_2(x))$
- $\forall x \in \text{vars}(b_1)\ \forall t \in \text{tags}(b_1(x)),\ \ b_1(x)(t) = b_2(x)(f(t))$
- $\forall t_1, t_2 \in \text{tags}(b_1),\ \ t_1 \leq t_2 \Leftrightarrow f(t_1) \leq f(t_2)$

and such that

$$\forall C \in \mathcal{C}_\mathcal{T},\ \ \forall t \in C\ t \leq f(t).$$

The stretching relation is a partial order on $\mathcal{B}_\mathcal{T}$. It gives rise to an equivalence relation between behaviors.

Definition 7.9 (Stretch equivalence). For a given set of observation points $\mathcal{T}$, two behaviors b_1 and b_2 are stretch-equivalent, noted $b_1 \lesssim b_2$, iff there exists another behavior b_3 less stretched than both b_1 and b_2, i.e., $b_1 \lesssim b_2$ iff

$$\exists b_3\ b_3 \leq_{\mathcal{B}_\mathcal{T}} b_1\ \ and\ \ b_3 \leq_{\mathcal{B}_\mathcal{T}} b_2.$$

The stretching (being less stretch than) preorder induces a semilattice, which admits a minimal element. We call *strict* behaviors those which are minimal for the

stretch relation on $\mathcal{T}$. For a given behavior b, the set of all behaviors that are stretch-equivalent to b on $\mathcal{T}$ defines its *stretch closure* on $\mathcal{T}$, noted b^*. This notion allows us to define *processes*.

Definition 7.10 (Stretch closure of a set of behaviors). The stretch closure of a set of behaviors p on a given set $\mathcal{T}$ of observation points is the set denoted by p^*, which includes all the behaviors resulting from the stretch closure of each behavior $b \in p$, i.e., $p^* = \bigcup_{b \in p} b^*$.

Definition 7.11 (Process). For a given set of observation points $\mathcal{T}$, a process is a stretch-closed set of behaviors $p \in \mathcal{P}(\mathcal{B})$, i.e., $p = p^*$.

We write $\text{vars}(p)$ to denote the set of variables of a process p. Equivalently, p is said to be defined on $\text{vars}(p)$. Every nonempty process contains a subset $p_\downarrow \subseteq p$ of strict behaviors (for each $b_1 \in p$, there exists a unique $b_2 \in p_\downarrow$ such that $b_2 \leqslant b_1$). In other words, strict behaviors are those which are minimal with respect to stretching on a set $\mathcal{T}$ of tags.

7.2.2 Primitive Constructs on Signals

For each primitive construct of the language, we express its denotational semantics in terms of sets of behaviors. Given a process p, we denote by $[\![p]\!]$ the set of all possible behaviors of p.

7.2.2.1 Instantaneous Relations/Functions

Semantics 7 (Relations/functions) *The denotational semantics of instantaneous relations/functions is defined as follows:*

$$[\![s_n := R(s_1, \ldots, s_{n-1})]\!] =$$
$$\{b \in \mathcal{B}_{|s_1,\ldots,s_n} \mid \text{tags}(b(s_1)) = \cdots = \text{tags}(b(s_n)) = C \in \mathcal{C}\backslash\emptyset$$
$$\forall t \in C, b(s_n)(t) = [\![R]\!](b(s_1)(t), \ldots, b(s_{n-1})(t))\}.$$

The denotational semantics of instantaneous relations is the set of behaviors b such that:

- The tags corresponding to each signal s_k involved in b represent the same chain C of tags.
- For each tag in C, the relation R holds between the values carried by the signals involved. Here, $[\![R]\!]$ denotes the pointwise interpretation of R.

7.2.2.2 Delay

Semantics 8 (Delay) *The denotational semantics of the delay construct is defined as follows:*

$$
\begin{aligned}
&[\![s_2 \texttt{ := } s_1 \texttt{ \$ 1 init } c]\!] = \\
&\{0_{|s_1,s_2}\} \cup \\
&\{b \in \mathcal{B}_{|s_1,s_2} \mid \quad \text{tags}(b(s_2)) = \text{tags}(b(s_1)) = C \in \mathcal{C}\backslash\emptyset, \\
&b(s_2)(\min(C)) = c, \\
&\qquad\qquad \forall t \in C\backslash\min(C), b(s_2)(t) = b(s_1)(\text{pred}_C(t))\}.
\end{aligned}
$$

The denotational semantics of the delay construct is the set of behaviors b such that:

- The tags corresponding to each signal $\mathtt{s}_k$ involved in b represent the same chain C of tags.
- At the initial tag of C, s_2 holds the value c in such behaviors.
- For all the other tags $t \in C$, the value taken by s_2 at t is the value carried by s_1 at the predecessor of t in C.

7.2.2.3 Undersampling

Semantics 9 (Undersampling) *The denotational semantics of the undersampling construct is defined as follows:*

$$
\begin{aligned}
&[\![s_3 \texttt{ := } s_1 \texttt{ when } s_2]\!] = \\
&\{b \in \mathcal{B}_{|s_1,s_2,s_3} \mid \text{tags}(b(s_3)) = \{t \in \text{tags}(b(s_1)) \cap \text{tags}(b(s_2)) | b(s_2)(t) = tt\} = C \\
&\qquad\qquad \in \mathcal{C} \forall t \in C, b(s_3)(t) = b(s_1)(t)\}.
\end{aligned}
$$

The set of behaviors resulting from the denotational interpretation of the undersampling construct is such that:

- The set of tags corresponding to s_3 is the intersection of the set of tags associated with s_1 and the set of tags at which s_2 carries the value *true* .
- At each tag of s_3, the value held by s_3 is that of s_1 in the behavior.

7.2.2.4 Deterministic Merging

Semantics 10 (Merging) *The denotational semantics of the deterministic merging construct is defined as follows:*

$$
\begin{aligned}
&[\![s_3 \texttt{ := } s_1 \texttt{ default } s_2]\!] = \\
&\{b \in \mathcal{B}_{|s_1,s_2,s_3} \mid \text{tags}(b(s_3)) = \text{tags}(b(s_1)) \cup \text{tags}(b(s_2)) = C \in \mathcal{C} \\
&\qquad\qquad \forall t \in C, b(s_3)(t) = b(s_1)(t) \text{ if } t \in \text{tags}(b(s_1)) \text{ else } b(s_2)(t)\}
\end{aligned}
$$

The denotation of the deterministic merging construct is such that:

- The set of tags corresponding to s_3 is the union of those associated with s_1 and s_2.
- The value taken by s_3 is that of s_1 at any tag t common to s_1 and s_3; otherwise, s_3 takes the value carried by s_2 at its tags, which do not belong to the tags of s_1.

7.2.3 Primitive Constructs on Processes

7.2.3.1 Composition

Semantics 11 (Composition) *For a given set of observation points $\mathcal{T}$, the composition on $\mathcal{T}$ of processes p_1 and p_2 is a process p such that* $\text{vars}(p) = \text{vars}(p_1) \cup \text{vars}(p_2)$*:*

$$[\![\texttt{p = p1 | p2}]\!] = (\ \{b \mid b_{|X_1} \in p_1, b_{|X_2} \in p_2\}\ )^*.$$

The composition of two processes p_1 and p_2 is the stretch closure of the behaviors b such that the projection of these behaviors on the respective sets of variables associated with p_1 and p_2 yields behaviors that belong to p_1 and p_2, respectively.

7.2.3.2 Local Definition

Semantics 12 (Restriction) *The restriction, noted p where s, of a process p defined on X to a process defined on $X \backslash \{s\}$ is defined as the following set of behaviors:*

$$[\![\texttt{p where s}]\!] = (\ \{b_2 \mid \exists b_1 \in p \ \wedge\ b_2 = b_{1/\{s\}}\}\ )^*.$$

The restriction of a process p to a signal s is the stretch closure of the behaviors characterizing p projected on the set of variables associated with p, which is the complementary set of $\{s\}$.

7.3 Exercises

7.1. Propose both operational and denotational interpretations for the semantics of the following statements:

- Clock extraction `(| clk := ^s |)`
- Clock extraction from a Boolean signal: `(| clk := when c |)`
- Clock intersection `(| clk :=` s_1 `^*` s_2 `|)`
- Clock union: `(| clk :=` s_1 `^+` s_2 `|)`

7.2. Give an operational interpretation for the semantics of the following processes:

- Process `P1`

```
1: process P1 =
2:    { integer N }
3:    ( ? integer s1;
4:      ! boolean cond; integer s2)
5:    (| s2 := N * s1
6:     | cond := s1 > s2
7:     |);%process P1%
```

- Process P2

```
1: process P2 =
2:     ( ? boolean b1; integer s1;
3:       ! boolean b4;)
4:     (| b2 := when b1
5:      | s2 := s1 $ 1 init 0
6:      | b3 := s2 < 5
7:      | b4 := b2 default b3
8:      |)
9:   where
10:     event b2; boolean b3; integer s2;
11: end;%process P2%
```

Expected Learning Outcomes:

- The **operational semantics** of SIGNAL is defined in terms of **transition systems** where the states consist of processes. A transition is fired upon the occurrence of the events manipulated by these processes.
- The **denotational semantics** presented in this chapter relies on the **tagged signal model** of Lee and Sangiovanni-Vincentelli. It is considered nowadays as the reference semantic model of polychrony.

References

1. Benveniste A, Le Guernic P, Jacquemot C (1991) Synchronous programming with events and relations: the SIGNAL language and its semantics. Science of Computer Programming, Elsevier North-Holland, Inc, Amsterdam, The Netherlands 16(2):103–149
2. Le Guernic P, Gautier T (1991) Data-flow to von Neumann: the SIGNAL Approach. In: Gaudiot J-L, Bic L (eds) Advanced Topics in Data-Flow Computing, Prentice-Hall, Englewood Cliffs, NJ, pp 413–438
3. Le Guernic P, Talpin J-P, Le Lann J-C (2003) Polychrony for system design. Journal for Circuits, Systems and Computers 12(3):261–304
4. Plotkin GD (1981) A Structural Approach to Operational Semantics. Technical Report DAIMI FN-19, University of Aarhus, Denmark
5. Plotkin GD (2004) The origins of structural operational semantics. Journal of Logic and Algebraic Programming, Elsevier, (60-61):3–15 (preprint)
6. Scott D, Strachey C (1971) Toward a mathematical semantics for computer languages. In: Proceedings of the Symposium on Computers and Automata, Polytechnic Institute of Brooklyn, pp 19–46

Chapter 8
Formal Model for Program Analysis

Abstract This chapter presents the main intermediate representations that serve for the analysis of clocks and data dependencies in programs. A SIGNAL program is a formal specification that is basically composed of equations describing relations for both values and clocks of the signals involved. This essence of the language allows one to mathematically reason on the properties of such a specification. The reasoning framework allowed by SIGNAL is the algebraic domain $\mathbb{Z}/3\mathbb{Z}$, the set of integers modulo 3. The intrinsic properties of this domain typically enable one to deal efficiently with clock properties in SIGNAL programs. Section 8.1 presents the encoding of SIGNAL constructs in $\mathbb{Z}/3\mathbb{Z}$ (Le Guernic and Gautier in *Advanced Topics in Data-Flow Computing*, Prentice Hall, pp. 413–438, 1991). Then, Sect. 8.2 deals with the representation of the data dependencies expressed by SIGNAL programs using a specific directed dependency graph.

8.1 The Synchronization Space: $\mathcal{F}_3$

The purpose of the algebraic encoding of the synchronization relations is twofold:

1. The detection of synchronization error.
2. The deduction of control hierarchy associated with the program based on an order relation defined on clocks as follows: a clock c_1 is said to be greater than a clock c_2, which is denoted by $c_1 \geq c_2$, if the set of instants of c_2 is included in the set of instants of c_1 (i.e., c_2 is an undersampling of c_1); the set of clocks with this relation is a lattice.

This section first addresses the $\mathbb{Z}/3\mathbb{Z}$ encoding of signal clocks and values (Sect. 8.1.1). Then, it presents how the primitive constructs of SIGNAL are encoded (Sect. 8.1.2), as well as some extended constructs (Sect. 8.1.3). Finally, a general form of encoded programs is given in Sect. 8.1.4.

A. Gamatié, *Designing Embedded Systems with the SIGNAL Programming Language: Synchronous, Reactive Specification*, DOI 10.1007/978-1-4419-0941-1_8,
© Springer Science+Business Media, LLC 2010

8.1.1 Encoding Abstract Clocks and Values

The coding space for signal clocks and values consists of the finite field $\mathcal{F}_3$ of integers modulo 3: $\mathbb{Z}/3\mathbb{Z} = \{-1, 0, 1\}$. Such a set holds very interesting properties that are used to reason about the properties of encoded statements. For instance, let x be a variable defined on $\mathcal{F}_3$ and $n \in \mathbb{N}$, then:

1. $x^{2n} = x^2$
2. $x^{2n+1} = x$
3. $x + x = -x$
4. $\forall x \neq 0, 1/x = x$
5. $x(1 - x^2) = 0$
6. $(f(x^2))^{2n} = f(x^2)$

Now, let us consider a simple example of a SIGNAL statement consisting of the expression

$$\texttt{s}_3 \texttt{ := } \texttt{s}_1 \texttt{ when } \texttt{s}_2,$$

where signals $\texttt{s}_1$ and $\texttt{s}_3$ are of the same type and $\texttt{s}_2$ is of Boolean type. According to the semantics of the undersampling operator, the following assertions hold:

- If $\texttt{s}_1$ is present (i.e., $\texttt{s}_1$ is defined), and $\texttt{s}_2$ is present and carries the value *true*, then $\texttt{s}_3$ is present and $\texttt{s}_1 = \texttt{s}_3$.
- If $\texttt{s}_1$ is absent (i.e., $\texttt{s}_1$ is undefined), or $\texttt{s}_2$ is absent, or $\texttt{s}_2$ is present and carries the value *false*, then $\texttt{s}_3$ is absent (i.e., not defined).

From the above assertions, we can observe that useful information about the signals is, on the one hand, the value of $\texttt{s}_2$ (*true* and *false*) and, on the other hand, the status of all signals involved (presence and absence). This information represents the basic two notions that are taken into account in the algebraic encoding of the synchronization relations in $\mathcal{F}_3$:

$$\textit{true} \to +1, \ \textit{false} \to -1, \ \text{absent} \to 0, \ \text{present} \to \pm 1.$$

A consequence is that if s is the encoding of a signal $\texttt{s}$, the presence of $\texttt{s}$ is clearly represented by s^2. Typically, s^2 may be considered as the proper clock of signal $\texttt{s}$. Notice that the above encoding fully takes into account the value of Boolean signals. For non-Boolean signals, it only expresses their status and not their values.

8.1.2 Encoding Primitive Constructs

We give the $\mathcal{F}_3$ encoding of the primitive constructs of SIGNAL.

8.1.2.1 Instantaneous Relations/Functions

`s`$_n$ `:= R(s`$_1$`,...,s`$_{n-1}$`)`. The encoding in $\mathcal{F}_3$ of such a statement for any type of signals involved is as follows:

$$s_n{}^2 = s_1{}^2 = \ldots = s_{n-1}{}^2.$$

The above equalities only express constraints on the status of signals `s`$_n$, `s`$_1$, and `s`$_{n-1}$, meaning that they have the same clock. When these signals are of Boolean type, a constraint can also be specified on their values in addition to their status, as shown in the next example.

Example 8.1. Let us consider the statements `s`$_3$ `:= s`$_1$ `or s`$_2$ and `s`$_5$ `:= not s`$_4$. Their value relations are, respectively, encoded in $\mathcal{F}_3$ by

$$s_3 = s_1 s_2 (1 - s_1 - s_2 - s_1 s_2) \text{ and } s_5 = -s_4.$$

8.1.2.2 Delay

`s`$_2$ `:= s`$_1$ `$ 1 init c`. The encoding in $\mathcal{F}_3$ of this statement for any type of signals involved is similar to that of instantaneous relations/functions:

$$s_2{}^2 = s_1{}^2.$$

If `s`$_2$ and `s`$_1$ are of Boolean type, we can deduce the following encoding:

$$\begin{aligned} \xi_{n+1} &= (1 - s_1{}^2)\xi_n + s_1, \xi_0 = c, \\ s_2 &= s_1{}^2 \xi_n, \end{aligned}$$

where ξ_n denotes the current state, ξ_{n+1} is its next state according to any (hidden) clock that is more frequent than the clock of `s`$_1$, and ξ_0 is the initial state.

8.1.2.3 Undersampling

`s`$_2$ `:= s`$_1$ `when b`. For non-Boolean signals, the encoding of clocks in $\mathcal{F}_3$ of this statement is as follows:

$$s_2{}^2 = s_1{}^2(-b - b^2).$$

For Boolean signals, we get the following encoding:

$$s_2 = s_1(-b - b^2).$$

This equality may be interpreted as follows: s_2 holds the same value as s_1 ($s_2 = s_1$) when b is *true* (i.e., when the following holds: $-b - b^2 = 1$).

8.1.2.4 Deterministic Merging

`s3 := s1 default s2`. For non-Boolean signals, the encoding of clocks in $\mathcal{F}_3$ of this statement is as follows:

$$s_3{}^2 = s_1{}^2 + s_2{}^2(1 - s_1{}^2).$$

For Boolean signals, we get the following encoding:

$$s_3 = s_1 + s_2(1 - s_1{}^2).$$

This equality may be interpreted as follows: s_3 has a value when s_1 is defined, i.e., when $s_1^2 = 1$ (then s_3 holds the same value as s_1: $s_3 = s_1^2 s_1 = s_1$), or when s_2 is defined but s_1 is not, i.e., when $(1 - s_1^2)s_2^2 = 1$ (then s_3 holds the same value as s_2: $s_3 = (1 - s_1^2)s_2^2 s_2 = (1 - s_1^2)s_2$).

8.1.2.5 Composition

`P | Q`. The encoding of a composition `P | Q` of processes in $\mathcal{F}_3$ is the union of the encodings of its composing processes `P` and `Q`.

8.1.2.6 Local Definition

`P where s`. Local definitions do not really affect the encoding in $\mathcal{F}_3$ except for the visibility restriction in process definition.

8.1.3 Encoding Some Extended Constructs

On the basis of the encoding given in the previous sections, we illustrate how the encoding of extended constructs can be derived. Here, we consider the *memorization* construct: `s2 := s1 cell b init c`, which is equivalent to

```
(|s2 := s1 default (s2$1 init c) | y ^= x ^+ (when b) |)
```

using the primitive operators of SIGNAL. Let us consider this equivalent form to define the encoding in $\mathcal{F}_3$. In addition, for the sake of clarity, let us decompose the first equation in the composition as follows:

```
(|s2 := s1 default z | z := s2$1 init c |).
```

- **Case 1**: $\mathtt{s}_1$ and $\mathtt{s}_2$ are of Boolean type.
  To determine the value of Boolean signal $\mathtt{s}_2$, we apply the definitions for each construct to obtain the following system:

$$s_2 = s_1 + (1 - {s_1}^2)z,$$
$$z = {s_2}^2\xi_n, \quad \xi_{n+1} = s_2 + (1 - {s_2}^2)\xi_n, \quad \xi_0 = c,$$
$$ {s_2}^2 = (s_1 + (1 - {s_1}^2)(-b - b^2))^2,$$

  which is transformed into

$$s_2 = s_1 + (1 - {s_1}^2){s_2}^2\xi_n,$$
$$\xi_{n+1} = s_2 + (1 - {s_2}^2)\xi_n, \quad \xi_0 = c,$$
$$ {s_2}^2 = (s_1 + (1 - {s_1}^2)(-b - b^2))^2.$$

  In the third equation of the system, we notice that $(-b - b^2)^2 = (-b - b^2)$, since $(-b - b^2)$ equals 1 when $b = 1$, otherwise it equals 0. Using the properties of $\mathcal{F}_3$ mentioned in Sect. 8.1.1, we can simplify this equation as follows:

$$ {s_2}^2 = {s_1}^2 + (1 - {s_1}^2)(-b - b^2).$$

  The clock of $\mathtt{s}_2$ is the union of the clocks of $\mathtt{s}_1$ and `when b`. By replacing ${s_2}^2$ in the first equation, we obtain

$$s_2 = s_1 + (1 - {s_1}^2)(-b - b^2)\xi_n.$$

  Now, let us replace s_2 and ${s_2}^2$ in the second equality. We then have

$$\xi_{n+1} = s_1 + (1 - {s_1}^2)\xi_n, \quad \xi_0 = c.$$

  The variable ξ_n memorizes the value of signal $\mathtt{s}_1$ whenever $\mathtt{s}_1$ is present. One can see that $\mathtt{s}_2$ equals $\mathtt{s}_1$ whenever $\mathtt{s}_1$ is present (${s_1}^2 = 1$). Otherwise, whenever $s_1 = 0$ and $b = 1$, $\mathtt{s}_2$ equals ξ_n (i.e., the latest value of $\mathtt{s}_1$). Whenever $s_1 = 0$ and $b = 0$ or $b = -1$, we have $s_2 = 0$ (i.e., absent). This is actually the semantics of the *memorization* construct.
- **Case 2**: $\mathtt{s}_1$ and $\mathtt{s}_2$ are of non -Boolean type.
  We can only reason on the clocks of signals involved in this case. The second statement in the equivalent expression above for the *memorization* consisting of the *synchronization* is encoded as

$$ {s_2}^2 = {s_1}^2 + (-b - b^2)^2 - a^2(-b - b^2)^2.$$

  Then, using the properties already mentioned, we can transform this equation into

$$ {s_2}^2 = {s_1}^2 + (1 - a^2)(-b - b^2),$$

  which is the same expression as in the Boolean case.

The first statement combines *deterministic merging* and *delay* operators. Then, on the basis of the above decomposition, we have

$$s_2{}^2 = s_1{}^2 + z^2 - a^2 z^2 \quad \text{and} \quad z^2 = s_2{}^2 \quad \Rightarrow \quad s_2{}^2 = s_1{}^2 + (1 - s_1{}^2) s_2{}^2.$$

By substituting s_2 on the right-hand side of this equation with its value above resulting from the *synchronization* construct, we obtain

$$s_2{}^2 = s_1{}^2 + (1 - s_1{}^2)(s_1{}^2 + (1 - a^2)(-b - b^2)).$$

Since $(1 - s_1{}^2)s_1{}^2 = 0$ and $(1 - s_1{}^2)^2 = 1 - s_1{}^2$, we deduce the above expression:

$$s_2{}^2 = s_1{}^2 + (1 - s_1{}^2)(-b - b^2).$$

This shows that the system is coherent!

The flower clock of the Jardin Anglais (Geneva, Switzerland), 2009

8.1.4 General Form of an Encoded Program

The general form of an encoded SIGNAL program consists of a system of polynomial dynamical equations over the set $\mathbb{Z}/3\mathbb{Z}$. Such a representation offers adequate reasoning support. In particular, it is considered in the SIGALI tool [1] for the symbolic model checking of SIGNAL programs.

The composition of the basic $\mathbb{Z}/3\mathbb{Z}$ equations, presented in the previous sections, yields the following system of polynomial dynamical equations:

$$\begin{cases} X_{t+1} = P(X_t, Y_t) & (8.1) \\ Q(X_t, Y_t) = 0 & (8.2) \\ Q_0(X_t) = 0 & (8.3) \end{cases}$$

where $X_{t,t\in\mathbb{N}}$ is a vector of n states variables, i.e., $X_{t,t\in\mathbb{N}} \in (\mathbb{Z}/3\mathbb{Z})^n$, and $Y_{t,t\in\mathbb{N}}$ is a vector of m event variables, i.e., $Y_{t,t\in\mathbb{N}} \in (\mathbb{Z}/3\mathbb{Z})^m$. The different equations of the system have the following meaning:

- Equation (8.1) describes the evolution of state variables according to the logical time (represented by the index $t \in \mathbb{N}$) in the system. It therefore reflects the dynamical aspect of the system. Finally, it consists of a vectorial function from $(\mathbb{Z}/3\mathbb{Z})^{n+m}$ to $(\mathbb{Z}/3\mathbb{Z})^n$.
- Equation (8.2) denotes the static constraints, which are also considered as invariant properties of the system. It consists of a set of equations that express clock properties.
- Equation (8.3) represents the initialization of all state variables $X_{t,t\in\mathbb{N}}$. It is composed of n equations.

The automata-based interpretation of the above polynomial dynamical system for symbolic model checking in SIGALI can be seen briefly as follows: the initial states of the associated automaton are the solutions of the initialization equation (8.3); for any current state $X_{t,t\in\mathbb{N}} \in (\mathbb{Z}/3\mathbb{Z})^n$ and any event $Y_{t,t\in\mathbb{N}} \in (\mathbb{Z}/3\mathbb{Z})^m$ that satisfy the constraints encoded by (8.2) the system can perform a transition of the automaton towards a new state X_{t+1} obtained through (8.1).

Typical properties [1] such as liveness, reachability, and invariance can therefore be verified on the basis of the $\mathbb{Z}/3\mathbb{Z}$ encoding form. For instance, for liveness verification, which amounts to checking that the system cannot reach a state from which no transition can occur, the following formulation is considered: a state X_t is *alive* iff there exists an event Y_t such that $Q(X_t, Y_t) = 0$; it follows that a set of states is alive iff each of its states is alive.

8.2 Conditional Dependency Graph

The encoding in $\mathcal{F}_3$ enables one to deal with clock relations when the signals involved in programs are of non-Boolean type. The reasoning approach for SIGNAL programs also includes *dependency graphs* to encode data dependencies. These graphs are needed to detect cyclic definitions, e.g., in a statement such as `s := s + 1`, and to generate executable specifications.

A classical dataflow graph is not expressive enough to represent the data dependencies of SIGNAL programs. Since these programs handle signals whose clocks may be different, the dependencies are not static and depend on these clocks.

For that reason, the graph has to express *conditional* dependencies, where the conditions are nothing but the clocks at which dependencies are effective. Moreover, in addition to dependencies between signals, the following relation has to be considered: for any signal `s`, the values of `s` cannot be known before its clock; in other words, `s` depends on $\mathtt{s}^2$, noted: $\mathtt{s}^2 \xrightarrow{\mathtt{s}^2} \mathtt{s}$. This relation is assumed to be implicit in the sequel.

The *conditional dependency graph* associated with a given SIGNAL program consists of a labeled directed graph where:

- Vertices are the signals, plus clock variables.
- Arcs represent dependence relations.
- Labels are polynomials on $\mathcal{F}_3$ which represent the clocks at which the relations are valid.

Section 8.2.1 describes the data dependencies inferred from the primitive constructs of SIGNAL. An illustration is given in Sect. 8.2.2 for the way these dependencies are exploited during the analysis of a program to check the presence of dependency cycles.

8.2.1 Dependencies in Primitive Constructs

We describe the conditional dependencies associated with each SIGNAL primitive construct.

8.2.1.1 Instantaneous Relations/Functions

s_n `:=` `R(`s_1`,...,`s_{n-1}`)`. The dependency associated with this statement is as follows:

$$s_1 \xrightarrow{s_n^2} s_n, \quad \ldots, \quad s_{n-1} \xrightarrow{s_n^2} s_n.$$

8.2.1.2 Delay

s_2 `:=` s_1 `$ 1 init c`. This construct does not induce any dependency between the signals involved (nevertheless, remember that every signal depends on its proper clock).

8.2.1.3 Undersampling

s_2 `:=` s_1 `when b`. The dependency associated with this statement is as follows:

$$s_1 \xrightarrow{s_2^2} s_2.$$

8.2.1.4 Deterministic Merging

s_3 `:=` s_1 `default` s_2. The dependency associated with this statement is as follows:

$$s_1 \xrightarrow{s_1^2} s_3, \qquad s_2 \xrightarrow{(1-s_1^2)s_2^2} s_3.$$

8.2.1.5 Composition

`P | Q`. The graph of a composition `P | Q` of processes is the union of the graphs of its composing processes `P` and `Q`.

8.2.1.6 Local Definition

`P where s`. Local definitions do not affect the graph definition for `P`.

Given a program `P`, its corresponding dependency graph and clock information are used to generate the definition of each signal of `P`. So, they play a central role regarding the efficiency of the code generation process in SIGNAL.

8.2.2 Example: Checking Dependency Cycles

The presence of a dependency cycle in a program is a synonym of deadlock, i.e., instantaneous self-dependency. During the compilation of synchronous languages, all programs affected by this issue are considered as incorrect.

During the code generation process (which includes the compilation phase), the SIGNAL compiler can detect the presence of a cycle in the dependency graph, as illustrated in the example presented below. In that case, no code can be generated for such a program.

Example 8.2 (Dependency cycle in a program). Let us consider the following program, called `Cyclic_P`:

```
1:  process Cyclic_P =
2:    ( ? integer i;
3:      ! integer o; )
4:    (| s1 := i+s2
5:     | s2 := 2*s1
6:     | o := s1+1
7:     |)
8:    where
9:      integer s1, s2;
10:  end;
```

In this program, one can trivially observe through the equations at lines `4` and `5` that both signals `s1` and `s2` depend on each other. As a result, the analysis of the dependency graph associated with `Cyclic_P` yields a cycle.

Such an issue is reported by the compiler in a specific diagnostic file, named by convention `programName_CYC.SIG`, which is generated automatically when invoking any code generation option, e.g., `-c` for C code, `-c++` for C++ code, or `-java` for Java code (see Chap. A for a description of these options). This new

option enables one to transform the analyzed program such that an order is found according to which all operations are scheduled and sequential code can be generated directly. If the conditional dependency graph of the program actually contains a cycle, then the dependencies concerned are indicated in a generated diagnostic file.

Here, the content of the `Cyclic_P_CYC.SIG` file obtained is as follows:

```
1:  process Cyclic_P_CYC =
2:      ( )
3:    (| (| s1 := i+s2
4:       | s2 := 2*s1
5:       |)
6:     | (| s1 --> s2
7:       | s2 --> s1
8:       |)
9:     |)
10: %Cyclic_P_CYC%;
```

The compiler isolates in `Cyclic_P_CYC` only the definitions concerned with the cycle from the original program. These definitions are indicated in the first part of the generated program (i.e., from line `3` to line `5`). The other part of the program explicitly describes how the dependency cycle occurs from these definitions: line `6` expresses that the value of `s2` depends on that of `s1`, whereas the reverse is expressed by line `7`.

We have to notice that in some conditional dependency graphs there could be so-called *pseudo* cycles, meaning that the dependencies involved in such cycles are never valid at the same time during execution. Thus, the apparent cycles observed statically never become effective since their labeling clock expressions are never satisfied at the same time.

Since clocks are encoded in $\mathcal{F}_3$, this information can be used to check actual cycles: they are such that the product of the labels of their arcs is not null. This may be compared with the cycle sum test of [3] to detect deadlock on the dependency graph of a dataflow program.

In the next chapter, we show how the SIGNAL encoding in $\mathcal{F}_3$ is used together with the conditional dependency graph by the compiler to answer fundamental questions about the properties of a given program, and to generate automatically a corresponding executable code.

8.3 Exercises

8.1. Give the encoding in $\mathcal{F}_3$ of the statement: `clk := ^s`.

8.2. Let $\mathtt{b}_1$ and $\mathtt{b}_2$ be two Boolean signals. Propose a $\mathbb{Z}/3\mathbb{Z}$ encoding of the statement: $\mathtt{b} := \mathtt{b}_1 \neq \mathtt{b}_2$.

Expected Learning Outcomes:

- The analysis of SIGNAL programs relies on their **encoding in the algebraic domain** $\mathbb{Z}/3\mathbb{Z} = \{-1, 0, 1\}$, the set of integers modulo 3. This domain has interesting algebraic properties that enable one to suitably address those of SIGNAL programs.
- The basic encoding scheme is as follows:
  - For Boolean signals: -1, 0, and 1 mean false, absent, and true, respectively.
  - For non-Boolean signals: 0 and 1 mean absent and present, respectively.
- The general form of an encoded program is a system of polynomial dynamical equations over $\mathbb{Z}/3\mathbb{Z}$.
- In SIGNAL, **data dependencies** are captured via a **graph representation**. In such a graph, edges are labeled with clock expressions, which indicate when the corresponding data dependency is active. This graph is called a **conditional dependency graph**.
- The $\mathbb{Z}/3\mathbb{Z}$ encoding and the conditional dependency graph are the key intermediate representations that are used to analyze SIGNAL programs during the compilation phase.

References

1. Le Borgne M, Marchand H, Rutten É, Samaan M (2001) Formal verification of SIGNAL programs: Application to a Power Transformer Station Controller. Science of Computer Programming 41(1):85–104
2. Le Guernic P, Gautier T (1991) Data-flow to von Neumann: the SIGNAL approach. In: Gaudiot J-L, Bic L (eds) Advanced Topics in Data-Flow Computing. Prentice-Hall, Englewood Cliffs, NJ, pp 413–438
3. Wadge WW (1979) An Extensional treatment of dataflow deadlock. In: Kahn G (ed) Semantics of Concurrent Computation, LNCS vol 70. Springer-Verlag, London, UK, pp 285–299

Chapter 9
Compilation of Programs

Abstract This chapter focuses on the combination of the $\mathbb{Z}/3\mathbb{Z}$ encoding and the conditional dependency graph, presented in Chap. 8, to analyze programs during the compilation phase, referred to as *clock calculus*, and to automatically generate a corresponding optimized executable code, guaranteed to be correct by construction with respect to the functional properties of the programs considered. Section 9.1 indicates the different aspects that are addressed during the compilation of SIGNAL programs. Only the most specific aspects are detailed in this chapter. Section 9.2 first presents simple, yet typical program analyses in a very pragmatic way. These analyses take advantage of the $\mathbb{Z}/3\mathbb{Z}$ encoding to manipulate clocks and reason on their properties. Then, Sect. 9.3 deals with the characterization of SIGNAL programs regarding the monoclock and multiclock implementation models, based on the associated clock information. Finally, Sect. 9.4 briefly explains how this information is used for the code generation process.

A. Gamatié, *Designing Embedded Systems with the SIGNAL Programming Language: Synchronous, Reactive Specification*, DOI 10.1007/978-1-4419-0941-1_9,
© Springer Science+Business Media, LLC 2010

9.1 Overview

The compilation of SIGNAL programs goes beyond the usual compilation process in general-purpose programming languages such as C and Java. More precisely, it deals with the following issues:

Syntax, types, and declarations. A program must be correct regarding the syntactic rules defined in the SIGNAL grammar [5]. All specified expressions must be well typed. All variables must be declared. All manipulated signal variables must be defined except those corresponding to input signals. These correctness analyses are classic in the usual languages.

Issues related to declarative languages. Owing to its declarative nature, the SIGNAL language requires its program to satisfy the basic property of single assignment. Here, further issues are the synthesis of a global control structure associated with the program, and the calculation of a global data dependency graph.

SIGNAL-specific issues. The relational multiclock nature of SIGNAL allows a programmer to locally (or partially) specify synchronization relations between different program subparts. These relations, which must be checked from a global point of view with respect to the program itself, include clock constraints, i.e., invariant properties (see Sect. 8.1.4).

The last two issues are addressed by considering the clock properties and the data dependency information extracted from the analyzed programs. The $\mathbb{Z}/3\mathbb{Z}$ encoding and the conditional dependency graph, presented in the previous chapter, serve as the main supports for reasoning.

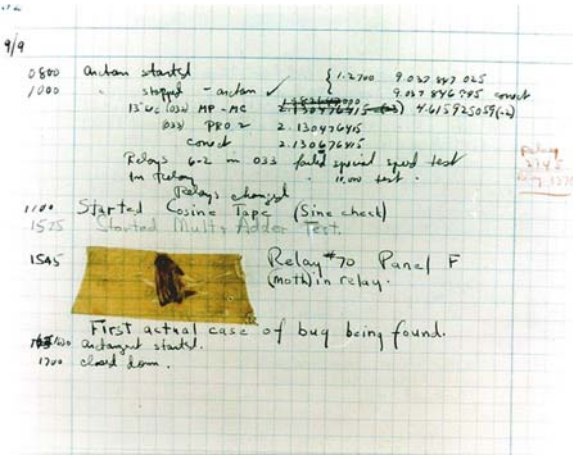
Practically, the SIGNAL compiler identifies some basic steps that enable one to address the above issues [4]. Given a program `P1` to be analyzed:

1. It is transformed into another SIGNAL program `P2` whose statements are all expressed with primitive constructs. That is, all extended constructs in `P1` are equivalently reformulated in terms of the primitive constructs of the language (see Chap. 5). This is achieved via some transformation techniques [9]. During these transformations, the correct declarations and the single assignment principle are also checked.
2. From the resulting program `P2`, clock and data dependency properties are extracted. Chapter 8 gives the properties associated with each primitive construct. As a result, the conditional dependency graph corresponding to `P2` is obtained. Type checking is performed on-the-fly during the construction of this graph.
3. The clock and data dependency properties deduced from the previous step are analyzed so as to check that there are no synchronization errors and to synthesize a hierarchy of clocks and data dependencies. The resulting structure is therefore considered for program optimizations [6], analysis of dynamic properties of programs [7], and code generation.

In the next sections, the focus is mainly on the latter step.

9.2 Abstract Clock Calculus: Analysis of Programs

In SIGNAL, the *clock calculus* refers to the static analysis process implemented in the SIGNAL compiler. It enables one to detect possible specification inconsistencies in a given program, which lead to *bugs* in the corresponding implementations. During this analysis, invariant properties of clocks are inferred and resolved [4, 11]. These properties consist of synchronization and dependency relations, as presented in the previous chapter.



The first actual bug found in the Mark II Aiken Relay Calculator, 1947 (public domain picture)

From this analysis, the clock and dependency information of the program is organized into hierarchical forms which are very useful for optimization and for transformation of programs such as code generation.

Section 9.2.1 discusses some typical issues in program analysis and how they are addressed. Then, Sect. 9.2.2 presents the synthesis of clock hierarchy from a program. Such a hierarchical clock structure is very important during the code generation.

9.2.1 Typical Program Analysis Issues

We present, through short examples, an important analysis of invariant properties of specifications. Among questions relevant during such an analysis, we can mention the following ones:

- Does a program exhibit contradictions?
- Is a program setting constraints on its inputs?
- Is a program cycle-free?
- Is a program deterministic?
- Does a program satisfy some property?

To be able to answer these questions, we use the basic notions already introduced for this purpose: the algebraic encoding of statements in $\mathcal{F}_3$ and the conditional dependency graphs.

9.2.1.1 Does a Program Exhibit Contradictions?

Consider a program `P1` defined as follows:

```
1: process P1 =
2: ( ? integer s1;
3:   ! integer s2, s3, s4; )
4: (| s2 := s1 when (s1 > 0)
5:  | s3 := s1 when not (s1 > 0)
6:  | s4 := s2 + s3
7:  |);
```

For brevity, let us write α for the expression `(s1 > 0)`. We have $\hat{}$`s1` $= \alpha^2$. The clock calculus of the program `P1` yields

$$\begin{aligned}
&\text{Equation at } \texttt{line 4} \Rightarrow \text{clock of } \texttt{s2} \equiv \alpha^2(-\alpha - \alpha^2) \equiv -\alpha - \alpha^2\\
&\text{Equation at } \texttt{line 5} \Rightarrow \text{clock of } \texttt{s3} \equiv \alpha^2(\alpha - \alpha^2) \equiv \alpha - \alpha^2\\
&\text{Equation at } \texttt{line 6} \Rightarrow \text{clocks of } \texttt{s2}, \texttt{s3} \text{ and } \texttt{s4} \text{ are equal}
\end{aligned}$$

The following equality therefore results: $-\alpha - \alpha^2 = \alpha - \alpha^2$, when $\alpha = 0$, i.e., empty (or null) clock. This means that `s1` as well as the output signals of `P1` are always absent. Thus, the program does nothing.

Notice that the presence of empty clocks in a program does not necessarily mean an error in the specification. Instead, it is sometimes needed to statically check that some events that characterize undesirable behaviors in programs never happen.

The above property can be checked with the compiler by using the `-tra` option. By convention, the generic name of the diagnostic files generated with this option is `programName_TRA.SIG`. Here, the content of this file is as follows:

```
1: process P1_TRA =
2:     ( ? integer s1;
3:       ! integer s2, s3, s4; )
4:   pragmas
5:      Main
6:   end pragmas
7: %P1_TRA%;
8:  %^0 ^= s1 ^= s2 ^= s3 ^= s4***WARNING: null clock signals%
```

All empty clocks, are indicated in a *warning* message at line `8`, which specifies a clock constraint that expresses the fact that $\hat{}$`s1`, $\hat{}$`s2`, $\hat{}$`s3`, and $\hat{}$`s4` are equal to the empty clock $\hat{}$`0`.

9.2.1.2 Is a Program Setting Constraints on Its Inputs?

Let us consider the following program:

```
1: process P2 =
2: ( ? integer s1;
3:   ! integer s2, s3; )
```

```
4: (| s2 := s1 when (s1 > 0)
5:  | s3 := s1 + s2
6:  |);
```

For brevity, we again write α for the expression `(s1 > 0)`. It follows that $\alpha^2 = s_1^2$ and the clock calculus of `P2` yields

$$\begin{array}{l}\text{Equation at } \texttt{line 4} \Rightarrow s_2^2 = s_1^2(-\alpha - \alpha^2)\\ \text{Equation at } \texttt{line 5} \Rightarrow s_3^2 = s_1^2 = s_2^2\end{array}$$

By replacing s_1^2 and s_2^2 with α^2 in the clock equality corresponding to the equation at `line 4`, we have

$$\alpha^2 = \alpha^2(-\alpha - \alpha^2),$$

where the possible solutions are either $\alpha^2 = 0$ or $(\alpha^2 \neq 0)$, $-\alpha - \alpha^2 = 1 \Leftrightarrow 1 + \alpha + \alpha^2 = 0$, whence $\alpha = 1$.

From the above analysis, it appears that when the input signal `s1` is present, i.e., the case $\alpha^2 \neq 0$, the conditional expression `s1 > 0` denoted by α must be *true*, since $\alpha = 1$. This is the constraint imposed by the program `P2` on its unique input `s1`. If it can be proved that the environment in which the program will be executed satisfies that constraint, then the above specification of `P2` is safe, meaning that it will react to incoming values. Otherwise, if the environment is not proved to guarantee the constraint, the program may never reacts to any incoming values, i.e., any occurrence of `s1` is ignored; this leads to an unsafe behavior.

The compilation of `P2` with the compiler, using the `-tra` option, generates the following program with a clock constraint *warning* message:

```
1:  process P2_TRA =
2:    ( ? integer s1;
3:      ! integer s2, s3; )
4:    pragmas
5:       Main
6:    end pragmas
7:    (| (| CLK_s1 := ^s1
8:        | CLK_s1 ^= s1 ^= s2 ^= s3
9:        | ACT_CLK_s1{}
10:       |)
11:    | (| (| CLK ^= CLK_s1 |) |)%**WARNING: Clocks constraints%
12:    |)
13:    where
14:    event CLK_s1;
15:   process ACT_CLK_s1 =
16:         ( )
17:      (| (| CLK := when (s1>0) |)
18:       | (| s2 := s1
19:          | s3 := s1+s2
20:          |)
21:       |)
22:      where
23:      event CLK;
24:      end
```

```
25:   %ACT_CLK_s1%;
26:   end
27: %P2_TRA%;
```

The clock constraint indicated at line `11` expresses that the clock `CLK`, which is defined as `when(s1>0)` at line `17`, must be equal to `CLK_s1`, which is the clock of `s1`. It only holds if the condition `s1>0` is *true* whenever `s1` occurs.

9.2.1.3 Are There Clocked Cyclic Definitions in a Program?

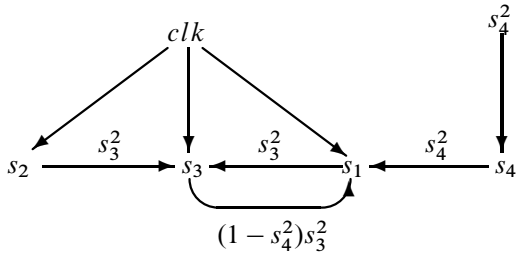
This issue has been already discussed in part through an example in Sect. 8.2.2. Another illustration is given through the following simple program:

```
1: process P3 =
2:  ( ? dreal s2, s4;
3:    ! dreal s1, s3; )
4:  (| s3 := sin(s1) + s2
5:   | s1 := s4 default s3
6:   |);
```

The clock calculus yields the following constraints (for convenience, the variable clk is used here to designate a clock that is common to some signals),

$$clk = s_3^2 = s_2^2 = s_1^2 = s_4^2 + (1 - s_4^2)s_3^2,$$

and the corresponding conditional graph is as follows:



Owing to the short circuit involving `s3` and `s1`, this program will be deadlocked unless the product of the labels of arcs $s_3 \longrightarrow s_1$ and $s_1 \longrightarrow s_3$ is null, meaning that the short circuit is never effective.

Now, let us address the resolution of this short-circuit issue in `P3`. We are interested in the following clock constraint:

$$(1 - s_4^2)s_3^2 = 0.$$

On the other hand, since we have (from the constraints inferred by the clock calculus)

$$s_1^2 = s_4^2 + (1 - s_4^2)s_3^2,$$

we can deduce the following clock equality:

$$s_1^2 = s_4^2 + 0, \text{ i.e., } s_1^2 = s_4^2.$$

The last equality represents the required condition to obtain a cycle-free program corresponding to P3. Taking this equality into account, we obtain the program behavior implemented by the new process called P3bis below:

```
1: process P3bis =
2: ( ? integer s2, s4;
3:   ! integer s1, s3; )
4: (| s3 := sin(s4) + s2
5:  | s1 := s4
6:  |);
```

In the program P3bis, since s1 and s4 have the same clock, the statement s1 := s4 default s3 of P3 reduces to s1 := s4 because of the priority imposed by the deterministic merging operator. Then, s1 can be replaced by s4 in the other equation (at line 4).

From the practical viewpoint, the above clock constraints can also be obtained via a diagnostic file, which is automatically generated by the compiler. For that, the user should use a code generation option, e.g., -c for C code, combined with the -tra option.

In contrast to the Cyclic_P program presented in Sect. 8.2.2, here the compiler generates a file, named P3_bDC_TRA.SIG, indicating the clock constraints that must be satisfied to have a cycle-free program corresponding to P3. When the compiler is able to determine the required conditions to be satisfied by the clocks of a program so as to avoid cyclic dependencies, it only generates a diagnostic file containing these conditions (i.e., P3_bDC_TRA.SIG). Otherwise, it generates a diagnostic file that explicitly specifies the detected cycles as illustrated in Sect. 8.2.2.

On the basis of the same reasoning that allowed us to deduce the P3bis program previously, the compiler generates the following diagnostic file for process P3:

```
1:  process P3_bDC_TRA =
2:      ( ? dreal s2, s4;
3:          boolean C_s4, C, C_s1;
4:        ! dreal s1, s3; )
5:    pragmas
6:       Main
7:    end pragmas
8:    (| (| Tick := true
9:        | when Tick ^= C_s4 ^= C ^= C_s1
10:       | ACT_Tick{}
11:       |)
12:    | (| (| when C ^= when C_44 |)
13:       | (| when C_s1 ^= when C_41 |)
14:       |)%**WARNING: Clocks constraints%
15:    |)
16:   where
17:   boolean Tick;
18:   process ACT_Tick =
19:      ( )
20:    (| when Tick ^= C_41 ^= C_44
21:     | (| when C_s4 ^= s4 |)
22:     | (| when C_s1 ^= s2 ^= s1 ^= s3
23:       | ACT_C_s1{}
```

```
24:         |)
25:      | (| C_41 := C_s4 or C_s1
26:         | C_44 := (not C_s4) and C_s1
27:         |)
28:      |)
29:     where
30:     boolean C_41, C_44;
31:     process ACT_C_s1 =
32:        ( )
33:      (| (| s1 := (s4 when C_s4) default (s3 when C)
34:          | s3 := sin(s1)+s2
35:          |) |)
36:     %ACT_C_s1%;
37:     end
38: %ACT_Tick%;
39: end
40: %P3_bDC_TRA%;
```

In `P3_bDC_TRA`, the *warning* message mentioned at line `14` concerns the two constraints defined at lines `12` and `13`. In this program, the Boolean variables prefixed by `C` encode clocks. For instance, `C_s4` and `C_s1`, respectively, encode the clocks of `s4` and `s1`. Whenever such a Boolean variable holds the value true, it means that the encoded clock is present (see lines `21` and `22`).

Let us focus first on the clock constraint at line `13`, which expresses the fact that the clock of `s1` is the same as the clock encoded by `when(C_41)`. The definition of `C_41` is given at line `25`: it is equal to the disjunction of `C_s4` and `C_s1` (so, `when(C_41)` denotes the set of instants at which either `s4` or `s1` is present). As a result, the constraint at line `13` holds only if the following equality is true: `C_s1 = C_s4 or C_s1`. Since, the unique solution is that `C_s1 = C_s4`, it means that `s1` and `s4` are required to have the same clock.

Now, by considering the fact that `C_s1 = C_s4` in the other clock constraint (line `12`), we prove the Boolean variable `C_44` defined at line `26` has the value *false*, whence `C` also has the value *false*. As a consequence, the second argument of the `default` operator is never taken into account in the equation at line `33`. Finally, this equation reduces to `s1 := s4` and we obtain the same result as before.

9.2.1.4 Is a Program Temporally Deterministic?

We often need deterministic behaviors when specifying a program for the simple reason that this facilitates the behavioral analysis of such a program. By *temporal determinism*, we mean that the program reacts only on the occurrence of its inputs. We can refer to this program as a function.

Consider the following program, which specifies a counter with an external reset:

```
1: process P4 =
2:  ( ? event r;
3:    ! integer cnt, pre_cnt;)
4:  (| cnt := (0 when r) default (pre_cnt + 1)
5:   | pre_cnt := cnt $ 1 init 0
6:   |);
```

Its clock calculus yields the following clock equalities:

$$\text{Equation at } \texttt{line 4} \Rightarrow cnt^2 = r^2 + (1 - r^2)pre_cnt^2$$
$$\text{Equation at } \texttt{line 5} \Rightarrow cnt^2 = pre_cnt^2$$

from which we can deduce that $pre_cnt^2 \geq r^2$.

This clearly shows that the outputs of the program `P4`, `cnt` and `pre_cnt`, can occur while the input `r` is absent. So, this program is not temporally deterministic.

If one wishes to make the `P4` program deterministic, a possible solution consists in inserting the following synchronization equation in the process:

```
pre_cnt ^= (r default i)
```

where `i` is an additional input of `P4`. Now, we can observe that all reacting instants of the program are completely specified.

Similarly to the previously analyzed programs, the compilation of `P4` with `-c` and `-tra` options produces a diagnostic file that contains a program annotated with clock constraint *warning* messages. The resulting program is similar to `P3_bDC_TRA`.

9.2.1.5 Does a Program Satisfy Some Property?

SIGNAL can be used as a partial proof system, as illustrated for the following simple program:

```
1: process P5 =
2: ( ? boolean s1;
3:   ! boolean s2, s3;
4: (| s3 := s1 when (s1 /= s2)
5:  | s2 := s1 $ 1 init 0
6:  |);
```

Here, all manipulated signals are of Boolean type. Roughly speaking, the program `P5` defines an output signal `s3`, which takes the value of input signal `s1` whenever this value is different from the previous value of `s1`, defined as the signal `s2`.

Let us show that at any instant $t \geq 1$, the value of `s3` is equal to the negation of the value of `s3` at the previous instant $t - 1$.

The clock calculus of this process yields

$$\text{Equation at } \texttt{line 4} \Rightarrow s_3 = s_1(s_1 s_2 - s_1^2 s_2^2) \Leftrightarrow s_3 = (s_1^2 s_2 - s_1^3 s_2^2) \Leftrightarrow s_3 = s_1 s_2(s_1 - s_2)$$
$$\text{Equation at } \texttt{line 5} \Rightarrow s_1^2 = s_2^2$$

As a first observation, from the expression $s_3 = s_1 s_2(s_1 - s_2)$, one can deduce that `s3` is absent when:

- Either `s1` is absent, which means that $s_1^2 = 0$ (the same holds for `s2`)

- Or the current value of `s1`, denoted by s_1, is identical to its previous value, represented by s_2

  On the other hand, if we denote by s'_3 the next value of s_3, we have

$$s'_3 = s'_1 s'_2 (s'_1 - s'_2),$$
$$s'_2 = s_1.$$

Now, let us calculate the product $s_3 s'_3$ in which s_3 and s'_3 are replaced by their corresponding expressions and s'_2 is replaced by s_1:

$$s_3 s'_3 = s_1^2 s'_1 s_2 (s_1 - s_2)(s'_1 - s_1).$$

This product is nonnull only when $s_1 \neq s_2$ and $s'_1 \neq s_1$ in the presence of all signals involved. We have to notice that when these signals are present, they hold as a value either 1 or -1, which, respectively, encode the Boolean constant values *true* and *false*.

From this observation, considering that the product $s_3 s'_3$ is nonnull leads to the following two situations:

1. Either s_1 is equal to 1 and s'_1 and s_2 are both equal to -1
2. Or s_1 is equal to -1 and s'_1 and s_2 are both equal to 1

In both cases, one can deduce that $s'_1 = s_2$. As a result, the product $s_3 s'_3$ becomes

$$s_3 s'_3 = s_1^2 s_2^2 (s_1 - s_2)(s_2 - s_1) = -s_1^2 s_2^2 (s_1 - s_2)^2.$$

According to the properties of $\mathbb{Z}/3\mathbb{Z}$, if $s_3 s'_3$ is nonnull, then each of the terms s_1^2, s_2^2, and $(s_1 - s_2)^2$ is inevitably equal to 1; thus

$$s_3 s'_3 = -1.$$

This is true when either $s_3 = 1$ and $s'_3 = -1$ or $s_3 = -1$ and $s'_3 = 1$, which proves the property on the values of the output signal `s3`.

9.2.2 Hierarchy Synthesis for Abstract Clocks

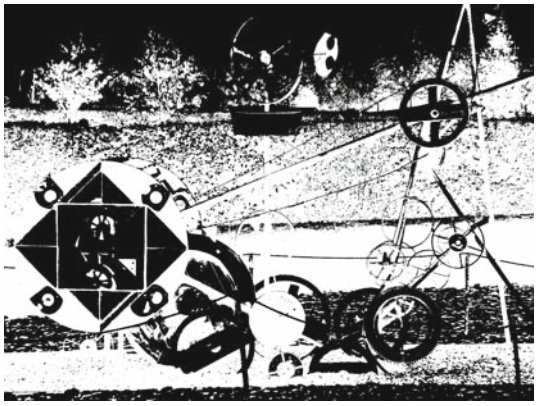
Another purpose of the clock calculus is to define a clock hierarchy associated with an analyzed program [1, 2]. Such a hierarchy can be considered afterwards for various purposes, among which we can mention an optimized code generation as discussed later in this chapter.

The synthesis of clock hierarchy in SIGNAL programs relies on an efficient algorithm that has been implemented in the compiler with respect to the following important rules:

- If `b` is a *free* Boolean signal (meaning that `b` results from the evaluation of a function with non -Boolean arguments, or it is an input signal of the program, or it is the status of a Boolean memory), then the clock defined by the set of instants at which `b` is *true* (i.e., `when b`) and the clock defined by the set of instants at which `b` is *false* (i.e., `when (not b)` are put under the clock of `b` (i.e., `^b`). Both are called *downsamplings*.

- If a clock `clk` is a subclock of another clock `clk_bis`, then every subclock of `clk` is a subclock of `clk_bis`.
- Let `clk` be a clock defined as a function of downsamplings $\texttt{clk}_1, \ldots, \texttt{clk}_n$. If all these downsamplings are subclocks of `clk_bis`, then `clk` is also a subclock of `clk_bis`.

The resulting hierarchy is a collection of interconnected trees of clocks, called a *forest*. The partial order defined by this forest represents *dependencies* between clocks: the actual value of a clock `clk` may be needed to compute the actual value of a given clock `clk_bis` only if `clk` lies above `clk_bis` according to this partial order. No hierarchy is defined on the roots of the trees, but constraints can exist.



Abstract picture of an artwork from Jardin des Cultures MOSAÏC (Houplin-Ancoisne, France), 2009

When this forest reduces to a single tree, then a single master clock exists, from which other clocks derive. In this latter case, the program can be executed in master mode, i.e., by requiring the data from the environment. Such a program is referred to as an *endochronous* program. Figure 9.1 illustrates the clock hierarchy of an endochronous program. It is described by a unique tree where the root node represents the master clock (*Ck*). We can notice that from this global tree, one can derive several "endochronous" subtrees (e.g., T_i).

If several trees remain, as illustrated in Fig. 9.2, additional synchronization has to be provided by the external world or environment (e.g., small real-time kernels [3]) or by another SIGNAL program. In this case, the program is referred to as an *exochronous* program.

The conditional dependency graph is attached to the forest in the following way. The signals available at a given clock are attached to this clock, and so are the expressions defining these signals. The conditional hierarchical graph obtained is the basis for sequential as well as distributed code generation.

Moreover, the proper syntax of SIGNAL can be used to represent this graph. For that purpose, the compiler rewrites the clock expressions as SIGNAL Boolean expressions: the operator `default` represents the upper bound of clocks (i.e., the sum) and the `when` operator represents the lower bound (i.e., the product). Then, any clock expression may be recursively reduced to the sum of monomials, where each monomial is a product of downsamplings (otherwise, the clock is a root). The definitions of the signals are also rewritten to make explicit the clocks of the calculations that define these signals.

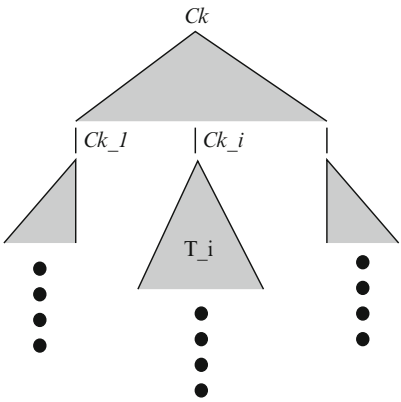


Fig. 9.1 Clock hierarchy of an endochronous program

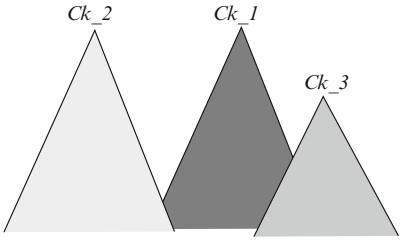


Fig. 9.2 Clock hierarchy of an exochronous program

The rewritten program is equivalent to the initial one, but the clock and dependency calculus is now solved, and all the clocks handled in the program are made precisely explicit. The program so obtained will be referred to as the *solved form* of the program considered.

9.3 Exploiting Hierarchies of Abstract Clocks in Practice

This section shows a use of the previous clock hierarchy information to characterize SIGNAL programs with respect to different implementation models. Two kinds of programs are distinguished here: an endochronous program (see Sect. 9.3.1) and an exochronous program (Sect. 9.3.2). In addition, it is shown how the latter is transformed into the former (see Sect. 9.3.3).

9.3.1 Endochronous Programs

Let us consider the following program, called `Zero_Counter`:

```
1:  process Zero_Counter =
2:     ( ? integer s;
3:       ! integer cnt; )
4:     (| cnt ^= when (s = 0)
```

```
5:      | cnt := pre_cnt + 1
6:      | pre_cnt := cnt$1 init 0
7:      |)
8:    where
9:      integer pre_cnt;
10: end; %process Zero_Counter%
```

The basic behavior described by this program consists in counting the occurrences of input signal `s`, where `s` takes zero as a value.

9.3.1.1 Clock Hierarchy

The clock hierarchy shown in Fig. 9.3 corresponds to the program `Zero_Counter`. We can observe that the clocks of `cnt` and `pre_cnt` denoted by *clk_cnt* and *clk_pre_cnt*, respectively, are equal. They are defined as the set of logical instants at which signal `s` holds the value 0, noted as $[s = 0]$. There is a unique root in the resulting clock hierarchy: *clk_s*, the clock of `s`. So, `Zero_Counter` is endochronous.

The SIGNAL elementary processes defined with monoclock primitive constructs (where all the variables of all signals involved are distinct), i.e.,

- s_n `:= R(`s_1`,...,`s_{n-1}`)` (instantaneous relations/functions)
- s_2 `:=` s_1 `$ 1 init c` (delay)

are endochronous.

In contrast, the processes defined with multiclock primitive constructs, i.e.,

- s_2 `:=` s_1 `when b` (undersampling)
- s_3 `:=` s_1 `default` s_2 (deterministic merging)

are not endochronous.

9.3.1.2 Hierarchized Conditional Dependency Graph

The above clock hierarchy is used as central information together with the conditional dependency graph (see Chap. 8) associated with the program `Zero_Counter` to define a new graph, called *hierarchized* conditional dependency graph.

Such a graph can be visualized in a textual format, under the form of a SIGNAL program that is generated automatically by the compiler on demand by the user (option `-tra`). The file named `Zero_Counter_TRA.SIG` contains the resulting graph.

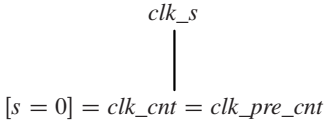


Fig. 9.3 Clock hierarchy for process `Zero_Counter`

The following program corresponds to the generated[1] graph associated with Zero_Counter after compilation:

```
1:  process Zero_Counter_TRA =
2:       ( ? integer s;
3:         ! integer cnt;
4:       )
5:     pragmas
6:        Main
7:     end pragmas
8:     (| (| CLK_s := ^s
9:         | CLK_s ^= s
10:        | ACT_CLK_s{}
11:        |) |)
12:    where
13:    event CLK_s;
14:    process ACT_CLK_s =
15:         ( )
16:       (| (| CLK_cnt := when (s=0)
17:           | CLK_cnt ^= cnt
18:           | ACT_CLK_cnt{}
19:           |) |)
20:       where
21:       event CLK_cnt;
22:       process ACT_CLK_cnt =
23:            ( )
24:          (| CLK_cnt ^= pre_cnt
25:           | (| cnt := pre_cnt+1
26:              | pre_cnt := cnt$1 init 0
27:              |)
28:           |)
29:          where
30:          integer pre_cnt;
31:          end
32:       %ACT_CLK_cnt%;
33:       end
34:    %ACT_CLK_s%;
35:    end
36:  %Zero_Counter_TRA%;
```

In the program Zero_Counter_TRA, line 8 defines the clock of signal s. At line 10, there is a call to the subprocess ACT_CLK_s, which defines the actions to be performed at this clock. Such actions consist of first the definition at line 16 of CLK_cnt the clock of cnt, and a call at line 18 to a subprocess of ACT_CLK_s, called ACT_CLK_cnt. The new subprocess defines the actions to be performed at the clock CLK_cnt. It first states that the clock of signal pre_cnt is the same as CLK_cnt at line 24. Finally, lines 25 and 26 define how cnt is increased at CLK_cnt.

[1] For the sake of clarity, all occurrences of the string pattern "XZX" automatically generated by the compiler have been replaced by "CLK_cnt" to denote the clock associated with the signal cnt.

From a global point of view, one can observe that `Zero_Counter_TRA` suitably reflects the clock hierarchy depicted in Fig. 9.3. In addition, each clock in this graph is associated with the specific set of actions that must be performed at the corresponding instants.

Such a program is very useful because it first allows one to see the compilation result; and it also serve to analyze clock synchronization issues in the initial program.

9.3.2 Exochronous Programs

Let us consider a new program called `Exochronous_Proc`, defined as follows:

```
process Exochronous_Proc =
    ( ? integer s1, s2;
      ! integer s3; )
    (| s3 := 0 when(s1 = 1) default (s2 + 10)
     |); %process Exochronous_Proc%
```

The behavior described by this program is such that the output signal `s3` takes the value 0 when signal `s1` occurs with the value 1; otherwise, `s3` takes the value of signal `s2` increased by 10.

9.3.2.1 Clock Hierarchy

The clock analysis of this process yields the hierarchy illustrated in Fig. 9.4.

In contrast to the previous program `Zero_Counter`, here there is no unique clock. Thus, `Zero_Counter` is said to be exochronous. Note that clk_s3 depends on both the subclock $[s1 = 0]$ of clk_s1 and the clock clk_s2. More precisely, it consists of their union. In such a situation, the clock clk_s3 is set at the same level as the clock with the highest level in the clock hierarchy, among $[s1 = 0]$ and clk_s3. Thus, it shares the same level with clk_s3.

There are two actual root clocks clk_s1 and clk_s2, from which all the other clocks of the program can be determined. The combination of the conditional dependency graph and the clock hierarchy of `Zero_Counter` leads to the hierarchical conditional dependency graph represented by the `Exochronous_Proc_TRA` program, discussed below and generated automatically by the SIGNAL compiler.

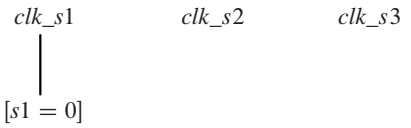


Fig. 9.4 Clock hierarchy for process `Exochronous_Proc`

9.3.2.2 Hierarchized Conditional Dependency Graph

In Exochronous_Proc_TRA, one can distinguish the definition of the clock of s1 and s2 at lines 8 and 12, respectively. The call to the subprocess ACT_CLK_s1 at line 10 defines the clock CLK_6 that corresponds to $[s1 = 1]$. Line 15 defines another clock, CLK, which denotes the set of instants at which s2 is present while the value of s1 is different from 1. The clocks CLK_6 and CLK are both used in the equation at line 18 to define the value of signal s3. At line 16, the clock of s3 is defined as the union of CLK_6 and CLK.

```
1:  process Exochronous_Proc_TRA =
2:    ( ? integer s1, s2;
3:      ! integer s3;
4:    )
5:   pragmas
6:     Main
7:   end pragmas
8:    (| (| CLK_s1 := ^s1
9:        | CLK_s1 ^= s1
10:       | ACT_CLK_s1{}
11:       |)
12:    | (| CLK_s2 := ^s2
13:       | CLK_s2 ^= s2
14:       |)
15:    | (| CLK := CLK_s2 ^- CLK_6 |)
16:    | (| CLK_s3 := CLK_6 ^+ CLK_s2
17:       | CLK_s3 ^= s3
18:       | s3 := (0 when CLK_6) default ((s2+10) when CLK)
19:       |)
20:    |)
21:  where
22:    event CLK_s3, CLK, CLK_s2, CLK_6, CLK_s1;
23:    process ACT_CLK_s1 =
24:     ( )
25:     (| (| CLK_6 := when (s1=1) |) |)
26:    %ACT_CLK_s1%;
27:  end
28:  %Exochronous_Proc_TRA%;
```

One can observe that here also the clock hierarchy shown in Fig. 9.4 is entirely reflected by the Exochronous_Proc_TRA program.

9.3.3 Endochronization of Exochronous Programs

9.3.3.1 A First Example

Let us consider the Exochronous_Proc program presented previously. The associated specification leads to an exochronous program. It can be slightly modified such as to make this program endochronous.

The general idea for the solution of this endochronization consists in introducing additional Boolean signals and clock constraints to the original specification as follows:

- The added Boolean signals are defined as inputs of the program.
- Each input signal *s* of `Exochronous_Proc` is associated with an added Boolean signal *b* through a synchronization constraint, which expresses that the clock of *s* is a downsampling of *b*, e.g., `s ^= when b`.
- All added Boolean signals are made synchronous.

By applying the above modifications to the `Exochronous_Proc` program, we obtain the following program:

```
1: process Endo_Exochronous_Proc =
2:     ( ? boolean C_s1, C_s2;
3:         integer s1, s2;
4:       ! integer s3; )
5:     (| s3 := 0 when(s1 = 1) default (s2 + 10)
6:      | s1 ^= when C_s1
7:      | s2 ^= when C_s2
8:      | C_s1 ^= C_s2
9:      |);%process Endo_Exochronous_Proc%
```

The added Boolean signals are `C_s1` and `C_s2`.

The new clock hierarchy associated with this program is described by the `Endo_Exochronous_Proc_TRA` program,[2] shown below:

```
1:  process Endo_Exochronous_Proc_TRA =
2:    ( ? boolean C_s1, C_s2;
3:        integer s1, s2;
4:      ! integer s3; )
5:    pragmas
6:      Main
7:    end pragmas
8:    (| (| CLK_C_s1 := ^C_s1
9:        | CLK_C_s1 ^= C_s1 ^= C_s2
10:       | ACT_CLK_C_s1{}
11:       |) |)
12: where
13:   event CLK_C_s1;
14:   process ACT_CLK_C_s1 =
15:   ( )
16:   (| (| CLK_s1 := when C_s1
17:       | CLK_s1 ^= s1
18:       | ACT_CLK_s1{}
19:       |)
20:    | (| CLK_s2 := when C_s2
21:       | CLK_s2 ^= s2
```

[2] Here also, for the sake of clarity, all occurrences of the string patterns "`XZX`" and "`XZX_19`," automatically generated by the compiler have been replaced by "`CLK_s1`" and "`CLK_s2`," respectively, to denote the clocks associated with signals `s1` and `s2`.

```
22:       |)
23:    | (| CLK_s3 := CLK_s2 ^+ CLK
24:       | CLK_s3 ^= s3
25:       | (| s3:= (0 when CLK) default ((s2+10) when CLK_25) |)
26:       |)
27:    | (| CLK_25 := CLK_s2 ^- CLK |)
28:    |)
29:   where
30:    event CLK_25, CLK_s3, CLK_s2, CLK_s1, CLK;
31:    process ACT_CLK_s1 =
32:     ( )
33:     (| (| CLK := when (s1=1) |) |)
34:    %ACT_CLK_s1%;
35:   end%ACT_CLK_C_s1%;
36: end%Endo_Exochronous_Proc_TRA%;
```

Now, one can observe that there is a single root clock, `CLK_C_s1` (defined at line `8`).

9.3.3.2 Another Example: Resettable Counter

Let us consider again the `R_COUNTER` process defined in Sect. 4.3.2:

```
1:  process R_COUNTER =
2:   { integer v0; }
3:   ( ? boolean reset;
4:     ! boolean reach0;
5:       integer v; )
6:   (| zv := v $ 1 init 0
7:    | vreset := v0 when reset
8:    | zvdec := zv when (not reset)
9:    | vdec := zvdec - 1
10:   | v := vreset default vdec
11:   | reach0 := true when (zv = 1)
12:   |)
13:  where
14:    integer zv, vreset, zvdec, vdec;
15:  end; %process R_COUNTER%
```

The following clock information can be deduced in $\mathcal{F}_3$:

- Equation at `line 6`: $\mathtt{zv}^2 = \mathtt{v}^2$
- Equation at `line 7`: $\mathtt{vreset}^2 = -\mathtt{reset} - \mathtt{reset}^2$
- Equation at `line 8`: $\mathtt{zvdec}^2 = \mathtt{zv}^2(\mathtt{reset} - \mathtt{reset}^2)$
- Equation at `line 9`: $\mathtt{vdec}^2 = \mathtt{zvdec}^2$
- Equation at `line 10`: $\mathtt{v}^2 = \mathtt{vreset}^2 + \mathtt{vdec}^2(1 - \mathtt{vreset}^2)$
- Equation at `line 11`: $\mathtt{reach0}^2 = -\alpha - \alpha^2$

Here α denotes the expression `(zv = 1)`. On the other hand, from the expression `(zv = 1)`, we can also deduce that $\mathtt{zv}^2 = \alpha^2$.

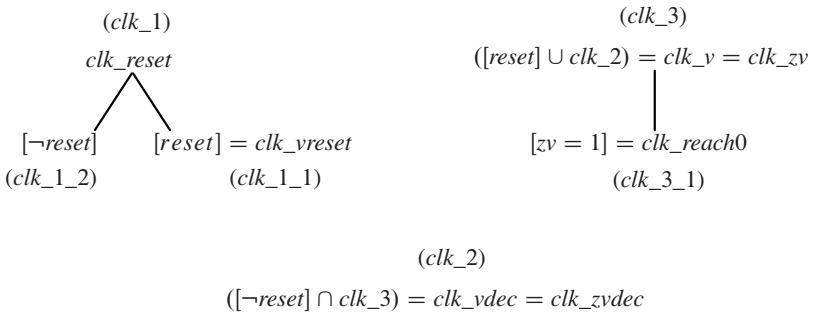


Fig. 9.5 Clock trees resulting from the analysis of R_COUNTER

The above clock information can be simplified into the following equalities between clocks:

$$\begin{aligned}
\texttt{vreset}^2 &= -\texttt{reset} - \texttt{reset}^2 && \text{(a)}\\
\texttt{zv}^2 = \alpha^2 = \texttt{v}^2 &= (-\texttt{reset} - \texttt{reset}^2) + \texttt{v}^2(\texttt{reset} - \texttt{reset}^2) && \text{(b)}\\
\texttt{vdec}^2 = \texttt{zvdec}^2 &= \texttt{v}^2(\texttt{reset} - \texttt{reset}^2) && \text{(c)}\\
\texttt{reach0}^2 &= -\alpha - \texttt{v}^2 && \text{(d)}
\end{aligned}$$

The clock calculus of the R_COUNTER program yields the three clock trees represented in Fig. 9.5. These trees are rooted by the clocks *clk*_1, *clk*_2, and *clk*_3. There are relations between them via the clocks that form their nodes. For instance, clock *clk*_2 is a subset of clock *clk*_1_2, which denotes the set of instants at which reset carries the value *false*, noted $[\neg reset]$.

Here, an important question is whether or not a unique clock tree can be defined from these three clock trees. In other words, is the R_COUNTER program endochronous? If this is the case, one could determine when every signal of R_COUNTER is defined on the basis only of the input of the program, i.e., reset.

Let us consider the definition of *clk*_2. The signal zvdec is defined *when reset carries the value false* and the signals v and zv are defined (i.e., are present). This is expressed by the equations at lines 6 and 8 in the program, and is verified by substituting the variable reset with -1 in equalities (b) and (c). We obtain

$$\begin{aligned}
\texttt{zv}^2 = \alpha^2 = \texttt{v}^2 \quad & \text{(b')} \textit{ with } [\texttt{reset} \leftarrow -1]\\
\texttt{vdec}^2 = \texttt{zvdec}^2 = \texttt{v}^2 \quad & \text{(c')} \textit{ with } [\texttt{reset} \leftarrow -1]
\end{aligned}$$

The new equality (c′) specifies that when reset carries the value *false*, the signals vdec and zvdec are defined if v is defined.

But the new equality (b′) does not completely define the clock of v when reset carries the value *false*. It states that v can be either present or absent because the possible solutions of this equality are 0 and 1.

From the equality (b′), we can conclude that the unique input reset of the program R_COUNTER is not sufficient alone to decide on the presence of v. Since

no master clock exists in this program, R_COUNTER is not endochronous and is nondeterministic. Note that R_COUNTER is exochronous.

The clock hierarchy illustrated in Fig. 9.5 is represented by the following SIGNAL process:

```
(| (| clk_1 ^= reset
    | (| clk_1_1 := true when reset
       | clk_1_1 ^= vreset
       | clk_1_2 := true when (not reset)
       |)
    |)
 | (| clk_2 := clk_1_2 when clk_3
    | clk_2 ^= vdec ^= zvdec
    |)
 | (| clk_3 := clk_1_1 default clk_2
    | clk_3 ^= v ^= zv
    | (| clk_3_1 := true when(zv=1)
       | clk_3_1 ^= reach0
       |)
    |)
 |)
```

The hierarchy is syntactically represented by the successive composition operations from inner levels to outer levels. The variables clk_i represent names (generated by the compiler) of the clocks considered as signals.

Now, let us consider the following program ENDO_R_COUNTER, where the process R_COUNTER is used in a given context:

```
process ENDO_R_COUNTER =
  { integer v0; }
  ( ? boolean h;
    ! boolean reach0;
      integer v; )
  (| h ^= v
   | reset := (^reach0 when (^h)) default (not (^h))
   | (reach0, v) := RCOUNT {v0}(reset)
   |)
  where
    boolean reset;
    ...
  end;
```

An external clock, represented by the Boolean input signal h, defines the instants at which v takes a value, i.e., is defined. The reset signal is synchronous with h and it carries the value *true* exactly when reach0 is present.

Now, a master clock ($h^2 = v^2 = reset^2$) and the tree illustrated in Fig. 9.6 can be built by the compiler. ENDO_R_COUNTER is therefore endochronous and deterministic.

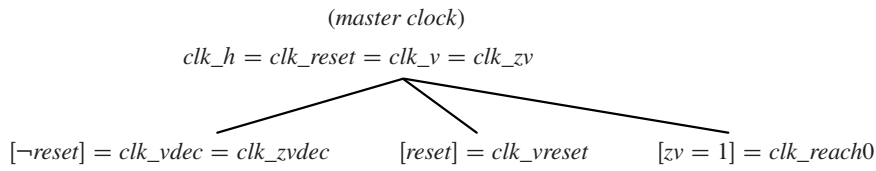
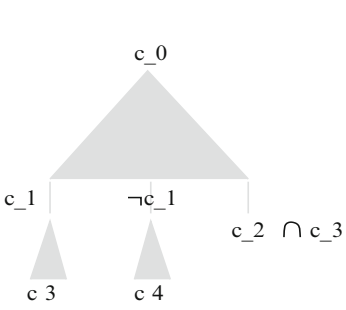


Fig. 9.6 Clock tree resulting from the analysis of ENDO_R_COUNTER




```
while true do
  if cond(c_0)
  then if cond(c_1)
       then if cond(c_3)
            then ... end
       else if cond(c_4)
            then ... end
       end
       if cond(c_2) and cond(c_3)
       then ...
       end
  end
end
```

Fig. 9.7 Code generated for an endochronous program

9.4 Code Generation

In this section, we only discuss how sequential code is produced with the signal compiler from endochronous programs. We point out that the compiler also enables one to generate distributed code [10]. A more detailed discussion of the different code generation strategies in the POLYCHRONY environment can be found elsewhere [8].

The code generation of a SIGNAL program P strongly relies on the clock hierarchy resulting from the clock calculus of P (see Fig. 9.7). This code can be obtained in different target languages, among which the most used are C, C++, and Java.

When a program P is proved to be endochronous, the generation of its associated code is straightforward. Each node of the clock tree corresponding to P is characterized by a Boolean expression that expresses a condition. The program statements that depend on each node are computed whenever the associated condition is evaluated to be true, meaning that the expressed clock is present.

On the other hand, the code generation takes into account the conditional dependency graph that characterizes the order according to which statements are to be computed at the same clock instants.

Figure 9.4 roughly illustrates a sequential code consisting of an infinite loop that executes statements depending on the values of the conditional expressions corresponding to the clocks of an endochronous program. Such a code follows a clock-driven execution model (see Chap. 2). One can observe the tree of conditional

statements resulting from the clock hierarchy. For instance, the statements associated with clock `c_3` are executed only if `c_0` and then `c_1` are satisfied.

This generation schema strongly contributes to having an optimized code from SIGNAL programs.

9.4.1 An Example of a Generated Code Sketch

To give a more precise idea of what an automatically generated code looks like for SIGNAL programs, let us consider again the program `Zero_Counter` analyzed in Sect. 9.3.1. The corresponding C code generated by the compiler, referred to as the *body* of the program, is as follows (a few comments, have been manually added to help the reader understand the code):

```
/*
  Generated by Polychrony version  V4.15.10
*/
#include "Zero_Counter_types.h"
#include "Zero_Counter_externals.h"
#include "Zero_Counter_body.h"

/*  ==> parameters and indexes */
/*  ==> input signals     */
static int s;
/*  ==> output signals    */
static int cnt;
/*  ==> local signals     */
static logical C_pre_cnt;

/*** Note that clocks are encoded as Boolean variables ***/

EXTERN logical Zero_Counter_initialize()
/*** Here are initialized all state variables of the
     program before entering the main loop of the generated
     reactive code ***/
{
    cnt = 0;
    Zero_Counter_STEP_initialize();
    return TRUE;
}
static void Zero_Counter_STEP_initialize()
/*** Here are initialized all state variables of the
     program at the end of each reaction corresponding to
     a single logical instant ***/
{
}
```

```
EXTERN logical Zero_Counter_iterate()
{
    if (!r_Zero_Counter_s(&s)) return FALSE;
/*** read s: returns false if end of file ***/
/*** r_Zero_Counter_s is defined in an input/output module,
     which can be adapted to the environment: files,
     sensors, etc. ***/

    C_pre_cnt = s == 0;
    if (C_pre_cnt)
        {
        cnt = cnt + 1;
        w_Zero_Counter_cnt(cnt);

/*** write the computed value of cnt in an output file
     typically ***/
        }
    Zero_Counter_STEP_finalize();

/*** reinitializing all state variables for the next
     logical instant ***/

    return TRUE;
}

EXTERN logical Zero_Counter_STEP_finalize()
{
    Zero_Counter_STEP_initialize();
    return TRUE;
}
```

The above code is generated in a file named *Zero_Counter_body.c*.

9.5 Exercises

9.1. Give the analysis result of the following processes. If there are any problems, suggest a solution to solve them:

1.
```
process S_OccurrenceNumber =
  ( ? integer s;
    ! integer cnt; )
  (| cnt := (cnt $ 1 init 0) + 1 when ^s |);
```
2.
```
process Bi_ValuedState =
  (? event up, down;
   ! integer s)
  (| s1 := up when (pre_s /= 1)
   | s2 := down when (pre_s /= 2)
   | s := (1 when s1) default (2 when s2)
```

```
  | pre_s := s$1 init 1
  |)
where
  integer pre_s;
  event s1, s2;
end;
```

Expected Learning Outcomes:

- The **compilation of any** SIGNAL **program** `P1` can be summarized by the following **basic steps**:
  1. Transformation of `P1` into another SIGNAL program `P2` where all extended constructs are replaced by their corresponding expression in terms of primitive constructs
  2. Extraction of clock and data dependency properties from `P2`
  3. Analysis of these properties and synthesis of a hierarchy of clocks and data dependencies for code generation
- The **clock calculus** is the heart of the compilation phase of SIGNAL programs, during which various properties, mainly related to clocks, are statically checked via the $\mathbb{Z}/3\mathbb{Z}$ encoding, e.g.,
  – Existence of contradictory clock definitions
  – Existence of constraints on the program's inputs
  – Determinism of a program

  These properties aim at guaranteeing the consistency of synchronization relations between signals.
- If a master (master) clock can be found in a program, it is determined by the clock calculus via a **hierarchization of the whole program's clocks**.
- An **endochronous** program is a program for which a master clock exists. It is implemented by a monoclocked system model.
- An **exochronous** program is a program for which there are several locally master clocks. It is implemented by a multiclocked system model.
- After the clock calculus, **code generation** is possible. This is performed automatically on the basis of the synthesized clock hierarchy information.

References

1. Amagbégnon P, Besnard L, Le Guernic P (1995) Implementation of the data-flow synchronous language SIGNAL. In: ACM Conference on Programming Language Design and Implementation (PLDI'95), pp 163–173
2. Amagbégnon TP (1995) Forme canonique arborescente des horloges de SIGNAL. PhD thesis, Université de Rennes I, IFSIC, France (document in French)
3. Benveniste A, Berry G (1991) The Synchronous approach to reactive and real-time systems. Proceedings of the IEEE 79(9):1270–1282
4. Besnard L (1992) Compilation de SIGNAL : horloges, dépendances, environnement. PhD thesis, Université de Rennes I, IFSIC, France (document in French)

5. Besnard L, Gautier T, Le Guernic P (2008) SIGNAL v4 – INRIA Version: Reference Manual. Available at: http://www.irisa.fr/espresso/Polychrony
6. Chéron B (1991) Transformations syntaxiques de Programmes SIGNAL. PhD thesis, Université de Rennes I, IFSIC, France (document in French)
7. Dutertre B (1992) Spécification et preuve de systèmes dynamiques. PhD thesis, Université de Rennes I, IFSIC, France (document in French)
8. Besnard L, Gautier T, Talpin J-P (2009) Code Generation Strategies in the POLYCHRONY Environment. Research report number 6894, INRIA. Available at: http://hal.inria.fr/docs/00/37/24/12/PDF/RR-6894.pdf
9. Gautier T (1984) Conception d'un langage flot de données pour le temps réel. PhD thesis, Université de Rennes I, IFSIC, France (document in French)
10. Gautier T, Le Guernic P (1999) Code Generation in the SACRES Project. In: Safety-critical Systems Symposium, SSS'99. Springer, Huntingdon, UK
11. Nebut M (2003) An overview of the SIGNAL clock calculus. In: Synchronous Languages, Applications, and Programming (SLAP'03). Electronic Notes in Theoretical Computer Science, Porto, Portugal 88:39–54

Chapter 10
Advanced Design Concepts

Abstract This chapter presents some very useful concepts of the SIGNAL language which are necessary for more advanced design. The first concept is the notion of *modularity*, presented in Sect. 10.1. It promotes reusability, which is very important when designing complex systems. Section 10.2 describes different ways to specify a system by *abstracting* away the unessential details. In Sect. 10.3, the *oversampling* mechanism is introduced, which enables one to refine a given clock into faster clocks. Finally, Sect. 10.4 presents how *assertions* are described in SIGNAL programming.

10.1 Modularity

In SIGNAL, *modules* are very important when one needs to define libraries of *reusable* components or services. The general notation of a module named `M` is as follows:

```
module M =

%type declarations%
type t1 = integer;
...

%constant declarations%
constant c1 = 1000;
...

%process and function declarations or definitions%
process P1 = ...;
process P2 = ...;

end;%module M%
```

A. Gamatié, *Designing Embedded Systems with the SIGNAL Programming Language: Synchronous, Reactive Specification*, DOI 10.1007/978-1-4419-0941-1_10,
© Springer Science+Business Media, LLC 2010

In the above syntax, a module is very similar to a class in object-oriented languages such as Java and C++. However, a major difference is that in a module attributes cannot be declared as in Java or C++ classes. Only types and constants can be defined in the body of the module besides processes and functions (which can be seen as the methods of a class).

For instance, the application executive (APEX) component library of POLYCHRONY, which is dedicated to the design of integrated modular avionic systems, includes a high number of services [3], enabling an application to gain access to the functionality of a real-time operating system. Figure 10.1 shows an extract of this library. The different modules shown contain the following elements:

- Services dedicated to the management of APEX *processes* (which are entities similar to UNIX threads)
- Services that allow processes to exchange messages via buffers
- All constants and types used by all the other modules of the library

A module is therefore a very useful structuring concept for large system descriptions based on component-oriented design.

10.2 Abstraction

Sometimes, a SIGNAL designer may need to import some external predefined functionality. Typically, this functionality can have been defined in another language, e.g., C, Java, or C++. It is captured in SIGNAL via the notion of *external processes*, introduced in Sect. 10.2.1. More precisely, such processes are specified as *black box* or *gray box* abstractions according to the level of detail the designer would like to make explicit. These abstractions are seen as SIGNAL specifications of a functionality rather than its programming. Sections 10.2.2 and 10.2.3, respectively, present these abstract views.

10.2.1 External Processes

SIGNAL allows one to abstract a process by an interface so that the process can be used afterwards as a black box through its interface. This interface describes parameters, input–output signals, and the clock and dependence relations between them. Subprocesses that are specified by an interface without the entire internal behavior are considered as external. They may be separately compiled processes or models of some devices, e.g., a sensor or an actuator.

Example 10.1 (An external process). In the following process specification, `Q` is considered as an external process in `P3`.

```
process P3 =
  { integer N; }
```

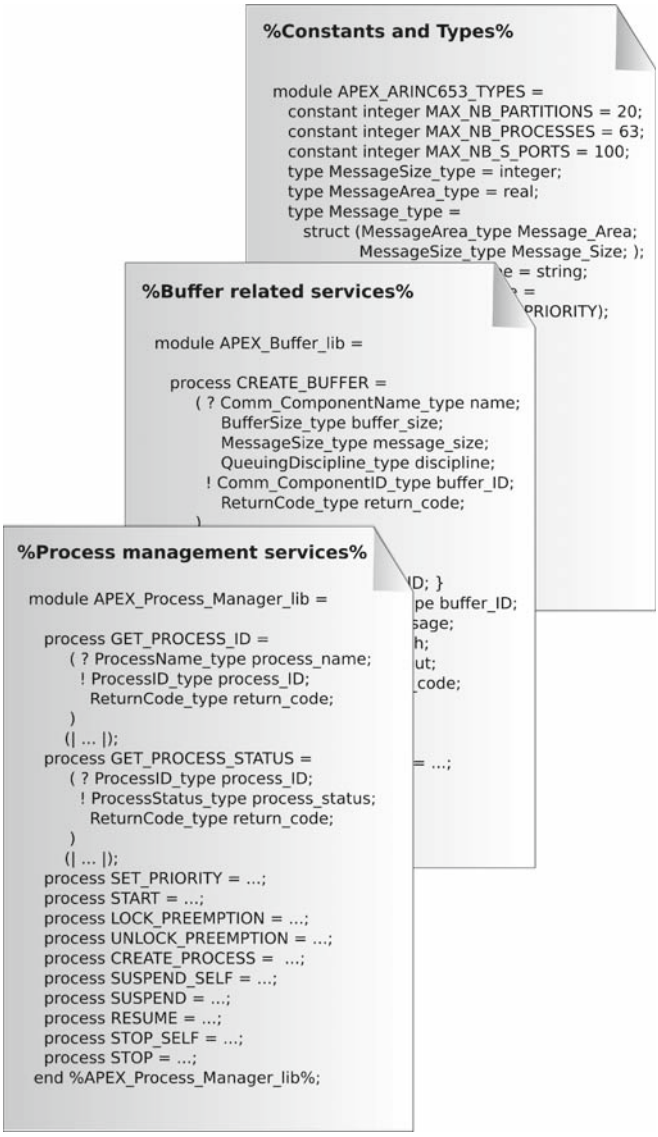


Fig. 10.1 An extract of the Application Executive library of POLYCHRONY

```
( ? integer s1;
    boolean ok;
  ! integer s2; )
(| tmp := Q{N}(s1)
 | s2:= tmp when ok
 |)
 where
   integer tmp;
```

```
    process Q =
      { integer M; }
      ( ? integer s1;
        ! integer s2; );
end; %end P3%
```

For a simulation of P3, the compiler will require the external code associated with Q to be integrated into the code generated automatically.

10.2.2 Black Box Model

In a component-based design, the level of detail required by the description of each component takes into account at least the specification of interface properties. These properties indicate how a component could be connected to other components. In some cases, this is enough to characterize the component, and to allow one to study behavioral properties of a global system including such components.

A SIGNAL description of a component can be obtained by "encapsulating" the component in a process model, which will thereafter be used as a black box. The interface of this process describes the inputs and outputs of the component. It also specifies possible dependency relations as well as clock constraints between inputs/outputs. Such a model, as shown in Fig. 10.2, is partially considered as external during the static analysis of the process since its internal definition is not known yet. Only interface properties are treated by the compiler.

In SIGNAL, the following notation expresses a precedence constraint between signals s1 and s2 within a logical instant:

```
s1 --> s2.
```

When this precedence is only valid at the instants of a given clock clk, the corresponding notation is as follows:

```
{s1 --> s2} when clk.
```

Example 10.2 (Black box specification).

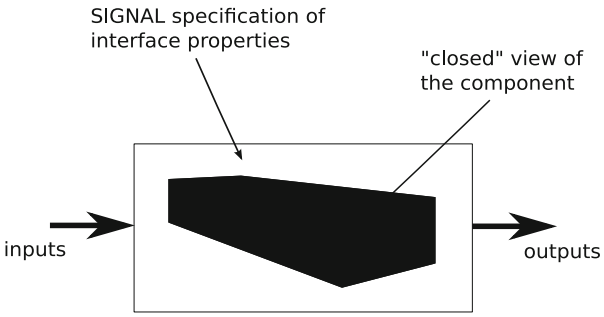


Fig. 10.2 Black box model of a component

```
process Q_black_box =
  { integer M; }
  ( ? integer s1;
    ! integer s2; )
 spec
  (| (| s1 --> s2 |)
   | (| s1 ^= s2 |)
   |); %Q_black_box%
```

In the black box abstraction `Q_black_box`, the input signal `s1` and the output signal `s2` are synchronous, and `s1` precedes `s2`; however, it is not described how `s2` is obtained from `s1`.

The above example is a simple illustration of using SIGNAL for specification rather than programming. Note the `spec` keyword (see also the SIGNAL grammar given on page 217), which introduces the clock and dependency properties specified on input and output signals. The implementation of the `Q_black_box` component, obtained via its programming, must satisfy these specified interface properties.

10.2.3 Gray Box Model

The main limitation of the black box model is that it does not allow a sharp analysis of the component by the compiler. As a result, if one wants to be able to reason more on the properties of a component, a richer abstraction is necessary.

Some analysis of the component can be done so as to split it into functionally coherent subparts. Such analysis primarily relies on dependency relations and constraints of synchronization between the signals manipulated in the process representing the component considered. Each of the subparts identified will itself be represented by a black box model. The interconnections between subparts and their activation clocks are clearly specified in the *gray box* model obtained. This is what is illustrated in Fig. 10.3, where the main component is composed of four subparts.

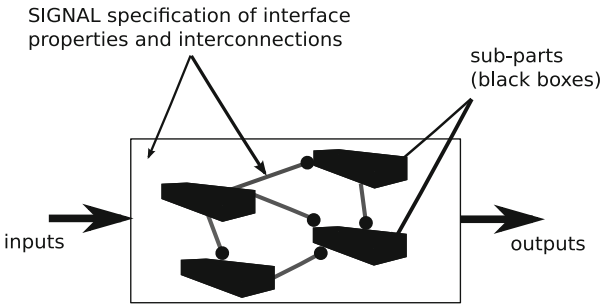


Fig. 10.3 Gray box model of a component

The gray box model offers a more detailed view than the black box model of a component; thus, it is more suitable for behavioral analysis. The lower the granularity of the subparts is, the more precise the component model is. The means that one can consider again gray box abstractions of each subparts and so on, until the desired level of detail is reached, typically when the main component model becomes a *white* box, i.e., a completely analyzable and executable specification.

Example 10.3 (Gray box specification).

```
process Q_grey_box =
{ integer M; }
( ? integer s1;
  ! integer s2; )
(| (| {s1 --> lab1} when (^s1)
    | lab1 --> lab2
    |)
 | (| lab1::(s3,s4):= black_box_1{M}(s1)
    | lab1 ^= when (^s1)
    | lab2::s2:= black_box_2{M}(s3)
    | lab2 ^= when (^s4)
    |)
 |)
where
 label lab1, lab2;
 integer s3; boolean s4;
 process black_box_1 =
  { integer M; }
  ( ? integer s1;
    ! integer s3;
      boolean s4;)
  spec
   (| (| {s1 --> s3} when s4
       | s1 --> s4
       |)
    | (| s1 ^= s4
       | s3 ^= when s4
       |)
    |); %black_box_1%
 process black_box_2 =
  { integer M; }
  ( ? integer s3;
    ! integer s2; )
  spec
   (| (| s3 --> s2 |)
    | (| s3 ^= s2 |)
    |); %black_box_2%
end; %Q_grey_box%
```

Here, the gray box abstraction shows that `Q_grey_box` is composed of two subcomponents `black_box_1` and `black_box_2` that exchange information. The labels `lab1` and `lab2` are used to specify synchronization and scheduling properties between the interfaces of the subcomponents.

The above example is another illustration of a specification defined with SIGNAL. The implementation of the `Q_grey_box` component will consist in defining completely all signals involved.

For systems defined by assembling components from different origins, also referred to as *intellectual properties*, box abstraction constitutes a suitable means to achieve their integration. The underlying formalism used for the integration is sometimes termed "glue" description language. For instance, the description of system architecture using the METAH avionics architecture description language [4] is based on this idea. External modules are encapsulated in black boxes specified in METAH. The behavior of the resulting system is analyzed on the basis of the properties extracted from encapsulated modules (e.g., execution time of the code associated with a module, scheduling policy adopted in a module).

10.3 Oversampling

A useful notion of SIGNAL is the *oversampling* mechanism. It consists of a temporal refinement of a given clock c_1, which yields another clock c_2, faster than c_1. By faster, it is meant that c_2 contains more instants than c_1.

Let us consider the following process called `k_Overspl`:

```
1: process k_Overspl =
2:   { integer k;}
3:   ( ? event c1;
4:     ! event c2; )
5:   (| count:= (k-1 when c1) default (pre_count-1)
6:    | pre_count:= count $ 1 init 0
7:    | c1 ^= when (pre_count <= 0)
8:    | c2:= when (^count)
9:    |)
10:   where
11      integer count, pre_count;
12:  end; %process k_Overspl%
```

This process illustrates an oversampling such that the output signal `c2` contains k instants per instant of the input signal `c1`. `k` is a constant integer parameter (line `2`). Signals `c1` and `c2`, respectively, denote the input and output of the process. The local signals `count` and `pre_count` serve as counters to define k instants in `c2` per instant in `c1` (lines `7` and `8`).

A corresponding trace where $k = 4$ is as follows:

c1 :	*tt* · · · *tt* · · ...
count :	3 2 1 0 3 2 1 ...
pre_count :	0 3 2 1 0 3 2 ...
c2 :	*tt tt tt tt tt tt tt tt* ...

The next example specifies the behavior of a refillable tank. Again, oversampling is used to achieve the description.

Example 10.4 (A simple model of a tank).

The SIGNAL model presented here describes the behavior of a refillable tank that has a maximum capacity [1]. The tank is filled only when it tends to become empty.

```
process Tank =
  { integer capacity; }
  ( ? event fill;
    ! event almost_empty;
  )
 (| fill ^= when((pre_n <= 2) and (pre_n >= 0))
  | pre_n := n$1 init 0
  | cnt := zn - 1
  | n := (capacity when fill) default cnt
  | almost_empty := when(n <= 2)
  |)
 where
   integer n, pre_n, cnt;
 end;
```

The body of the corresponding SIGNAL process can be explained as follows. Whenever the tank is filled, denoted by the occurrence of the input signal `fill`, the level of water in the tank, represented by `n`, starts to decrease until it reaches 0. The output signal `almost_empty` occurs whenever the level of water in the tank is critical, here meaning that it is less than or equal to 2. This warning signal therefore enables the tank to be refilled.

Observe that the input signal `fill` can be received only when the tank's status is critical, i.e., `pre_n` $\in [0, 2]$. In particular, the clock of the local signal `n` is greater in terms of the set of instants than those of the input and output signals of the process.

10.4 Assertion

The SIGNAL language defines so-called *intrinsic processes* [2], which are usable without declaration as with user-defined processes. The *assertion* belongs to this family of processes. It is particularly useful when specifying assumptions on input signals or guarantees on output signals of a process.

In the SIGNAL language, assertions are identified by the keyword `assert`. They can be specified as constraints either on Boolean signals or on clock relations.

10.4.1 Assertion on Boolean Signals

An assertion on a Boolean signal consists of a process with a single Boolean input and no output as follows:

```
process assert =
  ( ? boolean b;
    ! )
  (| b ^= when b |);
```

The meaning of this process is that the Boolean signal b must hold the value *true* whenever it occurs.

Example 10.5 (Assertion on Booleans). In the following process, the specified assumption says that signals x and y never occur at the same time, i.e., their associated clocks are exclusive.

```
process Exclusive_signals =
 ( ? integer x, y;
   ! )
 (| b:= (x ^* y) default false when (x ^+ y)
  | assert (not b)
  |)
 where
  boolean b
end;
```

10.4.2 Assertion on Clock Constraints

Assertions can also be used on clock constraints defined by using extended operators such as ^<, ^>, ^#, and ^=. In that case, they assert that the specified clock constraints are satisfied.

Example 10.6 (Assertion on clock constraints). In the following process, the specified assertion introduces an assumption on clock equivalence [2]:

```
process Two_oversampling =
 ( ? integer s1, s2;
   ! boolean b1, b2;
 )
 (| b1 := Oversampling(s1)
  | b2 := Oversampling(s2)
  | assert (| when b1 ^= when b2 |)
  |)
 where
  process Oversampling =
   (? integer s;
    ! boolean b;
   )
   (| z := s default v
    | v := (z $ init 1) - 1
    | b := v <= 0
    | s ^= when b
    |)
   where
     integer z, v;
   end;
end;
```

More examples on assertions could be found in the SIGNAL reference manual [2].

10.5 Exercises

10.1. Let A be an integer signal whose value is between 0 and 99. Define a process that produces for each value of A two integer digits, one representing the *tens*, and the other one representing the *units*. As an example, for values of A of 35; 7; 10, one expects 3; 5, 0; 7, 1; 0.

10.2. Let `n` be a positive or null integer signal. Define a process `N_Events` that generates a sequence of `n` events after each occurrence of `n`, and before its next occurrence. A decreasing counter can be used for the definition of this process.

Expected Learning Outcomes:

- The SIGNAL language proposes a **module** concept that enables one to define libraries of reusable components. This concept is particularly interesting when dealing with the design of large applications.
- A process can be specified either completely or only partially. In the latter case, the process may be typically described via its interface properties. The **black box** and **gray box** notions allow one to define such abstract views. However, for a functional simulation of these models, a corresponding implementation must be given in the chosen target general-purpose language.
- The **oversampling** mechanism is one of the outstanding features of SIGNAL. It is mainly used when one needs to specify faster clocks from slower clocks. Thus, it is very useful for the description of temporal refinements.
- **Assertions** can be explicitly defined in SIGNAL specifications when one needs to describe some assumptions in programs.

References

1. Beauvais J-R, Rutten É, Gautier T, Houdebine R, Le Guernic P, Tang YM (2001) Modeling STATECHARTS and activitycharts as SIGNAL equations. ACM Transactions on Software Engineering and Methodology (TOSEM) 10(4):397–451
2. Besnard L, Gautier T, Le Guernic P (2008) SIGNAL v4 – INRIA Version: Reference Manual. Available at: http://www.irisa.fr/espresso/Polychrony
3. Gamatié A, Gautier T (2002) Synchronous modeling of modular avionics architectures using the SIGNAL language. Research Report 4678, INRIA. Available at: http://www.inria.fr/rrrt/rr-4678.html
4. Vestal S (1997) MetaH support for real-time multi-processor avionics. In: IEEE Workshop on Parallel and Distributed Real-Time Systems

Chapter 11
GALS System Design

Abstract This chapter focuses on the design of globally asynchronous, locally synchronous (GALS) systems, which were mentioned in Chap. 3. The polychronous semantic model of the SIGNAL language has very interesting properties that favor a comfortable design of these systems. The implementation generated from a SIGNAL model of a distributed system can be trustworthily guaranteed to be correct by construction regarding the expected message exchanges within the system. This chapter presents some basic elements for the correct design of GALS systems. In SIGNAL, over the past decade, there have been significant studies on program distribution, mainly led by Benveniste and Le Guernic. This chapter is devoted to the results obtained from these studies and their applicability. Section 11.1 first indicates some application domains in which safe system distribution is crucial. Then, Sect. 11.2 presents some key notions that are usable to address system distribution issues. Finally, a methodology based on these notions is proposed in Sect. 11.3.

11.1 Motivation in Some Application Domains

There are different reasons to distribute a system: the performance enabled by the use of several computation units for a better response time, the geographical delocalization of the system elements, the replication of systems for fault tolerance, etc.

The deployment of distributed embedded systems often requires the interconnection of several subsystems, which are often heterogeneous, through a communication network. Among the typical application domains that are concerned, the following can be mentioned:

Transportation: The last generation of cars and aircraft have increasingly combined computers and electrical linkages to make more efficient and comfortable the control of the engine by a driver or a pilot. Such a combination leads to the so-called fly-by-wire systems, e.g., adopted in the Toyota Prius hybrid automobile or in the Airbus A380. Fly-by-wire systems allow designers to lower the weight of a car or an airplane while improving flexibly and reliability by reducing the complexity and fragility of mechanical circuits and hydraulic

A. Gamatié, *Designing Embedded Systems with the SIGNAL Programming Language: Synchronous, Reactive Specification*, DOI 10.1007/978-1-4419-0941-1_11,

© Springer Science+Business Media, LLC 2010

control systems that were widely used in previous car and aircraft generations. Moreover, they significantly reduce maintenance efforts. The safe interaction between the functionalities hosted on different computers in a fly-by-wire system is critical.

Telecommunication: Cellular phone manufacturers and the Internet face substantial problems related to distributed and mobile embedded systems deployed on a (wireless) network. These systems often need to interact to be able to provide users with the expected functionalities. The communication infrastructures considered should scale from small to large systems. Therefore, a major issue is how to achieve (1) compatibility between heterogeneous devices and (2) robust and transparent communications in a global network of embedded systems while guaranteeing the performance and real-time properties.

Industrial automation: This domain covers the manufacturing and power industry, industrial robots, and embedded mechatronic systems. The latter comprise technologies from engineering disciplines in computer software and hardware as well as in mechanics and automatic control. Distributed architectures are largely used for these systems. Among the relevant issues in this domain are the design of architectures for scalable and reconfigurable mechatronic systems and the design of embedded distributed control.

As an answer to the above needs, the distribution of an embedded system should follow well-defined methodologies that offer the necessary concepts, to clearly express the inherent concurrency of the system, and to validate its behavior with respect to both functional and nonfunctional properties. SIGNAL programming aims at contributing to this goal via the concepts [1, 2] introduced in the next chapter.

11.2 Theoretical Foundations

The multiclocked nature of SIGNAL enables one to model systems in which each component can have its own activation clock. Such a model is very attractive for the description of globally asynchronous, locally synchronous (GALS) systems. However, since these systems include asynchronous mechanisms, e.g., for communication, it is important to be able to ensure the equivalence of (1) the behavior of their corresponding SIGNAL models and (2) the behavior of their asynchronous implementations.

Figure 11.1 depicts a trace from both synchronous and asynchronous viewpoints. In the synchronous view (on the top), the synchronization between observed events is represented via the $\perp$ symbol, which means absence. In the asynchronous view, such synchronization information does not exist; only the order of the event's values is preserved.

To reason about the equivalence of these different viewpoints, two basic notions have been identified in the polychronous model: *endochrony* (Sect. 11.2.1) and *endo-isochrony* (Sect. 11.2.2) [8]. The denotational semantic model presented in Chap. 7 is used to formally characterize these notions.

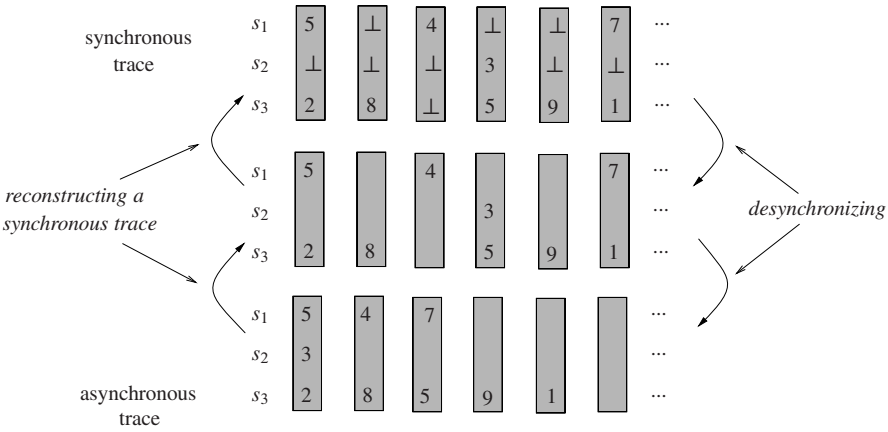


Fig. 11.1 From synchronous to asynchronous and back

11.2.1 Endochrony

In Chap. 9, we saw that the compilation of a SIGNAL program can lead to two kinds of clock hierarchies: endochronous and exochronous clock structures. The former hierarchy consists of a single tree of clocks. The clocks are organized according to the set inclusion. The largest set of instants, denoting the fastest clock of a process, is on the root node of the clock tree. The latter hierarchy consists of several clock trees. The reaction of the associated program depends on the environment, which decides which clock trees should be activated on each reaction step. However, as was shown in Chap. 9, an exochronous program can be transformed into an endochronous program.

Thanks to the hierarchy defined in a clock tree, an endochronous process knows, from a given asynchronous flow of significant values, i.e., different from $\perp$, how to reconstruct exactly the flow of its inputs and outputs with the adequate synchronization information via the use of the signal absence information $\perp$. This is illustrated in Fig. 11.1 where an asynchronous trace at the bottom part of the figure is resynchronized into a synchronous trace shown at the top of the figure. For instance, the first value of signal s_2, which is received at the same time as those of signals s_1 and s_3 in the asynchronous trace, is effectively read only on the fourth logical instant in the synchronous trace.

Before the formal definition of the endochrony property, we first introduce the notion of *behavior relaxation*, which is very useful to compare flows of values according to an asynchronous viewpoint.

Remember that behavior stretching induces an equivalence relation that preserves at the same time the simultaneity and the occurrence order of events within behaviors (see Chap. 7, page 104). In other words, considering two different behaviors b_1 and b_2, if b_1 represents a stretching of b_2, noted $b_2 \preceq_{\mathcal{B}_\mathcal{T}} b_1$, then both b_1 and b_2

share the same synchronization relations between their common signals. Behavior relaxation rather induces a less constrained equivalence relation than stretching in the sense that if b_1 results from the relaxation of b_2, then b_1 does not necessarily preserve the initial synchronization relations in b_2; *however, the order of events is identical in both* b_1 *and* b_2.

Thus, relaxation makes it possible to compare behaviors according to observed values, i.e., their functional properties. This is particularly well suited to characterize the desynchronization of synchronous specifications. The deployment of such specifications on a distributed architecture usually requires the relaxation of certain behaviors.

Definition 11.1 (Behavior relaxation). For a given set of observation points $\mathcal{T}$, a behavior b_1 is a relaxation of a behavior b_2, noted $b_1 \sqsubseteq b_2$, iff

$$\text{vars}(b_1) = \text{vars}(b_2) \quad \text{and} \quad \forall x \in \text{vars}(b_1),\ b_{1|\{x\}} \leq_{\mathcal{B}_\mathcal{T}} b_{2|\{x\}}.$$

The relaxation relation induces a partial order on behaviors. It defines a *flow equivalence* of different behaviors, i.e., an equivalence of observed value sequences. Intuitively, two behaviors are said to be *flow-equivalent* iff they have the same tag domain, and the observed values of their signals occur in the same order.

Definition 11.2 (Flow equivalence). For a given set of observation points $\mathcal{T}$, two behaviors b_1 and b_2 are flow -equivalent, noted $b_1 \approx b_2$, iff there exists a behavior b_3 such that

$$b_3 \sqsubseteq b_1 \quad \text{and} \quad b_3 \sqsubseteq b_2.$$

The equivalence class of a behavior b according to the flow-equivalence relation forms a semilattice admitting a strict behavior, noted $b_{\approx}$, i.e., the representative element of this class.

The flow equivalence is a minimal correction criterion for the refinements of a strict synchronous specification on an asynchronous architecture. Indeed, the behaviors associated with this specification and those resulting from its transformation towards an asynchronous execution must have the occurrences of their signal values in the same order to preserve the functional properties.

Using the above notions, let us define now the endochrony property.

Definition 11.3 (Endochrony). A process p is endochronous on a (possibly empty) subset $E \subseteq \text{vars}(p)$ iff

$$\forall b, b' \in p,\ (b_{|E})_{\approx} = (b'_{|E})_{\approx} \Rightarrow b \lesssim b'.$$

The process p is said to be endochronous if it is endochronous on the whole set of its inputs.

In the above definition, the left-hand side of the implication specifies that the behaviors b and b' of process p have the same class of flow equivalence, on signals

associated with the variables of E. In other words, regarding these signals, the order of the actual values in b is identical to that in b'; only the synchronization constraints may differ.

Saying that p is endochronous simply amounts to considering that b and b' are equivalent with respect to stretching. Indeed, if there are several flows with identical values, the behaviors produced by an endochronous process p will inevitably have the same synchronization constraints since the read/write instants of the values present in the flows are a priori uniquely determined thanks to the endochrony property. These constraints are induced by the clock tree associated with p.

From the above observation, the behaviors b and b' hold identical flows of values and synchronizations. They are therefore equivalent with respect to stretching: this is expressed by the right-hand side of the implication.

The importance of determinism in real-time systems was mentioned in Chap. 1: it facilitates the study of the behaviors of the system since they are predictable. In the multiclocked model, the determinism of a process is characterized by the fact that its behaviors depend entirely at any execution instant on the status and values of the inputs of this process.

Definition 11.4 (Determinism on a subset of variables). A process p is deterministic on a subset $E \subseteq \text{vars}(p)$ iff for all $b_1, b_2 \in p_{\downarrow}$ (the set of strict behaviors associated with p), for all $t \in \text{tags}(b_1) \cap \text{tags}(b_2)$:

$$(b_{1|E})|_{\leq t} = (b_{2|E})|_{\leq t} \Rightarrow (b_1)|_{\leq t} = (b_2)|_{\leq t},$$

where the notation $b|_{\leq t}$ means the prefix of the behavior b until the tag $t \in \text{tags}(b)$.

Definition 11.5 (Determinism). The process p is said to be deterministic if it is deterministic on the whole set of its inputs.

All SIGNAL elementary processes are deterministic (where all variables of the signal involved are distinct). In contrast, the elementary process `s2 := s1 default s2` is not deterministic since it only partially defines `s2`: when `s1` is not present, the value of `s2` is free.

The following property holds for the determinism of endochronous processes [8]:

Property 11.1. An endochronous process is deterministic.

Endochrony is an adequate criterion to study the equivalence between internal (synchronous) behavioral observations and external (asynchronous) behavioral observations in a system.

A deterministic and nonendochronous process can be straightforwardly extended to make it endochronous. This is achieved by considering additional Boolean signals that are used to characterize the clocks of the already defined signals (see also Chap. 9, page 136). These Boolean signals are made synchronous.

Example 11.1 (Endochronization of a deterministic process). Let us consider the elementary process `s3 := s1 default s2` defined with the deterministic merging operator. An endochronous version of the same process can be defined as follows:

```
1:  process Endochronous_merging =
2:      ( ? boolean clk_s1, clk_s1;
3:          integer s1, s2;
4:        ! integer s3;
5:       )
6:      (| s3 := s1 default s2
7:       | clk_s1 ^= clk_s2
8:       | s1 ^= when clk_s1
9:       | s2 ^= when clk_s2
10:      |)
```

where the Boolean signals `clk_s1` and `clk_s2` are introduced as new inputs of the process. These signals have the same clock. The instants at which they carry the value `true` characterize the clocks of `s1` and `s2`.

11.2.2 Endo-isochrony

The endo-isochrony property defines the conditions under which a synchronous composition is equivalent to an asynchronous composition, which involves *send/receive* communications. Asynchronous communications between endoisochronous processes can be entirely replaced by synchronous models of these communications.

The processes p and p', which are composed synchronously in Fig. 11.2, on the left-hand side, are both endochronous. The restriction of their behaviors to the whole set of their common variables, referred to as E, is also endochronous: p and p' are said to communicate endoisochronously.

These processes can be safely deployed on a GALS architecture while preserving the behaviors resulting from their synchronous composition. For that, each of them is provided with an endoisochronous communication interface, represented by a duplication of E in Fig. 11.2, on the right-hand side. These interfaces enable one to suitably exchange all required information. The instants at which this information is gotten are easily deduced owing to the fact that E is endochronous.

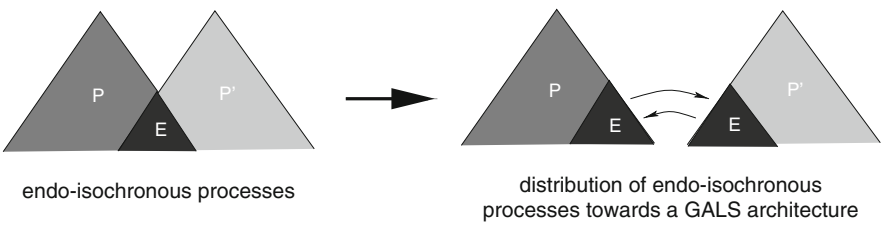


Fig. 11.2 Endoisochronous processes and globally asynchronous, locally synchronous (*GALS*) distribution

Before defining the endo-isochrony property, let us describe the meaning of asynchronous composition.

Definition 11.6 (Asynchronous composition). Given a behavior b that belongs to a process p such that $X = \text{vars}(p)$, and a behavior b' of another process p' such that $X' = \text{vars}(p')$, the asynchronous composition $p \parallel p'$ consists of relaxed behaviors b'' on signals that are common to b and b', i.e., $E = X \cap X'$:

$$p \parallel p' = \{\, b'' \mid \exists (b, b') \in p \times p' \\ b_{|X \setminus X'} \lesssim b''_{|X \setminus X'} \wedge b_{|E} \sqsubseteq b''_{|E} \wedge b''_{|X' \setminus X} \lesssim b'_{|X' \setminus X} \wedge b'_{|E} \sqsubseteq b''_{|E} \,\}.$$

In the above definition, let us recall that $\lesssim$ and $\sqsubseteq$, respectively, denote stretch equivalence and relaxation.

In addition, a notion of *flow invariance* is defined, which makes it possible to ensure that the refinement of a synchronous specification $p \mid p'$ in an asynchronous implementation $p \parallel p'$ preserves the flow of signal values in any behavior.

Definition 11.7 (Flow invariance). The composition of two processes p and p' is flow-invariant iff

$$\forall b \in p \mid p', \forall b' \in p \parallel p', (b_{|E})_{\approx} = (b'_{|E})_{\approx} \Rightarrow b \approx b',$$

where E represents the input signals of $p \mid p'$.

We can now define the *endo-isochrony* property, which is the other basic notion with endochrony that is necessary to design distributed applications according to a compositional approach, starting from their synchronous multiclocked specifications.

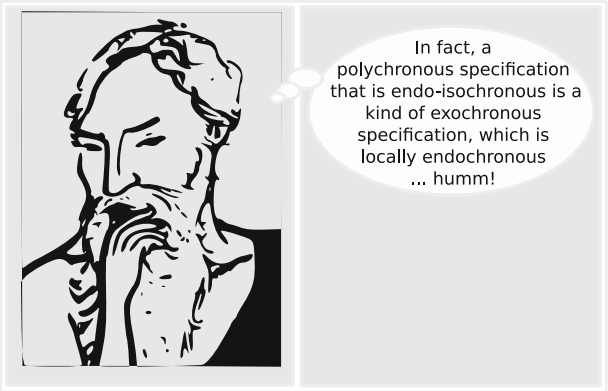
Definition 11.8 (Endo-isochrony). Two endochronous processes p and p' are endo-isochronous[1] iff

$$(p_{|E}) \mid (p'_{|E}) \text{ is endochronous,}$$

where $E = \text{vars}(p) \cap \text{vars}(p')$.

Endo-isochrony is a more restrictive version of the general *isochrony* criterion [3], which states that the synchronous composition of a pair of isochronous processes is equivalent to the asynchronous composition of these processes. Endo-isochrony imposes on any pair of processes that the restriction on their "communication subpart" be endochronous. It is a sufficient but not necessary criterion to have an isochronous pair of processes. The definition of endo-isochrony is more constructive.

[1] By abuse of language, one also speaks of endoisochronous communications between p and p'.



Example 11.2 (Endoisochronous processes). Let us consider the following processes P1 and P2:

```
process P1 =
  { integer init_val, emit_val, t; }
  ( ?
    ! integer s; )
  (|(| cnt:= ((init_val-1) when reset) default (pre_cnt-1)
     | pre_cnt := cnt $ 1 init 0
     | reset := when (pre_cnt <= 0)
     |)
   |(| s := emit_val when (pre_cnt = t) |)
   |)
  where
    integer cnt, pre_cnt;
    event reset;
end;

process P2 =
  { integer init_val, N, t; }
  ( ? integer s
    ! integer s1)
  (|(| cnt:= (init_val-1 when reset) default pre_cnt-1
     | pre_cnt := cnt $ 1 init 0
     | reset := when (pre_cnt <= 0)
     |)
   |(| s ^= when (pre_cnt = t)
     | s1 := s + init_val
     |)
   |)
 where
   integer cnt, pre_cnt;
   event reset;
end;
```

Process P1 periodically emits some output value. The period is defined by the static parameter t. An internal counter, denoted by the signal cnt, is considered to decide when the output value must be produced. The definition of P1 adopts the

oversampling mechanism presented in Chap. 10 (page 155). The same remark also holds for process `P2`, which receives the values produced by `P1`, and computes its output values.

Both processes `P1` and `P2` are endochronous. In each process, the master clock is that of the signal `cnt`. These processes are also endoisochronous because they only communicate via the signal `s`. The restriction of their synchronous composition `P1 | P2` to the set of signals {`s`} is trivially endochronous.

Another example of endoisochronous design is given in Chap. 12.

11.3 A Distribution Methodology

The design of a system on a GALS architecture, using the multiclock model of SIGNAL, consists in modeling the system as a set of endochronous components that communicate in an endoisochronous way. Endochrony and endo-isochrony can be checked statically with the SIGNAL compiler on the basis of clock hierarchies.

The design methodology implemented currently in POLYCHRONY for distributed embedded systems with the SIGNAL language is sketched in Fig. 11.3. It is decomposed into four basic steps that are detailed below chronologically.

Step 1: Specification of the system distribution. The application is first described as a hierarchical SIGNAL process $p = p_1 \mid \ldots \mid p_n$. Such a process is graphically represented in the graphical editor of POLYCHRONY as a network (or a graph) of interconnected boxes. Each box contains the SIGNAL code of a subprocess p_i. Thus, process p reflects the *logical* architecture of the application. At this stage, some preliminary verification can be done on process p to make sure that it conforms to the expected application requirements.

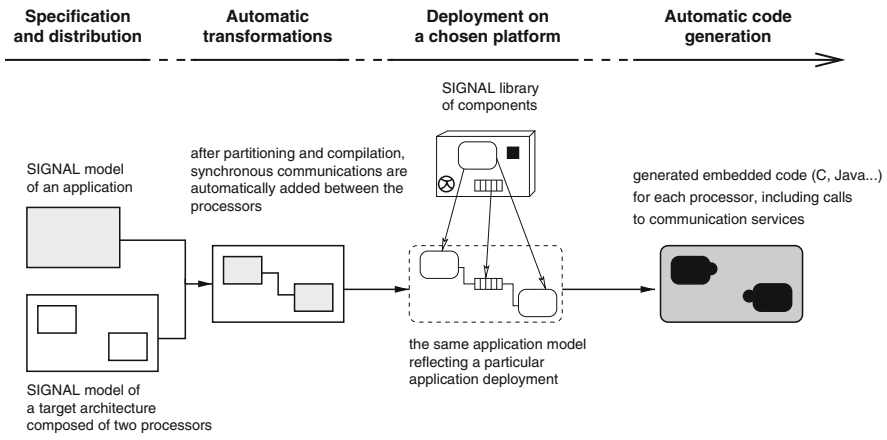


Fig. 11.3 A design methodology for distributed systems in SIGNAL

Afterwards, the *hardware* architecture on which p will be mapped is defined. Here, only processors are described (it is a simplified model). Concretely, each processor is represented as an empty box.

Now, one can proceed to the mapping of the logical graph specifying the application, i.e., process p, on the hardware graph, i.e., the architecture representation. The SIGNAL code associated with a logical box can be duplicated in several boxes denoting processors; so, code replication is allowed. From this mapping, a new description of the application $p' = p'_1 \mid \ldots \mid p'_m$ is obtained, which reflects the target hardware architecture (m denotes the number of processors). We can notice that the subprocesses p'_j are not necessarily endochronous. For instance, after the mapping, a process p'_j can be obtained from the composition of different subprocesses p_i characterized by different clock trees without a common root clock.

Step 2: Automatic transformations guaranteeing the functional correctness of the distribution. The main idea consists in transforming the above process p' so that (1) p' preserves the functional semantics of p and (2) the subprocesses p'_j become endochronous and p' itself becomes endoisochronous.

Each process p'_j is *compiled modularly in the context of* p: from the clock hierarchy of p, the clock hierarchy of p'_j is transformed in such a way that it becomes a clock tree, i.e., p'_j is endochronous. For that, Boolean interface signals are used (each root node in a clock hierarchy of a nonendochronous process is associated with a Boolean signal, and all these signals are made synchronous). On the other hand, some transformations are performed to render the communication part endochronous between the endochronous processes p'_j. For that, additional communicating signals are inserted between processes, which enable each process p'_j to receive all information it requires to execute. The result of these transformations is carried out automatically on the graphical representation of p' within the POLYCHRONY editor. In the transformed process p'' obtained, information exchanges between processes are described by instantaneous communications.

At this stage, some possible verification techniques are again applicable to p'' so as to ensure that it still preserves the functional properties of p.

Step 3: Deployment on specific platforms. To refine the descriptions obtained from the previous steps of the methodology, a set of predefined components is proposed, which is usable to instantiate various aspects of the system (see Fig. 11.3): asynchronous communication mechanisms such as the FIFO buffer modeled in Chap.12, execution supports (e.g., tasks and processes), system-level primitives for task management, as well as for synchronization and intertask communication services (e.g., the Application Executive service models [6]). These components have been modeled in SIGNAL and are grouped in a library that is made available in the POLYCHRONY platform.

Step 4: Automatic generation of a distributed code. The description obtained at the previous step can now be considered for a modular automatic code generation. Since, each "processor" is endochronous, the associated code can be generated independently from the action of the other processors. The same holds for

communications. This generation is necessarily achieved in the form of *clusters* [7, 5]. In SIGNAL, a cluster denotes a group of statements that depend on the same subset of input data. This type of code generation is particularly interesting in that it enables an efficient execution for each processor. The code corresponding to clusters is generated in C, C++, or Java.

In [4], the reader can find a case study where the distribution of an application follows the above design methodology.

11.4 Exercises

11.1. Let us consider the following processes. Discuss the determinism and the endochrony of each of them.

```
1.      (| s2:= s1-2
         | s3:= s2 $ 1 init 0
         |)
2.      (| s2:= s1*3
         | s3:= s2-4 when s2 >=0
         | s4:= s3 when s3>0
         |)
3.      (| s2:= s1 $ 1 init 0
         | s1:= s3 default s2-1
         | s3 ^= when s2>0
         |)
4.      (| s1:= s2 default s3 |)
5.      (| s2 ^< s1 |)
```

Expected Learning Outcomes:

- The polychronous model of SIGNAL allows for the **specification of distributed asynchronous embedded systems**.
- The design of GALS systems in SIGNAL relies on two key notions
  - **Endochrony**: The ability of a process to reconstruct a synchronized behavior from a flow of values received asynchronously
  - **Endo-isochrony**: The equivalence between the synchronous and asynchronous compositions
- A basic **design methodology** for these systems consists of the following steps
  1. Definition of a synchronous model of a system
  2. Checking/enforcing endo-isochrony on the model
  3. Insertion of synchronous models of asynchronous communication mechanisms
  4. Automatic code generation

References

1. Benveniste A (1998) Safety critical embedded systems: the SACRES approach. In: Proceedings of Formal techniques in Real-Time and Fault Tolerant Systems, FTRTFT'98 School, Lyngby, Denmark
2. Benveniste A, Caillaud B, Le Guernic P (1999) From synchrony to asynchrony. In: International Conference on Concurrency Theory, pp 162–177
3. Benveniste A, Caillaud B, Le Guernic P (2000) Compositionality in dataflow synchronous languages: specification and distributed code generation. Information and Computation 163: 125–171
4. Besnard L, Bournai P, Gautier T, Halbwachs N, Nadjm-Tehrani S, Ressouche A (2000) Design of a multi-formalism application and distribution in a data-flow context: an example. In: Gergatsoulis M, Rondogiannis P (eds) Intensional Programming II, Based on the Papers at the 12th International Symposium on Languages for Intentional programming (ISLIP'99), World Scientific, pp 8–30
5. Besnard L, Gautier T, Talpin J-P (2009) Code generation strategies in the POLYCHRONY environment. Research report number 6894, INRIA. Available at: http://hal.inria.fr/docs/00/37/24/12/PDF/RR-6894.pdf
6. Gamatié A, Gautier T, Le Guernic P, Talpin J-P (2007) Polychronous design of embedded real-time applications. ACM Transaction on Software Engineering Methodology 16(2):9
7. Gautier T, Le Guernic P (1999) Code generation in the SACRES project. In: Safety-critical Systems Symposium, SSS'99, Springer, Huntingdon, UK
8. Le Guernic P, Talpin J-P, Le Lann J-C (2003) Polychrony for system design. Journal for Circuits, Systems and Computers 12(3):261–304

Chapter 12
Design Patterns

Abstract This chapter presents some examples in SIGNAL that could be considered as design patterns. Through these examples, several useful programming concepts of the language, described throughout the book, are shown. An important goal is to help the reader choose which concepts are best suited for the definition of a solution to a given system design problem in SIGNAL. Sections 12.1 and 12.2 first present two kinds of design approaches: *top-down* and *bottom-up*. The former deals with the refinement of high-level specifications into more precise ones, whereas the latter promotes reusability. Both approaches are useful when dealing with the design of complex systems. Section 12.3 focuses on a few modeling problems that are very frequent in embedded systems. Section 12.4 illustrates an interesting situation where the clock oversampling mechanism of SIGNAL plays a key role. Finally, Sect. 12.5 presents a more elaborate example in which the endo-isochrony property is enforced by construction in the proposed SIGNAL model.

12.1 Refinement-Based Design (Top-Down)

The design of a system following a *top-down* approach consists in deriving, from a high-level specification, a more particular specification, which is, for instance, more constrained in terms of behaviors. In general, this refinement involves the identification of the different subsystems and the required communication mechanisms between them; the definition of the operations associated with these subsystems; and the encoding of the identified algorithms into programs.

In SIGNAL, a top-down modeling of a component consists in considering first an abstract description of this component, which thereafter is *refined* to derive a more precise description. The refinement process is reiterated until the desired final model is made explicit. On the basis of this principle, components are initially described by only specifying their interface properties as in black box abstractions (see Chap. 10, page 152). The next sections illustrate a top-down approach for the modeling of a communication service.

A. Gamatié, *Designing Embedded Systems with the SIGNAL Programming Language: Synchronous, Reactive Specification*, DOI 10.1007/978-1-4419-0941-1_12,
© Springer Science+Business Media, LLC 2010

12.1.1 A Blackboard Mechanism

Different standards for software and hardware design have been defined for integrated modular avionics architectures. The application executive (APEX) Aeronautical Radio Inc. (ARINC) 653 standard [1] proposes an application programming interface enabling the interaction between the application and the operating system in integrated modular avionics systems. This application programming interface includes services for the communication between executive entities.

The modeling approach adopted here is illustrated by considering the *read_blackboard* APEX service, which enables messages to be displayed and read in a blackboard. The input parameters of this service are the blackboard *identifier* and a *time-out* duration that denotes the maximum waiting time on a request when a blackboard is empty. The output parameters are a *message*, defined by its address and size, and a *return code* for the diagnostics of the service request. An informal specification of the *read_blackboard* service is as follows:

```
If inputs are invalid (that means the blackboard identifier is
    unknown or the time-out value is ''out of range'') Then
  Return INVALID_PARAM (as return code);
Else If a message is currently displayed on the specified
         blackboard Then
       send this message and return NO_ERROR;
Else If the time-out value is zero Then
       Return NOT_AVAILABLE;
Else If preemption is disabled or the current process is the
         error handler Then
       Return INVALID_MODE;
Else
       set the process state to waiting;
       If the time-out value is not infinite Then
          initiate a time counter with duration time-out;
       EndIf;
       ask for process scheduling (the process is blocked and
        will return to ''ready'' state by a display service
        request on that blackboard from another process or
        time-out expiration);
       If expiration of time-out Then
          Return TIMED_OUT;
       Else
          the output message is the latest available message
           of the blackboard;
          Return NO_ERROR;
       EndIf;
EndIf
```

In the following, the modeling of the *read_blackboard* service is presented through different steps; from one step to another, the SIGNAL description is progressively detailed.

12.1.2 SIGNAL Modeling of read_blackboard

12.1.2.1 Initial Model Specification

The process defined in Fig. 12.1 gives an abstract formal specification corresponding to the *read_blackboard* service:

This specification mainly expresses the interface properties of the service model. For example, line `14` specifies the logical instants at which a return code is produced. The variable `C_return_code` is a local Boolean signal that carries the value *true* whenever a return code is received on a read request, in other words, the Boolean signal `C_return_code` represents the clock of the return code signal. Indeed, on a *read_blackboard* service request, a return code retrieval is not systematic: when the *blackboard* is empty and the value of the input parameter `timeout` is not infinite,[1] the requesting process is suspended. The signal *C_return_code* therefore carries the value *false*. The suspended process must wait: either a message is displayed on the *blackboard*, or the expiration of the active time counter is initialized with `timeout`.

For the moment, `C_return_code` appears in the *read_blackboard* description as a local signal. It will be defined after a refinement of the initial service model description. At this stage, we assume that there only exist signals such that properties in which it is involved are satisfied. The statement at line `13` states that `C_return_code` and all input parameters are synchronous. So, whenever there is

```
1: process Read_Blackboard =
2:    { ProcessID_type process_ID; }
3:    ( ? Comm_ComponentID_type board_ID;
4:        SystemTime_type timeout;
5:      ! MessageArea_type message;
6:        MessageSize_type length;
7:        ReturnCode_type return_code; )
8:    (| (| {{board_ID, timeout} --> return_code}
9:                          when C_return_code
10:       | {{board_ID, timeout} --> {message, length}}
11:                         when (return_code = #NO_ERROR)
12:       |)
13:    | (| board_ID ^= timeout ^= C_return_code
14:       | return_code ^= when C_return_code
15:       | message ^= length ^= when (return_code = #NO_ERROR)
16:       |)
17:    |)
18:   where
19:      boolean C_return_code;
20:   end;%process Read_Blackboard%
```

Fig. 12.1 A specification of the *read_blackboard* service

[1] Infinite is represented in [1] by a special constant *INFINITE_TIME_VALUE*.

a read request, C_return_code indicates whether or not a return code should be produced. The statement at line 15 expresses the fact that the messages are obtained on a read request only when the return code value is NO_ERROR.

The statements specified in lines 8 and 10 describe the data dependency relations between the input and output parameters of the service. For instance, line 10 states that message and length depend on timeout and board_ID, at the logical instants where the return code carries the value NO_ERROR.

The level of detail provided by this first SIGNAL description of the *read_blackboard* service is expressive enough to check, for instance, the interface conformance with respect to data dependencies and clock relations of such a component during its integration within a larger system. In particular, for read requests, this model defines the conditions under which a message can be retrieved. However, it does not define **how** the retrieved messages are obtained from the blackboard.

In the next step, the initial model is refined by defining more behavioral properties.

12.1.2.2 Refined Model Specification

A more detailed version of the *read_blackboard* service is now illustrated in Fig. 12.2. In this refined model, the interface properties specified in the initial model still hold even though they are not explicitly mentioned. The new model gives more details about the service since internal properties are partially described in addition to interface properties.

Four main subparts are distinguished on the basis of the service informal specification. They are graphically represented in Fig. 12.3, which results from the POLYCHRONY graphical user interface. They are represented by inner boxes in this graphical representation. The CHECK_BOARD_ID and CHECK_TIMEOUT subparts verify the validity of input parameters board_ID and timeout. If these inputs are valid, PERFORM_READ tries to read the specified blackboard. Afterwards, it has to send the latest message displayed on the blackboard. The area and size of the message are specified by message and length. The subpart PERFORM_READ also transmits all the necessary information to the rightmost subpart in Fig. 12.3, referred to as GET_RETURN_CODE, which defines the final diagnostic message of the service request.

Each identified subpart is only shown via its interface. Its internal properties are defined later. Some relations can be specified between the interface signals of these subparts. For instance, in Fig. 12.2, the equations defined at lines 8, 15, 16, 17, and 18 express synchronization relations between the different signals.

In addition to these equations, the Boolean signal C_return_code, which was only declared in the previous step, is now defined since one has enough information to determine its values. This is described by the equation specified at line 23 in Fig. 12.2.

```
process Read_Blackboard =
  { ProcessID_type process_ID; }
  ( ? Comm_ComponentID_type board_ID;
      SystemTime_type timeout;
    ! MessageArea_type message;
      MessageSize_type length;
      ReturnCode_type return_code; )
  (| (| board_ID ^= timeout ^= present ^= outofrange ^=
          available ^= C_return_code |)
   | (| (present, board) := CHECK_BOARD_ID(board_ID) |)
   | (| (outofrange, available):= CHECK_TIMEOUT(timeout) |)
   | (| (message, length, is_err_handler, empty,
         preemp_enabled) := PERFORM_READ(board, board_ID,
                  timeout, available, outofrange)
     | preemp_enabled ^= when (not is_err_handler)
     | is_err_handler ^= when empty when available
     | message ^= length ^= when (not empty)
     | board ^= empty ^= when present
     |)
   | (| return_code := GET_RETURN_CODE(present,
          is_err_handler, empty, preemp_enabled,
                   outofrange, available)
     | C_return_code:= (when ((not present) or outofrange))
       default (when empty when (not available))
       default (when((not preemp_enabled)
                                default is_err_handler))
       default (when (not empty))
       default false
     | return_code ^= when C_return_code
     |)
  |)
 where
   APEX_Blackboard_type board;
   boolean C_return_code, present, outofrange,
           available, preemp_enabled, empty,
              is_err_handler;
end%process Read_Blackboard%;
```

Fig. 12.2 A refined model of the *read_blackboard* service

By iteration of the above refinement process, the internal properties of each identified subpart are progressively detailed. Finally, a complete SIGNAL specification of the service can be obtained as illustrated in [2].

The above top-down approach consisting in transforming progressively black box specifications into white box descriptions (via intermediate gray box specifications) shows how SIGNAL offers a way to refine high-level specifications into executable implementations of system components.

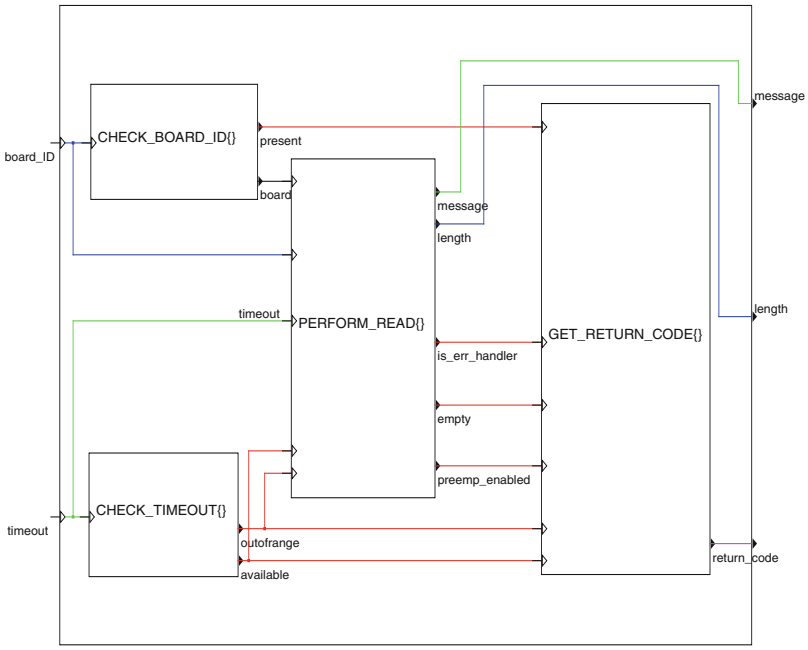


Fig. 12.3 Graphical view of the refined *read_blackboard* model

12.2 Incremental Design (Bottom-Up)

In the *bottom-up* approach, a system is built via the combination of basic blocks, such as the components (or intellectual properties) of a library. These components are to be combined so as to define new components, which can themselves be combined, until the system description desired by the designer is obtained. The bottom-up approach is therefore incremental.

The modularity of the SIGNAL language favors such an approach. Let us consider a component C, modeled by a SIGNAL process P as follows:

$$P = P_1 \mid \ldots \mid P_n,$$

where each P_i ($i \in [1..n]$) indicates the process associated with a subcomponent C_i of C. A possible construction scenario starts with the definition of P_1. Then, P_2 is defined and composed with P_1 to obtain $P_{1,2}$. In a similar way, after the definition of P_3, it is composed with $P_{1,2}$ to define $P_{1,2,3}$, and so on. The same composition process is reiterated until one obtains $P_{1,\ldots,n}$, which models the component C.

In Sect. 12.2.2, a bottom-up approach is illustrated, where a FIFO message queue is defined in SIGNAL.

12.2.1 A FIFO Message Queue

Let us consider a FIFO message queue, called *basic FIFO*, that works as follows:

- On a write request, an incoming message is inserted in the queue regardless of its size limit. If the queue was previously full, the oldest enqueued message is lost. The other messages are shifted forward, and the new incoming message is put in the queue.
- On a read request, there is an outgoing message whatever the queue status is. If it was previously empty, two situations are distinguished: if there had not been any written message, an arbitrary message called *default message* is returned; otherwise the outgoing message is the message that was read last.

In the basic FIFO message queue, it is supposed that write/read requests never occur simultaneously.

12.2.2 SIGNAL Modeling of the FIFO Queue

12.2.2.1 Initial Model Design

A SIGNAL description corresponding to the basic FIFO message queue is detailed below.

```
1: process basic_FIFO =
2:   { type message_type;
3:     integer fifo_size,
4:     message_type default_msg; }
5:   ( ? message_type msg_in; event access_clk;
6:     ! message_type msg_out; integer nbmsg;
7:       boolean OK_write, OK_read;  )
```

The static parameters `message_type`, `fifo_size`, and `default_msg` denote the type of messages, the size limit of the queue, and a default message value, respectively. The input signals `msg_in` and `access_clk` are, respectively, the incoming message whose presence denotes a write request, and the queue access clock, i.e., the instants of read/write requests. The output signals are `msg_out`, `nbmsg`, `OK_write`, and `OK_read`. They represent, respectively, the outgoing message, the current number of messages in the queue, and the conditions under which writing and reading are possible.

The next statements define the body of the `basic_FIFO` process.

```
8: (| prev_nbmsg := nbmsg$1 init 0
9:  | OK_write := prev_nbmsg<fifo_size
10: | OK_read := prev_nbmsg>0
```

The equation specified at line `8` defines the local signal `prev_nbmsg`, which denotes the previous number of messages in the queue. This signal is used in the

statements at lines 9 and 10 to define, respectively, when the queue can be "safely" written, i.e., the size limit is not reached, and read, i.e., there is at least one retrievable message. This is the meaning of the signals OK_write and OK_read.

```
11: | nbmsg := ((prev_nbmsg+1) when (^msg_in) when OK_write)
        default ((prev_nbmsg-1) when (^msg_out) when OK_read)
        default prev_nbmsg
12: | nbmsg ^= access_clk
```

The equation at line 11 expresses how the current number of messages is calculated:

- Its previous value is increased by 1 when there is a write request, and if the queue was not full.
- It is decreased by 1 when there is a read request, and if the queue was not empty.
- Otherwise it remains unchanged.

The equation at line 12 states that the value of nbmsg changes whenever there is a request in the queue.

```
13: | queue := (msg_in window fifo_size) cell (^access_clk)
```

The message queue is defined by the equation at line 13. The signal queue is an array of dimension fifo_size that contains the fifo_size latest values of msg_in, expressed by the window operator. The cell operator makes the signal queue available when access_clk is present, i.e., whenever there is a request.

```
14: | msg_out := prev_msg_out when (not OK_read) when (^msg_out)
         default queue[fifo_size - prev_nbmsg] when (^msg_out)
15: | prev_msg_out := msg_out $ 1 init default_msg
    |)
```

Finally, the statement at line 14 expresses that on a read request, i.e., at the clock ^msg_out, the outgoing message is either the one previously read if the FIFO message queue is empty (defined at line 15) or the oldest message in the queue.

```
   where
16:  integer prev_nbmsg; [fifo_size]message_type queue;
17:  message_type prev_msg_out;
   end;%basic_FIFO%
```

In the illustrative trace shown below, the type of the message is integer, the size limit is 2, and the default message value is −1.

t :	t_0	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}	t_{11}	…
msg_in :	·	4	6	·	·	·	·	5	7	8	·	·	…
access_clk :	t	t	t	t	t	·	t	t	t	t	t	t	…
msg_out :	−1	·	·	4	6	·	6	·	·	·	7	8	…
nbmsg :	0	1	2	1	0	·	0	1	2	2	1	0	…
OK_write :	t	t	t	f	t	·	t	t	t	f	f	t	…
OK_read :	f	f	t	t	t	·	f	f	t	t	t	t	…

The basic_FIFO model can be seen as a building block that serves to construct further types of message queues. This is illustrated in the next section.

12.2.2.2 Enhanced Model Design

The previous FIFO model is refined by taking into account a few constraints for safety reasons. One would like to avoid overwriting the FIFO queue when the it is full; similarly, one would like to avoid reading an empty FIFO queue. Such constraints are specified in a straightforward way as illustrated in the following process, called *safe FIFO*.

```
1: process safe_FIFO =
2:   { type message_type;
3:     integer fifo_size;
4:     message_type default_msg; }
5:   ( ? message_type msg_in; event get_mess;
6:     ! message_type msg_out;
7:       boolean OK_write, OK_read; )
```

Here, the interface is slightly different from that of `basic_FIFO`. The static parameters are the same. A new input signal `get_mess` has been added. It denotes a read request. The signal `nbmsg`, which was previously an output of `basic_FIFO`, is now a local signal.

```
8:  (| access_clk := msg_in ^+ get_mess
9:   | new_msg_in := msg_in when OK_write
10:  | msg_out ^= get_mess when OK_read
11:  | msg_out, nbmsg, OK_write, OK_read) :=
        basic_FIFO{message_type, fifo_size, default_msg}
                                    (new_msg_in, access_clk)
     |)
```

The statement at line `8` defines the access clock as the union of instants at which read or write requests occur. The equations at lines `9` and `10` ensure a safe access to the queue in `basic_FIFO`. The process call in the statement at line `11` has the local signal `new_msg_in` as an input. This signal is defined only when `basic_FIFO` is not full; it is stated in the equation at line `9`. Similarly, line `10` expresses that on a read request, a message is received only when `basic_FIFO` was not previously empty.

```
   where
12:   use basic_FIFO;
13:   integer nbmsg; message_type new_msg_in; event access_clk;
    end;%safe_FIFO%
```

In the above local declarations, the `use` keyword used at line `12` enables one to import the description of the specified process, here `basic_FIFO`, into the `safe_FIFO` process. It is particularly useful when some processes defined in a module (e.g., representing a library of services or components) should be imported in a given context of usage. In that case, the whole module could be imported.

Within the following trace, the same parameters as for `basic_FIFO` are considered.

t :	t_0	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}	t_{11}	...
msg_in :	·	4	6	·	·	·	5	7	8	·	·	·	...
get_mess :	t	·	·	t	t	t	·	·	·	·	t	t	...
msg_out :	·	·	·	4	6	·	·	·	·	·	5	7	...
OK_write :	t	t	t	f	t	t	t	t	f	·	f	t	...
OK_read :	f	f	t	t	t	f	f	t	t	·	t	t	...

Through the bottom-up design example shown, we can observe that modularity and reusability are key features of SIGNAL programming. They favor component-based designs. By constraining a given component, one can derive other components. The most difficult task is the identification of the suitable basic components. It depends quite a lot on the creativity or the experience of the designer.

12.3 Control-Related Aspects

This section deals with the modeling of two usual control-related aspects: finite state machines (FSMs; Sect. 12.3.1) and preemption (Sect. 12.3.2).

12.3.1 Finite State Machines

FSMs are very useful to describe controlled behaviors. They are characterized by a set of states $\{s_{i,i\in 0..m}\}$, an initial state s_0, a set of events $\{e_{i,i\in 0..n}\}$, and a transition function τ that maps events and current states to the next states. Actions may take place either during transitions or within states.

12.3.1.1 Informal Description of an FSM

In the FSM depicted in Fig. 12.4, there are five states that are represented differently: *s0, s1, s2, ERR_empty, ERR_full*. The transitions are labeled by two events: *in, out*. In fact, this FSM abstracts the behavior of a 2-FIFO queue:

- The events *in* and *out*, respectively, denote write and read requests.
- A state sk, represented by a circle, denotes the fact that the queue currently contains k items. In other words, $\forall k \in \{0, 1, 2\}$:

$$(nbmsg = k \Rightarrow sk = true) \wedge (nbmsg \neq k \Rightarrow sk = false),$$

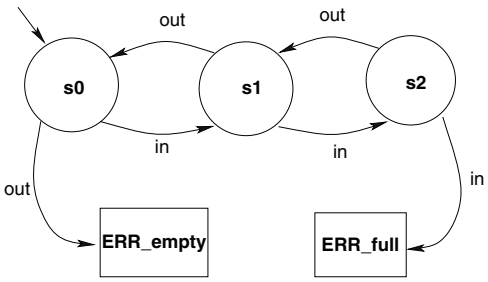


Fig. 12.4 2-FIFO queue abstraction

where *nbmsg* is the current number of messages in the message queue considered. The two special states *ERR_empty* and *ERR_full*, represented by rectangles, characterize "illegal" access to the queue: *ERR_empty* is reached on an attempt to read an empty queue, and *ERR_full* is reached when overwriting a full queue.

12.3.1.2 SIGNAL Model of the FSM

The SIGNAL specification of the above machine is very simple. Every state of the FSM is represented by a signal variable s_i, i.e., the definition of such a variable is an expression including the *delay* operator of SIGNAL. The clock of these states is constrained to be equal to the union of all clocks associated with occurring transition events. Boolean state variables are generally suitable for representing the states of an FSM. They hold the value *true* when the FSM is in the corresponding states. Otherwise, they hold the value *false*.

To illustrate the FSM encoding in SIGNAL, let us focus on `s0` and its associated transitions:

```
1: (| s0 := (true when prev_s1 when (^out))
2:          default (false when prev_s0 when(in ^+ out))
3:          default prev_s0
4:  | prev_s0 := s0 $ 1 init true
5:  | ...
6:  | s0 ^= s1 ^= s2 ^= ERR_empty ^= ERR_full ^= in ^+ out
7:  |)
```

Here, the states of the FSM are encoded by Boolean signals. The statement at line `1` specifies the fact that the FSM goes into `s0` if it was previously in `s1` while a read request is received. Line `2` specifies that the FSM leaves `s0` if it was previously in this state while a read or/and write request is received. The last statement of the first equation (line `3`) says that no transition takes place when no read/write request is received. Line `4` defines `prev_s0` as the previous state of `s0`.

All the other states are specified in a similar way. Note that all states are present whenever there is any request denoted by the occurrence of a transition event in the FSM (line `6`).

12.3.2 Preemption

To illustrate the modeling of preemption, let us consider again the ABRO example that was described in ESTEREL in Chap. 2. A possible SIGNAL encoding of this example is given in Fig. 12.5. Boolean state variables are used again to describe the expected behavior.

The ABRO process has three inputs A, B, and R and one output O. All these signals are of event type. The output O should occur when inputs A and B have both arrived provided that the input R does not occur first, otherwise R resets the behavior.

The first equation (line 4) defines the synchronization relation between the manipulated signals. There are four local Boolean signals A_received, B_received, from_R_before_O, and after_R_until_O that are also state variables, i.e., defined with a *delay* operator. These signals are set at the fastest clock in the program, which consists of the union of the clocks of input signals A, B, and R.

In the equation at line 5, the value of the signal A_received is initially set to *false*. It becomes *true* when the input A is received provided that R is absent. It holds the value *false* whenever R occurs. The value of the signal B_received is described in a similar way in the equation at line 7.

In the equation at line 9, the Boolean signal from_R_before_O is *true* when either O does not occur or R occurs. So, depending on its previous value, the decision to produce O is taken in the equation at line 12. In the equation at line 11, after_R_until_O denotes the previous value of from_R_before_O. Finally, the equation at line 12 says that O is produced when A and B have been received while neither O has been emitted nor R has been received since the last production of O.

```
1: process ABRO=
2: ( ? event A, B, R;
3:   ! event O;)
4: (| A_received ^= B_received ^= after_R_until_O ^= A ^+ B ^+ R
5:  | A_received := not R default A
6:                  default A_received $ init false
7:  | B_received := not R default B
8:                  default B_received $ init false
9:  | from_R_before_O := not O default R
10:                      default after_R_until_O
11: | after_R_until_O := from_R_before_O $ init true
12: | O := when A_received when B_received when after_R_until_O
13: |)
14: where
15:   boolean A_received, B_received, from_R_before_O,
16:           after_R_until_O;
17: end;
```

Fig. 12.5 The ABRO example described in SIGNAL

A possible execution trace of the ABRO process is as follows:

t :	t_0	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}	t_{11}	t_{12}	t_{13}	...
A :	t	·	t	·	·	·	t	·	·	t	·	·	t	t	...
B :	·	t	t	·	t	·	·	t	·	t	·	t	·	t	...
R :	·	·	·	t	·	t	·	·	t	·	t	·	·	t	...
O :	·	t	·	·	·	·	·	t	·	t	·	·	t	·	...

Boolean state variables (i.e., Boolean signals defined with the *delay* operator) and `event` signals (i.e., clocks) play an important role in the modeling of control-oriented behaviors.

12.4 Oversampling

12.4.1 Euclid's Algorithm for Greatest Common Divisor Computation

The *greatest common divisor* (GCD) of two integers x and y consists of the greatest integer that divides both x and y:

$$\gcd(x, y) = \max\{d \mid x \bmod d = 0 \text{ and } y \bmod d = 0\}.$$

For instance, $\gcd(108, 960) = 12$ and $\gcd(12, 18) = 6$.

When $y > 0$, we have $\gcd(0, y) = y$. This is justified by the fact that any strictly positive integer divides 0; and y is the greatest divisor of itself. Finally, the value $\gcd(0, 0)$ is undefined.

A well-known way to compute the GCD of two integers is *Euclid's algorithm*. Given a pair of integers x and y such that $0 \leq x < y$, the value of $\gcd(x, y)$ is obtained through the following recurrent form:

$$\begin{cases} \gcd(0, y) = y, \\ \gcd(x, y) = \gcd(y \bmod x, x), \end{cases}$$

For instance, $\gcd(12, 18) = \gcd(6, 12) = \gcd(0, 6) = 6$. Such an algorithm is quite simple to program in the usual languages either using iteration constructs such as *loops* or using recursive mechanisms.

A very simple way to model this algorithm can be formulated as follows: *if* $x = 0$*, the result is the value of* y*; otherwise the greatest of both integers* x *and* y *is replaced by the difference of their values, and the same process is applied again to the resulting integers.*

The next section describes a SIGNAL process that encodes such a model.

12.4.2 SIGNAL Modeling of the Algorithm

In the definition of the GCD_Euclid model, the only issue is how to describe loop iterations when all arguments x and y of the function gcd are different from 0 in the above algorithm. For that purpose, the oversampling mechanism of SIGNAL (see Sect. 10.3, page 155) can be used.

In the GCD_Euclid process shown in Fig. 12.6, the clocks of interface signals, i.e., x, y, and res, are smaller than that of the local signals xtemp and ytemp, which are synchronous (line 13). This illustrates a situation where a process has an internal master. The iteration steps take place at this internal master clock cycle until the result of gcd(x, y) is computed, i.e., when the condition ytemp = 0 holds. Afterwards, a new pair of input signal values is allowed; this is expressed by the first equation at line 5 in the process. The local signals xtemp and ytemp are used, respectively, as local copies of x and y during iterations.

> The oversampling mechanism is a key ingredient for the description of algorithms with iterations or recursion. In such situations, the faster clock at which the iteration/recursion steps take place is associated with local signals (and not with interface signals).

```
1: process GCD_Euclid =
2:    ( ? integer x, y;
3:     ! integer res;
4:    )
5:    (| x ^= y ^= when (pre_ytemp = 0)
6:     | xtemp := x default (pre_ytemp when c1) default
7:                 pre_xtemp
8:     | pre_xtemp := xtemp $ 1
9:     | ytemp := y default (pre_xtemp when c1) default
10:                ((pre_ytemp - pre_xtemp) when c2) default
11:                ytemp_1
12:    | pre_ytemp := ytemp $ 1
13:    | xtemp ^= ytemp
14:    | c1 := (pre_xtemp > pre_ytemp) when (pre_ytemp /= 0)
15:    | c2 := (pre_xtemp <= pre_ytemp) when (pre_ytemp /= 0)
16:    | res := xtemp when (ytemp = 0)
17:    |)
18:   where
19:     integer xtemp, ytemp, pre_xtemp, pre_ytemp;
20:     boolean c1, c2;
21: end%process GCD_Euclid%;
```

Fig. 12.6 Euclid's greatest common divisor algorithm described in SIGNAL

12.5 Endo-isochrony

This section presents the design of a system example in terms of endoisochronous processes as discussed in Chap. 11.

12.5.1 A Flight Warning System

A *flight warning system* (FWS) is used in the Airbus A340 aircraft. It was proposed by Aerospatiale (France) as a case study in [3]. It is in charge of deciding when and how to emit warning signals whenever there is an anomaly during the operational mode of an airplane. The system consists of two cyclic concurrent processes:

- Given an alarm a_i, the *alarm manager* process confirms a_i after a given period of time or removes a_i from the set of confirmed alarms depending on the fact that a_i is detected as being "present" or "absent."
- The *alarm notifier* process emits warning signals associated with confirmed alarms.

Even though the two processes share information, they execute independently. So, their activation clocks are a priori not required to be correlated. Such a system is particularly well suited to be described and analyzed with the polychronous model.

12.5.2 SIGNAL Modeling of the FWS

In the FWS model presented below, the way alarms are made "present" or "absent" is considered as beyond the scope of this work. So, it will not be addressed here.

12.5.2.1 Alarm Manager and Notifier Processes

Figures 12.7 and 12.8 give an excerpt of the process models composing the FWS. Both processes are defined in a very similar way. For that reason, only one of them is detailed: `Alarm_Manager` (see Fig. 12.7).

The interface of the `Alarm_Manager` process consists of two static parameters `k` and `delay` (line 2). It also includes an input signal `alarm_in` (line 3) and an output signal `alarm_out` (line 4), both declared as arrays of alarms. Their dimension is some fixed integer constant `n`. The type of an alarm, called `alarm_type`, is a structured type with two fields: `pres` (resp. `conf`) of Boolean type that is *true* when an alarm is "present" (respectively "confirmed") and *false* when an alarm is "absent" (respectively "removed").

The body of the `Alarm_Manager` process is composed of five equations. The local signals `cnt`, `zcnt`, and `start_confirm` are used to specify when the

```
process Alarm_Manager =
   { integer k,delay }
   ( ? [n] alarm_type alarm_in;
     ! [n] alarm_type alarm_out; )
   (| cnt:= ((k-1) when (zcnt = 0))
                default (zcnt - 1)
    | zcnt:= cnt $ init 0
    | start_confirm:= when(zcnt = delay)
    | alarm_in ^= start_confirm
    | alarm_out:= Alarm_Confirm(alarm_in)
    |)
  where
    integer cnt, zcnt;
    event start_confirm;
    process Alarm_Confirm =
       ( ? [n] alarm_type in;
         ! [n] alarm_type out; )
       (| array i to n-1 of
            out[i].conf := in[i].pres
          end
        |)%process Alarm_Confirm%;
end%process Alarm_Manager%;
```

Fig. 12.7 Excerpt of the SIGNAL code for alarm management

```
process Alarm_Notifier =
   { integer k,delay }
   ( ? [n] alarm_type alarm;
     ! event s_0,...,s_n-1; )
   (| cnt:= ((k-1) when (zcnt = 0))
            default (zcnt - 1)
    | zcnt:= cnt $ init 0
    | start_notif:= when (zcnt = delay)
    | alarm ^= start_notif
    | (s_0,...,s_n-1):= Alarm_Notif(alarm)
    |)
  where
    integer cnt, zcnt;
    event start_notif;
    process Alarm_Notif =
       ( ? [n] alarm_type alarm;
         ! event s_0,...,s_n-1; )
       (| s0:= when alarm[0].conf
        | ...
        | s_n-1:= when alarm[n-1].conf
        |)%process Alarm_Notif%;
 end %process Alarm_Notifier%;
```

Fig. 12.8 Excerpt of the SIGNAL code for alarm notification

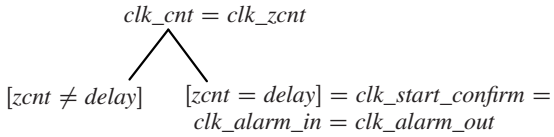
process confirms alarms in equations at lines 5–8. The signal cnt plays the role of a *counter* of logical instants. It is set to k-1 whenever its previous value zcnt becomes zero; otherwise its previous value is decreased by 1. The static parameter k can be seen as an initialization period value of cnt. The signal start_confirm denotes the logical instants at which a confirmation starts, described by the equations at lines 8 and 9. It occurs when the counter value reaches some amount of time denoted by delay within each cycle.

Alarm_Confirm is subprocess defining the notification treatment from lines 15–21. It uses the *array of processes* derived construct (see Sect. 5.4), which enables one to describe instantaneous iterations.

According to the clock properties of SIGNAL constructs, we can deduce the following system of clock constraints associated with the process Alarm_Manager (for a signal s, we denote by *clk_s* its associated clock):

$$\begin{aligned}
clk_cnt &= [zcnt = 0] \cup clk_zcnt && \text{(lines 5 and 6)}\\
clk_zcnt &= clk_cnt && \text{(line 7)}\\
clk_start_confirm &= [zcnt = delay] && \text{(line 8)}\\
clk_alarm_in &= clk_start_confirm && \text{(line 9)}\\
clk_alarm_out &= clk_alarm_in && \text{(lines 10 and 19)}
\end{aligned} \tag{12.1}$$

After reductions of the above system (12.1), the following clock hierarchy is obtained:



The master clock is the one of cnt, which is the same for zcnt. The other clocks of the process are defined following a partitioning of the values of zcnt with respect to the value of delay. As a result, the process Alarm_Manager is endochronous, and hence deterministic.

In a similar way, the process Alarm_Notifier is also proved to be endochronous. The SIGNAL compiler allows one to automatically check the endochrony property of a process based on this technique. Moreover, it generates a code associated with such a process for simulation.

12.5.2.2 The Global System

The SIGNAL process depicted by Fig. 12.9 represents the FWS model. From a structural viewpoint, the previous process models, i.e., Alarm_Manager and Alarm_Notifier, are subprocesses of the FW_System model.

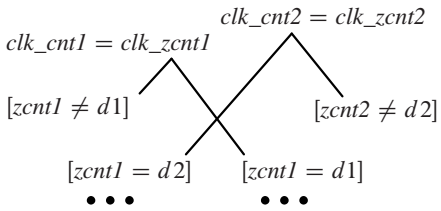
The FWS_System process takes as an input a collection of alarms represented by the alarm array (line 3), and produces as outputs the signals denoted by s_0, ..., s_n-1. These outputs are generated by the process Alarm_Notifier (see Fig. 12.8).

```
1: process FW_System =
2:    { integer k1,k2,d1,d2 }
3:    ( ? [n] alarm_type alarm;
4:      ! event s_0,...,s_n-1; )
5:    (| tmp:= Alarm_Manager{k1,d1}(alarm)
6:     | (s_0,...,s_n-1):=
7:               Alarm_Notifier{k2,d2}(tmp)
8:     |)
9:   where
10:    [n] alarm_type tmp;
11:    process Alarm_Manager = ...;
12:    process Alarm_Notifier = ...;
13: end%process FW_System%;
```

Fig. 12.9 Excerpt of the SIGNAL code of the flight warning system

The equations from line `5` to line `7` define the interaction between the concurrent processes composing the FWS: the input of process `Alarm_Notifier` is the output of process `Alarm_Manager`. This transfer of data is realized via the local signal `tmp`. This means that the two processes must agree on the set of specific instants at which this transfer is possible.

Let us analyze the clock properties induced by the definition of process `FW_System`. We obtain the following two-rooted clock hierarchy:



The clock trees associated with `Alarm_Manager` and `Alarm_Notifier` are clearly distinguished. There is no master clock because the clocks of the subprocesses (`clk_cnt1` for `Alarm_Manager` and `clk_cnt2` for `Alarm_Notifier`) are unique to the particular subprocess. However, they share common instants at which the alarm transfer is done between the manager and the notifier processes. Such instants are defined only when both `zcnt` in `Alarm_Manager` and `zcnt` in `Alarm_Notifier` are, respectively, equal[2] to `d1` and `d2`. Such a requirement appears under the form of a clock constraint generated by the SIGNAL compiler.

In such a situation, it means that:

- Either the environment of the two concurrent processes is able to guarantee that they can synchronize at the required specific logical instants, and the model will execute as expected.

[2] Note that `zcnt` in `Alarm_Manager` and `zcnt` in `Alarm_Notifier` are different signal instances; they have different clocks and evolve according to two different logical time scales.

- Or it is not possible, and the output of the whole system model is undefined whenever the clocks of the two processes fail to agree.

The `assert` intrinsic process offers a way to define assumptions in programs. It could be used here to specify the requirement for the concurrent execution of `Alarm_Manager` and `Alarm_Notifier` processes.

The corresponding code which is automatically produced by using the "`-force`" option of the compiler from such a model contains exception messages that are thrown whenever the clock constraints are not satisfied during code execution.

The above `FW_System` process satisfies the endo-isochrony property: first, `Alarm_Manager` and `Alarm_Notifier` models are both endochronous; second, their communicating part which consists of the single signal `tmp` (see Fig. 12.9) is trivially endochronous. Indeed, a single signal yields a clock hierarchy with only one clock node, which consequently forms a clock tree. The synchronous communication between the processes `Alarm_Manager` and `Alarm_Notifier` via `tmp` can be equivalently replaced by an asynchronous communication while still preserving the semantics of the `FW_System` model.

A constructive way to obtain endoisochronous models in SIGNAL consists in modularly composing endochronous parts of this model.

12.6 Exercises

12.1. Define a complete SIGNAL specification of the FSM example presented in Sect. 12.3.1 as an abstraction of a 2-FIFO queue.

In the same SIGNAL specification, define two Boolean signals `OK_write` and `OK_read` that, respectively, denote the fact that write and read requests are authorized or not.

12.2. Let us consider the FWS system model described in Sect. 12.5.2. Is there any way to define the `FWS_System` process such that the communication between the two concurrent processes becomes clock-constraint-free?

Expected Learning Outcomes:

In this chapter, some advanced aspects of SIGNAL programming have been illustrated via examples. These aspects are:

- **Top-down** design approach via specification refinements
- **Bottom-up** design approach via incremental compositions
- **Control-related features** via Boolean and event signal manipulation

- Clock **oversampling** via a model in which the input parameters are less frequent than its computed local signals in terms of logical instants
- **Endo-isochrony** via a system composed of two processes that communicate asynchronously

References

1. Airlines Electronic Engineering Committee (1997) ARINC specification 653: Avionics application software standard interface. Aeronautical radio, Inc, Annapolis, Maryland
2. Gamatié A, Gautier T (2002) Synchronous modeling of modular avionics architectures using the SIGNAL language. Research Report 4678, INRIA Available at: http://www.inria.fr/rrrt/rr-4678.html
3. Lopez N, Simonot M, Donzeau-Gouge V (2002) A methodological process for the design of a large system: two industrial case-studies. 7th International ERCIM Workshop in Formal Methods for Industrial Critical Systems (FMICS'02), University of Malaga, Spain, LNCS 66(2)

Chapter 13
A Synchronization Example Design with POLYCHRONY

Abstract This chapter aims to show how a solution to a complex synchronization problem can be designed in practice with the POLYCHRONY environment. The well-known *dining philosophers* problem is considered as a case study. It illustrates a concurrency model in which issues regarding resource allocation should be solved, such as unfair resource utilization and deadlock. Section 13.1 informally introduces the problem. Then, a solution is proposed in SIGNAL in Sect. 13.2. The POLYCHRONY analysis and code generation tools are used to check the correctness of the proposed solution and to simulate it.

13.1 The Dining Philosophers Problem

First, the problem is informally introduced in Sect. 13.1.1. Then, a solution is presented in Sect. 13.1.2. Finally, the solution is encoded in SIGNAL within POLYCHRONY in Sect. 13.2.

13.1.1 Informal Presentation

Dijkstra [1] first proposed the synchronization problem where five computers compete to gain access to five shared tape drive peripherals. Then, Hoare provided the model of dining philosophers[1] as a model for a multiprocess synchronization. The model consists of a group of philosophers, typically five of them arranged around a table. On this table there are five forks and five plates with food on them. The forks are arranged in-between the philosophers as shown in Fig. 13.1.

Each philosopher needs to have two forks for him to eat with. To get control of two forks, each philosopher must compete with neighboring philosophers. The philosophers can be in any of the three states: *thinking, hungry*, and *eating*. Whenever a philosopher is hungry, he will keep trying to get two forks and move into

[1] http://en.wikipedia.org/wiki/Dining_philosophers

A. Gamatié, *Designing Embedded Systems with the SIGNAL Programming Language: Synchronous, Reactive Specification*, DOI 10.1007/978-1-4419-0941-1_13,
© Springer Science+Business Media, LLC 2010

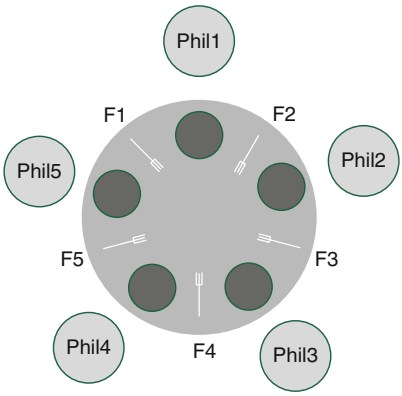


Fig. 13.1 The dining philosophers

the eating state. In the eating state, he will start consuming the food and return to the thinking state. As long as he is not hungry again, the philosopher will remain in the thinking state and will not try to get the forks.

If each philosopher grabs one fork and refuses to give it up, there can be a deadlock. Such a situation should be avoided. Another scenario which is not permitted is when one of the philosophers is preferred over the others and leaves one or more philosophers starving. These cases are analogous to the problems in resource allocation. This makes the dining philosophers problem a good example for dealing with concurrency.

13.1.2 A Solution

Multiple solutions have been proposed in the literature for the dining philosophers problem. Here, the solution implemented aims to be fair to each philosopher. For avoidance of deadlock, a priority is assigned to each philosopher, which determines the order in which forks are assigned. Once a philosopher has eaten, his priority will be the least and the rest of the philosophers will have their priorities raised. This will ensure that no philosopher ends up starving. The state diagrams considered for the philosopher states and for fork states are given in Fig. 13.2. For fairness in distribution of resources, a rotational priority scheme has been defined.

13.2 Design of the Solution Within POLYCHRONY

A preliminary observation about the problem is that the encoding of its solution will be a complex SIGNAL process. Thus, one must decompose it so as to be able to locally address various aspects of the programming, e.g., synchronization analysis and

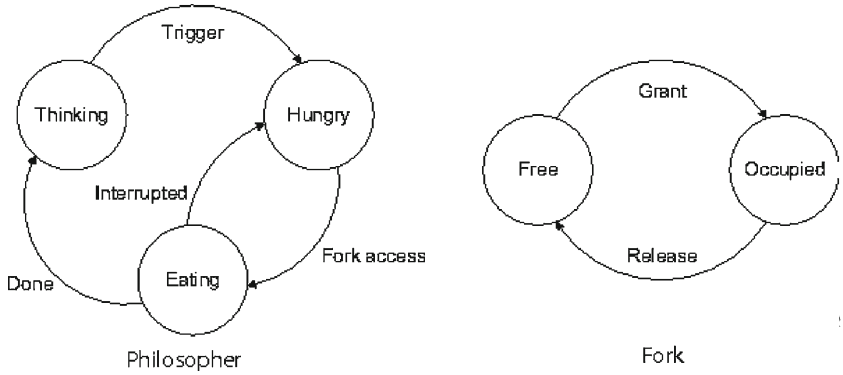


Fig. 13.2 States of philosophers (*left*) and forks (*right*)

functional simulation of an actor model when considered aside. When the expected local properties of actor models are satisfied, they could be composed together.

In the modeled solution [2], the actors are philosophers and forks. They will be modeled as two independent SIGNAL processes. A third process, referred to as the *main process* in the next section, will be defined to coordinate the information exchanged between both processes.

13.2.1 Modeling of Philosophers

13.2.1.1 SIGNAL Description

Each philosopher accepts the grant permission offers from the neighboring forks, a signal which triggers hunger and an interrupt event. All five philosophers execute concurrently and output their requests to neighboring forks and their states to the main process. This leads to the following interface for a philosopher:

```
1: process Philosopher =
2:  {integer N;}
3:  (? event tick, haveright, haveleft, htrigger, interupt;
4:   ! event askright, askleft, release;
5:     integer pstate; )
```

The static parameter `N` is used here to identify philosophers.

The input signal `tick` of `event` type denotes an explicit activation clock for the process representing a philosopher. It serves to count the number of cycles during which a philosopher is eating. It will be provided by the main process, which specifies the interaction between philosophers and forks. The other inputs `haveright`, `haveleft`, `htrigger`, and `interrupt` denote, respectively, grant permission

offers received by a philosopher from its right and left forks, hunger trigger, and interrupt event. They are also specified as `event` signals.

The outputs signals `askright` and `askleft` represent requests from a philosopher for its right and left forks, respectively. The signal `release` is emitted when an eating philosopher is full so that its hungry neighbors can take the forks. The philosopher therefore moves to the *thinking* state. A counter is implemented in the philosopher process to indicate the number of clock cycles required to finish eating.

The current state of a philosopher is given by the output integer signal `pstate`, according to the following convention:

- `0` means *thinking*
- `1` means *hungry*
- `2` means *eating*

The next equations define the way a philosopher asks for forks in order to eat. Let us consider the case where a philosopher asks for the fork on its right. This is specified by the statements from line `6` to line `10`:

```
6: (| askright := when(pstate=1) when(not Cond_haveright)
7:  | Cond_haveright := (haveright when(pre_pstate=1))
8:     default (false when (pre_pstate=2) when(interupt default
9:     (pre_counter=3))) default pre_Cond_haveright
10: | pre_Cond_haveright := Cond_haveright$ init false
```

A local Boolean state variable, named `Cond_haveright`, reflects whether or not the philosopher actually holds the fork. Its defining equation specifies that it holds the value *true* when the philosopher was hungry and received a grant permission offer concerning the fork on the right; and it holds the value *false* when the philosopher was interrupted while eating, or when it had finished eating.

The signal `Cond_haveright` is then used in the equation at line `6` to define the output signal `askright`, which is emitted whenever a philosopher entered the state *hungry* without the fork on its right-hand side.

In a very similar way, a request for a fork on the left-hand side of a philosopher is defined by the statements from line `11` to line `15`:

```
11: | askleft := when(pstate=1) when (not Cond_haveleft)
12: | Cond_haveleft := (haveleft when(pre_pstate=1)) default
13:    (false when (pre_pstate=2) when(interupt default
14:    (pre_counter=3))) default pre_Cond_haveleft
15: | pre_Cond_haveleft := Cond_haveleft$ init false
```

From line `16` to line `21`, the two equations specified describe how a philosopher switches from a state to another according to the automaton depicted in Fig. 13.2. Roughly speaking, the equation at line `16` says that a philosopher eats as soon as it gets the required forks while it is hungry; it returns to the *thinking* state as soon as it is full (note that the number of cycles required to finish eating is taken into account); and it becomes hungry when a hunger trigger event is received while it is thinking or when an interrupt event is received while it is eating.

```
16: | pstate := 2 when(pre_pstate=1) when(Cond_haveleft and
17:    Cond_haveright) default 0 when (pre_pstate=2) when
18:    (pre_counter=3) default 1 when ((when(pre_pstate=0)
19:    when htrigger) default (when(pre_pstate=2) when
20:    interupt)) default pre_pstate
21: | pre_pstate := pstate$ init 0
```

The next equation specifies that a philosopher releases its forks after it has finished eating, i.e., when returning to the *thinking* state from the *eating* state.

```
22: | release := when (pstate=0) when (pre_pstate=2)
```

The next two equations define the signal `counter` denoting the cycle counter when a philosopher is eating. Here, the number of authorized cycles is arbitrarily fixed to 3.

```
23: | counter:= (pre_counter+1) when (pstate=2) default 0
24: | pre_counter:= counter$ init 0
```

All local state variables manipulated in the process `Philosopher` are assumed to have the same clock, which is equal to the clock union of input signals. This is expressed by the following synchronization constraint:

```
25: | pstate ^= counter ^= Cond_haveright ^= Cond_haveleft ^=
26:    haveright ^+ haveleft ^+ htrigger ^+ interupt ^+ tick
27: |)
28: where
29:  boolean Cond_haveleft, Cond_haveright, pre_Cond_haveright,
30:          pre_Cond_haveleft;
31:  integer counter, pre_counter, pre_pstate;
32: end;
```

Let us call `Philosopher.SIG` the file containing the above SIGNAL description.

13.2.1.2 Analysis with the Compiler

To check the syntactic correctness of the `Philosopher` process, we use the command `signal` of the compiler with option `lis` as follows:

```
signal -lis Philosopher.SIG
```

The diagnostic file generated, named `Philosopher_LIS.SIG`, is strictly identical to `Philosopher.SIG`, without any additional annotation that notifies one of the usual compilation errors such as type or declaration mistakes. It means that the process is syntactically correct.

Now, we can address the clock analysis and ask for the construction of a hierarchized conditional dependency graph associated with process `Philosopher`. For this purpose, we use the option `tra` with the `signal` command:

```
signal -tra Philosopher.SIG
```

A sketch of the process generated in the file `Philosopher_TRA.SIG` is as follows:

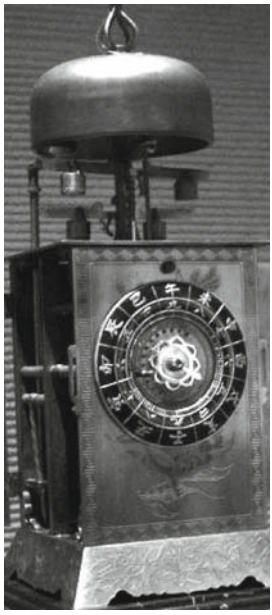
```
process Philosopher_TRA =
  ( ? event tick, haveright, haveleft, htrigger, interupt;
    ! event askright, askleft, release;
      integer pstate;
   )
pragmas
   Main
end pragmas
  (| (| haveright ^= haveright |)
   | (| CLK := CLK_14 ^* haveright |)
   | (| interupt ^= interupt |)
   | (| CLK_22 := CLK_139 ^- interupt |)
   | (| CLK_XZX := interupt ^+ CLK_139
   | ACT_CLK_XZX{}
   |)
   | (| CLK_29 := CLK_24 ^* CLK_27 |)
   | (| CLK_31 := CLK_29 ^- CLK |)
   | (| CLK_32 := CLK ^+ CLK_29 |)
   | (| CLK_35 := XZX_76 ^- CLK_32 |)
   | (| XZX_76 := CLK_104 ^+ tick
      | XZX_76 ^= pstate ^= pre_counter
      | ACT_XZX_76{}
      |)
 | (| haveleft ^= haveleft |)
 | (| CLK_45 := CLK_14 ^* haveleft |)
 | (| CLK_49 := CLK_27 ^* CLK_46 |)
 | (| CLK_51 := CLK_49 ^- CLK_45 |)
 | (| CLK_52 := CLK_45 ^+ CLK_49 |)
 | (| CLK_55 := XZX_76 ^- CLK_52 |)
 | ...
 |)
where
   constant integer N = 5;
   event CLK_139, CLK_126, CLK_104, CLK_103,
         CLK_102, CLK_84, CLK_82, CLK_81, XZX_57,
         CLK_75, CLK_73, CLK_71, CLK_55, CLK_52,
         ...
   integer pre_counter;
   process ACT_CLK_XZX =
    ( )
    (| CLK_XZX ^= XZX ^= XZX_46
     | (| CLK_24 := when XZX |)
     | (| CLK_46 := when XZX_46 |)
     | (| XZX := (interupt when interupt) default
                ((pre_counter=3) when CLK_22)
       | XZX_46 := (interupt when interupt) default
                   ((pre_counter=3) when CLK_22)
       |)
     |)
    where
      boolean XZX, XZX_46;
    end
```

```
53:   %ACT_CLK_XZX%;
54:   process ACT_XZX_76 =
55:    ( )
56:    (| XZX_76 ^= Cond_haveleft ^= Cond_haveright ^=
57:        pre_Cond_haveright ^= pre_Cond_haveleft ^=
58:        counter ^= pre_pstate
59:     | (| CLK_7 := when (not Cond_haveright) |)
60:     | (| CLK_10 := when (pstate=1) |)
61:     | ...
62:     |)
63:    where ... end%ACT_XZX_76%;
64:  end%Philosopher_TRA%;
```

First of all, there are no unsatisfied clock constraints in the specification of the `Philosopher` program. If there were unsatisfied clock constraints, a warning message would be generated in the process `Philosopher_TRA`.

On the other hand, according to the clock hierarchy reflected by the program generated, there is no unique root clock from which all other clocks of the process can be extracted by downsampling (see Sect. 9.2.2). Instead, there are several clocks that are defined at the same level. For instance, this is the case for `CLK`, `CLK_22`, and `CLK_XZX`, which are defined at lines `10`, `12`, and `13`, respectively. The `Philosopher` program is therefore *exochronous*.

However, one may ask for an automatic endochronization of the `Philosopher` program by the compiler. Such an endochronization may typically be similar to the first example shown in Chap. 9, on page 136, where additional Boolean input signals are considered in the initial program.



An eighteenth century Japanese clock, from Tokyo National Science Museum, 2004 (public domain picture)

The automatic endochronization of an exochronous program is performed only when the user asks for code generation.

13.2.1.3 Code Generation

The automatic code generation by the compiler allows one to obtain a simulation program in different languages. In SIGNAL, these languages are C, C++, and Java. For illustration, we consider the C language here. The code generation is performed by invoking the following command:

```
signal -c Philosopher.SIG -par=Philosopher.PAR
```

where the file `Philosopher.PAR` contains the value of the parameter `N` declared in process `Philosopher`. For instance, let us consider $N = 5$.

The result of this command consists of several files that are interrelated. Each file contains a specific aspect of the global C program to be simulated. For instance, the file `Philosopher_main.c` describes the `main()` function of the program as follows:

```
/*
  Generated by Polychrony version  V4.15.10
*/
#include "Philosopher_types.h"
#include "Philosopher_externals.h"
#include "Philosopher_body.h"
EXTERN void Philosopher_OpenIO();
EXTERN void Philosopher_CloseIO();

EXTERN int main()
{
    logical code;
    Philosopher_OpenIO();
    code = Philosopher_initialize();
    while(code)code = Philosopher_iterate();
    Philosopher_CloseIO();
}
```

In the `main()` part of this code, the files that contain input data are opened first. Then, all state variables, i.e., memorized variables, are initialized and the main loop executes the `Philosopher_iterate()` function (see below). Finally, the files that contain output data are closed.

All the functions called in `main()` are defined in the other files generated.

Let us focus on the file `Philosopher_body.c`; it is central because it describes the basic behavior performed at every loop step or reaction by the `Philosopher` process.

```
/*
  Generated by Polychrony version  V4.15.10
*/
#include "Philosopher_types.h"
#include "Philosopher_externals.h"
#include "Philosopher_body.h"

/*  ==> parameters and indexes */
/*  ==> input signals     */
static logical C_tick, C_haveright, C_haveleft,
               C_htrigger, C_interupt;
```

```
/*  ==> output signals    */
static logical askright, askleft, release;
static int pstate;
/*  ==> local signals     */
static logical Cond_haveleft, Cond_haveright;
static int counter;
static logical XZX, XZX_46, C_120, C_Cond_haveleft, C_140,
               C_145, C_148, C_156, C_159, C_174, C_179,
               C_182, C_198, C_release, C_204, C_207, C_210,
               C_askright, C_askleft, C_229, C_232, C_238,
               C_244, C_253;

EXTERN logical Philosopher_initialize()
{
    Cond_haveright = FALSE;
    Cond_haveleft = FALSE;
    counter = 0;
    pstate = 0;
    Philosopher_STEP_initialize();
    return TRUE;
}

static void Philosopher_STEP_initialize()
{
    C_145 = FALSE;
    C_156 = FALSE;
    C_179 = FALSE;
    C_207 = FALSE;
}

EXTERN logical Philosopher_iterate()
{
  if (!r_Philosopher_C_tick(&C_tick)) return FALSE;
  if (!r_Philosopher_C_haveright(&C_haveright)) return FALSE
  if (!r_Philosopher_C_haveleft(&C_haveleft)) return FALSE;
  if (!r_Philosopher_C_htrigger(&C_htrigger)) return FALSE;
  if (!r_Philosopher_C_interupt(&C_interupt)) return FALSE;
  C_120 = ((C_haveright || C_haveleft) || C_htrigger)
                    || C_tick;
  C_Cond_haveleft = C_120 || C_interupt;
  if (C_Cond_haveleft)
      {
      if (C_interupt) XZX = TRUE;
        else XZX = counter == 3;
      if (C_interupt) XZX_46 = TRUE;
        else XZX_46 = counter == 3;
      C_145 = pstate == 1;
      C_156 = pstate == 2;
      C_174 = counter == 3;
      C_179 = pstate == 0;
      C_198 = C_156 && C_174;
      }
  C_148 = (C_Cond_haveleft ? C_145 : FALSE);
  C_159 = (C_Cond_haveleft ? C_156 : FALSE);
```

```
  C_182 = (C_Cond_haveleft ? C_179 : FALSE);
  C_229 = C_haveleft && C_148;
  C_232 = C_haveright && C_148;
  C_238 = (C_Cond_haveleft ? (XZX && C_156) : FALSE);
  C_244 = (C_Cond_haveleft ? (C_156 && XZX_46) : FALSE);
  C_253 = (C_interupt && C_159) || (C_htrigger && C_182);
  if (C_Cond_haveleft)
      {
      if (C_229) Cond_haveleft = TRUE;
        else if (C_156 && XZX_46) Cond_haveleft = FALSE;
           else ;
      if (C_232) Cond_haveright = TRUE;
        else if (XZX && C_156) Cond_haveright = FALSE;
           else ;
      C_204 = C_145 && (Cond_haveleft && Cond_haveright);
      C_207 = C_204 || C_198;
      }
  C_210 = (C_Cond_haveleft ? C_207 : FALSE);
  if (C_Cond_haveleft)
      {
      if (C_204) pstate = 2; else if (C_198) pstate = 0;
        else if (!C_210 && C_253) pstate = 1; else ;
      w_Philosopher_pstate(pstate);
      C_140 = pstate == 1;
      C_release = C_156 && (pstate == 0);
      C_askright = !Cond_haveright && C_140;
      C_askleft = C_140 && !Cond_haveleft;
      if (pstate == 2) counter = counter + 1;
        else counter = 0;
      if (C_release)
          {
          w_Philosopher_release(TRUE);
          }
      if (C_askright)
          {
          w_Philosopher_askright(TRUE);
          }
      if (C_askleft)
          {
          w_Philosopher_askleft(TRUE);
          }
      }
  Philosopher_STEP_finalize();
  return TRUE;
}

EXTERN logical Philosopher_STEP_finalize()
{
  Philosopher_STEP_initialize();
  return TRUE;
}
```

In the above code, the input, output, and local variables are first declared. Then, initialization functions are defined: `Philosopher_initialize()`, which is called once, initializes all state variables before entering the main loop that performs reactions; `Philosopher_STEP_initialize()` is called at the end of each reaction, i.e., at every global loop step, to reinitialize the manipulated state variables.

The function `Philosopher_iterate()` implements a reaction: the actions that are performed at each logical instant, corresponding to a global loop step. Thus, it is the heart of the simulated behavior. One can observe that this function starts by reading input data. Each input is encoded by a logical variable. For instance, `C_tick` encodes the event `tick` and is read with the function `r_Philosopher_C_tick`.

Afterwards, a logical local variable `C_Cond_haveleft` encodes the fact that at least one of the inputs is read. It represents the greatest clock that contains all instants at which there is a reaction, meaning that some events occur in the `Philosopher` process. This is the reason why the statements following the definition of the variable `C_Cond_haveleft` in the `Philosopher_iterate()` function are almost executed under the condition that this logical variable is *true*.

The body of function `Philosopher_iterate()` ends up with the writing of computed output values in their corresponding variables. For instance, when the `release` signal is present, expressed via the test on its clock encoding `C_release`, then it is set to `TRUE` using the function `w_Philosopher_release`.

Finally, the state variables of the program are updated for the next reaction in the main execution loop.

Remark 13.1. If there were possible dependency cycles in the `Philosopher` process, the code generation process would not produce any C code. Instead, it would generate a diagnostic file, named `Philosopher_CYC.SIG`, in which all dependency cycles will be explicitly described.

13.2.1.4 Simulation

To simulate the resulting C code given in the previous section, one can first ask for the generation of a specific `Makefile` file to compile the code and produce an executable format. This is done by executing the following command:

```
genMake C Philosopher
```

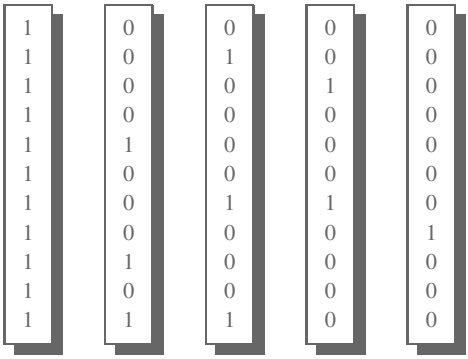
The file obtained is named `Makefile_Philosopher`. Now, the executable code can be produced by executing the next command:

```
make -f Makefile_Philosopher
```

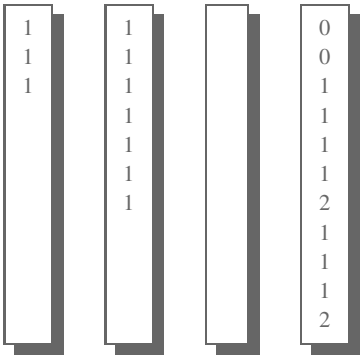
The executable program is generated in a file named `Philosopher`.

To perform a simulation, one has to define the input data, which are read at the beginning of the `Philosopher_iterate` function. They are described in separate files, one for each input signal, with the following convention: given an input signal named `s`, the associated file is named `Rs.dat` (`R` is put

for *read*). Logical values are represented by `0` and `1` for *false* and *true*, respectively. The following boxes illustrates from left to right typical contents of the files `RC_tick.dat` `RC_haveright.dat`, `RC_haveleft.dat`, `RC_htrigger.dat`, and `RC_interrupt.dat`:



After the execution of the program, its outputs are automatically generated in separate files similarly to input signals. If `s` denotes an output, its corresponding file is named by convention `Ws.dat` (`W` is put for *write*). The following boxes illustrates from left to right the files produced – `Waskright.dat`, `Waskleft.dat`, `Wrelease.dat`, and `Wpstate.dat`:



The above simulation results can be understood as the following trace:

t :	t_0	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}
tick :	t	t	t	t	t	t	t	t	t	t	t
haveright :	·	·	·	·	t	·	·	·	t	·	t
haveleft :	·	t	·	·	·	·	t	·	·	·	t
htrigger :	·	·	t	·	·	·	t	·	·	·	·
interrupt :	·	·	·	·	·	·	·	t	·	·	·
askright :	·	·	t	t	·	·	·	t	·	·	·
askleft :	·	·	t	t	t	t	·	t	t	t	·
release :	·	·	·	·	·	·	·	·	·	·	·
pstate :	0	0	1	1	1	1	2	1	1	1	2

Remark 13.2. In contrast to what is illustrated in the above trace, the files obtained via the simulation for the output signals `askright`, `askleft`, and `release` do not indicate the clock of these signals. They only show the occurrence values of these event signals. It is therefore hard to see exactly at which logical instants these events occur.

A simple way to straightforwardly obtain this information consists in adding three additional Boolean output signals to the process `Philosopher`. Each Boolean signal is used to encode the clock of an output event, and is made synchronous with `pstate`.

13.2.2 Modeling of Forks

13.2.2.1 SIGNAL Specification

A fork accepts the requests from the nearby philosophers along with their relative priorities. A decision on which philosopher should be granted access is taken on the basis of these priorities.

The interface of the SIGNAL model of a fork is as follows:

```
0: process Fork =
1: { integer M; }
2: (? event release, reqfright, reqfleft;
3:    integer rankfright, rankfleft;
4:  ! event gright, gleft;
5:    integer forkstatus; )
```

This interface is quite similar to that of Philosopher. The static parameter `M` identifies a fork. Fork requests from the right-hand side and left-hand side philosophers are, respectively, represented by the inputs `reqfright` and `reqfleft`. Since for any request, the asking philosophers must indicate the associated priority level, the inputs `rankfright` and `rankfleft` accordingly denote these priorities.

The outputs events `gright` and `gleft` are emitted if a grant permission offer is sent to either the right-hand-side philosopher or to the left-hand-side philosopher: `gright` denotes that the fork is held by the philosopher on the right and `gleft` denotes that the fork is held by the philosopher on the left.

The current status of a fork is reflected by the output `forkstatus` according to the following convention:

- `0` means *free*
- `1` means *occupied*

It is defined by the equations defined from lines `6–9`. Basically, a fork is *free* when it is released while being used by a philosopher; it becomes *occupied* when grant permission requests are received from philosophers while it is being unused.

```
6:  (| forkstatus:= (0 when release when(pre_forkstatus=1))
7:       default (1 when (gright default gleft) when
8:       (pre_forkstatus=0)) default pre_forkstatus
9:   | pre_forkstatus:= forkstatus$ 1 init 0
```

The next two equations define two local signals that capture the priorities of philosophers when they simultaneously ask for access to the same fork.

```
10:  | rankr:= rankfright when reqfleft
11:  | rankl:= rankfleft when reqfright
```

The local signals above are used to specify the way a philosopher is provided with a grant permission offer by a fork. Let us consider the equation defined from line `12` to line `14`. It says that the right-hand-side philosopher is assigned a fork when either it is the only requesting philosopher and the fork is unused or there are several simultaneous requesting philosophers for an unused fork and its priority is greater than or equal to that of the other competing philosophers (i.e., the left-hand-side philosopher).

A similar definition holds for the left-hand-side philosopher to specify the way it obtains a grant permission offer from a fork (see the equation from line `15` to line `17`).

```
12:  | gright := (when(pre_forkstatus=0) when (reqfright ^-
13:      reqfleft)) default (when(pre_forkstatus=0)
14:      when(rankr>=rankl))
15:  | gleft :=  (when(pre_forkstatus=0) when (reqfleft ^-
16:      reqfright)) default (when(pre_forkstatus=0)
17:      when(rankl>rankr))
```

Remark 13.3. In the definitions of `gright` and `gleft`, one can observe the numerical comparisons between `rankr` and `rankl`: *rankr*>=*rankl* and *rankl*>*rankr*. Such expressions require both signals `rankr` and `rankl` to be synchronous. For that reason, we do not use directly the inputs `rankfright` and `rankfleft` in these equations. In fact, the clocks of `rankfright` and `rankfleft` are a priori independent of each other.

Finally, in the following synchronization constraints, lines `18` and `19` specify that a grant permission request must always be associated with a priority level. This is particularly necessary in the case of simultaneous requests for the same fork.

```
18:  | rankfright ^= reqfright
19:  | rankfleft ^= reqfleft
20:  | forkstatus ^= reqfright ^+ reqfleft ^+ release
21:  |)
22: where
23:   integer pre_forkstatus, rankl, rankr;
24: end;
```

13.2.2.2 Analysis, Code Generation, and Simulation

The analysis of the `Fork` process with the compiler is very similar to that of the `Philosopher` process presented before. The code generation and simulation are also performed on the basis of the same principles. So, all these aspects will not be detailed here.

13.2.3 Coordination Between Philosophers and Forks

To illustrate the interaction between philosophers and forks, let us consider the situation where there are five forks and five philosophers. Such a situation is modeled by the following `Dining_philosophers` process that can be seen also as the main process. To be able to automatically generate a simulation code from this process, its definition is such that it is endochronous.

The clock used for counting cycles when philosophers are eating, the hunger triggering events, and the interrupt events are assumed to be provided by the environment of the main process, i.e., they are inputs of process `Dining_philosophers`.

```
1: process Dining_philosophers =
2: ( ? boolean Clk_tick,
3:             Clk_trigP1, Clk_trigP2, Clk_trigP3
4:             Clk_trigP4, Clk_trigP5,
5:             Clk_interuptP1, Clk_interuptP2,
6:             Clk_interuptP3, Clk_interuptP4,
7:             Clk_interuptP5;
8:   ! )
```

In the above interface, these inputs are represented as Boolean signals which have the same clock. When an input signal (e.g., `Clk_trigP1`) holds the value *true*, it means that its associated event (e.g., `trigP1`) is present; otherwise its associated event is absent. This is expressed by the following equations, specified from line `9` to line `19`.

```
9:  (| tick := when Clk_tick

10:  | trigP1 := when Clk_trigP1
11:  | trigP2 := when Clk_trigP2
12:  | trigP3 := when Clk_trigP3
13:  | trigP4 := when Clk_trigP4
14:  | trigP5 := when Clk_trigP5

15:  | interuptP1 := when Clk_interuptP1
16:  | interuptP2 := when Clk_interuptP2
17:  | interuptP3 := when Clk_interuptP3
18:  | interuptP4 := when Clk_interuptP4
19:  | interuptP5 := when Clk_interuptP5
```

The next equations instantiate five philosophers, five forks, and their exchanged signals.

```
20:  | (P1needf1,P1needf2,release1,pstate1):= Philosopher{1}
21:      (tick, when Clk_f1gP1$1, when Clk_f2gP1$1, trigP1,
22:        interuptP1)
23:  | (P2needf2,P2needf3,release2,pstate2):= Philosopher{2}
24:      (tick, when Clk_f2gP2$1,when Clk_f3gP2$1, trigP2,
25:        interuptP2)
26:  | (P3needf3,P3needf4,release3,pstate3):= Philosopher{3}
```

```
27:       (tick, when Clk_f3gP3$1,when Clk_f4gP3$1, trigP3,
28:         interuptP3)
29:   | (P4needf4,P4needf5,release4,pstate4):= Philosopher{4}
30:       (tick, when Clk_f4gP4$1,when Clk_f5gP4$1, trigP4,
31:         interuptP4)
32:   | (P5needf5,P5needf1,release5,pstate5):= Philosopher{5}
33:       (tick, when Clk_f5gP5$1,when Clk_f1gP5$1, trigP5,
34:         interuptP5)

35:   | (f1gP5,f1gP1,fstate1):= Fork{1}(release5 default release1,
36:        P5needf1, P1needf1, tmp_rankof51, tmp_rankof11)
37:   | (f2gP1,f2gP2,fstate2):= Fork{2}(release1 default release2,
38:        P1needf2,P2needf2,tmp_rankof12, tmp_rankof22)
39:   | (f3gP2,f3gP3,fstate3):= Fork{3}(release2 default release3,
40:        P2needf3,P3needf3,tmp_rankof23, tmp_rankof33)
41:   | (f4gP3,f4gP4,fstate4):= Fork{4}(release3 default release4,
42:        P3needf4,P4needf4,tmp_rankof34, tmp_rankof44)
43:   | (f5gP4,f5gP5,fstate5):= Fork{5}(release4 default release5,
44:        P4needf5,P5needf5,tmp_rankof45, tmp_rankof55)

45:   | tmp_rankof11 := rankof1 when P1needf1
46:   | tmp_rankof12 := rankof1 when P1needf2
47:   | tmp_rankof22 := rankof2 when P2needf2
48:   | tmp_rankof23 := rankof2 when P2needf3
49:   | tmp_rankof33 := rankof3 when P3needf3
50:   | tmp_rankof34 := rankof3 when P3needf4
51:   | tmp_rankof44 := rankof4 when P4needf4
52:   | tmp_rankof45 := rankof4 when P4needf5
53:   | tmp_rankof55 := rankof5 when P5needf5
54:   | tmp_rankof51 := rankof5 when P5needf1

55:   | Clk_f1gP1 := f1gP1 default false
56:   | Clk_f2gP1 := f2gP1 default false
57:   | Clk_f2gP2 := f2gP2 default false
58:   | Clk_f3gP2 := f3gP2 default false
59:   | Clk_f3gP3 := f3gP3 default false
60:   | Clk_f4gP3 := f4gP3 default false
61:   | Clk_f4gP4 := f4gP4 default false
62:   | Clk_f5gP4 := f5gP4 default false
63:   | Clk_f5gP5 := f5gP5 default false
64:   | Clk_f1gP5 := f1gP5 default false

65:   | Clk_pstate1 := ^pstate1 default false
66:   | Clk_pstate2 := ^pstate2 default false
67:   | Clk_pstate3 := ^pstate3 default false
68:   | Clk_pstate4 := ^pstate4 default false
69:   | Clk_pstate5 := ^pstate5 default false

70:   | Clk_fstate1 := ^fstate1 default false
71:   | Clk_fstate2 := ^fstate2 default false
72:   | Clk_fstate3 := ^fstate3 default false
73:   | Clk_fstate4 := ^fstate4 default false
74:   | Clk_fstate5 := ^fstate5 default false
```

```
75:   | Clk_P1needf1 := P1needf1 default false
76:   | Clk_P1needf2 := P1needf2 default false
77:   | Clk_P2needf2 := P2needf2 default false
78:   | Clk_P2needf3 := P2needf3 default false
79:   | Clk_P3needf3 := P3needf3 default false
80:   | Clk_P3needf4 := P3needf4 default false
81:   | Clk_P4needf4 := P4needf4 default false
82:   | Clk_P4needf5 := P4needf5 default false
83:   | Clk_P5needf5 := P5needf5 default false
84:   | Clk_P5needf1 := P5needf1 default false
```

The next equations describe the management of priorities associated with philosophers. For that, two local signals are considered: `updaterank` and `repletePhil` (see the definitions from line `85` to line `96`). The former denotes the priority of the philosopher who has just finished eating and the latter denotes this philosopher (whose priority has to be reset). Every switch from the *eating* state to the *thinking* state resets the priorities of all philosophers.

```
85:   | repletePhil:= 1 when (pstate1$ init 0=2 and
86:       pstate1=0) default 2 when (pstate2$ init 0=2
87:       and pstate2=0) default 3 when (pstate3$ init 0=2
88:       and pstate3=0) default 4 when (pstate4$ init 0=2
89:       and pstate4=0) default 5 when (pstate5$ init 0=2
90:       and pstate5=0) default 0

91:   | updaterank:= pre_rankof1 when (repletePhil = 1)
92:       default pre_rankof2 when (repletePhil = 2)
93:       default pre_rankof3 when (repletePhil = 3)
94:       default pre_rankof4 when (repletePhil = 4)
95:       default pre_rankof5 when (repletePhil = 5)
96:       default 0
```

The equations defined between lines `97` and `121` describe the priority definition for each philosopher via their associated rank: the higher the rank number, the greater the priority.

For instance, let us focus on the first equation (i.e., line `97`). It defines the rank of the philosopher identified by `1`. Whenever this philosopher finishes eating, its corresponding rank is set to `0`; otherwise its rank is increased by one. The possible values of a rank are taken in the interval [0..4]. A similar definition is considered for the rank of the other philosophers.

Roughly, this means that a philosopher who becomes full is immediately associated with the lowest priority level so that the other philosophers can be assigned forks in the case of competition for sharing forks with this process.

Note that initially the following order is assumed for ranks:

```
rankof1 > rankof2 > rankof3 > rankof4 > rankof5
```

```
97:   | rankof1:= 0 when ((pre_rankof1=updaterank) when
98:       (repletePhil>0)) default ((pre_rankof1 + 1) modulo 5)
99      when ((pre_rankof1<=updaterank) when (repletePhil>0))
100:    default pre_rankof1
```

```
101: | pre_rankof1:= rankof1$ init 4
102: | rankof2:= 0 when ((pre_rankof2=updaterank) when
103:     (repletePhil>0)) default ((pre_rankof2 + 1) modulo 5)
104:     when ((pre_rankof2<=updaterank) when (repletePhil>0))
105:     default pre_rankof2
106: | pre_rankof2:= rankof2$ init 3
107: | rankof3:= 0 when ((pre_rankof3=updaterank) when
108:     (repletePhil>0)) default ((pre_rankof3 + 1) modulo 5)
109:     when ((pre_rankof3<=updaterank) when (repletePhil>0))
110:     default pre_rankof3
111: | pre_rankof3:= rankof3$ init 2
112: | rankof4:= 0 when ((pre_rankof4=updaterank) when
113:     (repletePhil>0)) default ((pre_rankof4 + 1) modulo 5)
114:     when ((pre_rankof4<=updaterank) when (repletePhil>0))
115:     default pre_rankof4
116: | pre_rankof4:= rankof4$ init 1
117: | rankof5:= 0 when ((pre_rankof5=updaterank) when
118:     (repletePhil>0)) default ((pre_rankof5 + 1) modulo 5)
119:     when ((pre_rankof5<=updaterank) when (repletePhil>0))
120:     default pre_rankof5
121: | pre_rankof5:= rankof5$ init 0
```

There is a unique master clock from which the clocks of all signals can be deduced. For instance, this master clock is the same as that of the Boolean input signals of process `Dining_philosophers`. All the synchronization constraints are given below.

```
122: | Clk_tick ^= Clk_trigP1 ^= Clk_trigP2 ^= Clk_trigP3 ^=
123:   Clk_trigP4 ^= Clk_trigP5 ^= Clk_interuptP1 ^=
124:   Clk_interuptP2 ^= Clk_interuptP3 ^= Clk_interuptP4 ^=
125:   Clk_interuptP5 ^= rankof1 ^= rankof2 ^= rankof3 ^=
126:   rankof4 ^= rankof5 ^=Clk_pstate1 ^= Clk_pstate2 ^=
127:   Clk_pstate3 ^= Clk_pstate4 ^= Clk_pstate5 ^=
128:   Clk_fstate1 ^= Clk_fstate2 ^= Clk_fstate3 ^=
129:   Clk_fstate4 ^= Clk_fstate5 ^= Clk_f1gP1 ^=
130:   Clk_f2gP1 ^= Clk_f2gP2 ^= Clk_f3gP2 ^= Clk_f3gP3 ^=
131:   Clk_f1gP5 ^= Clk_f4gP3 ^= Clk_f4gP4 ^= Clk_f5gP4 ^=
132:   Clk_f5gP5 ^= Clk_P1needf1 ^= Clk_P1needf2 ^=
133:   Clk_P2needf2 ^= Clk_P2needf3 ^= Clk_P3needf3 ^=
134:   Clk_P3needf4 ^= Clk_P4needf4 ^= Clk_P4needf5 ^=
135:   Clk_P5needf5 ^= Clk_P5needf1 ^= repletePhil ^=
136:   updaterank ^= rankof1 ^= rankof2 ^= rankof3 ^=
137:   rankof4 ^= rankof5
138: |)
139: where
140:  use Philosopher;
141:  use Fork;
142:  event tick, release1, release2, release3, release4,
143:   release5, P1needf1, P1needf2, P2needf2, P2needf3,
144:   P3needf3, P5needf1, P3needf4, P4needf4, P4needf5,
145:   P5needf5, trigP1, trigP2, trigP3, trigP4, trigP5,
146:   interuptP1, interuptP2, interuptP3, interuptP4,
147:   interuptP5, f1gP1, f2gP1, f2gP2, f3gP2, f3gP3, f1gP5,
148:   f4gP3, f4gP4, f5gP4, f5gP5;
```

```
149:  integer tmp_rankof11, tmp_rankof12, tmp_rankof51,
150:   tmp_rankof33, tmp_rankof22, tmp_rankof23, tmp_rankof34,
151:   tmp_rankof44, tmp_rankof45, tmp_rankof55, repletePhil,
152:   updaterank, pre_rankof1, pre_rankof2, pre_rankof3,
153:   pre_rankof4, pre_rankof5, rankof1, rankof2, rankof3,
154:   rankof4, rankof5, pstate1, pstate2, pstate3, pstate4,
155:   pstate5, fstate1, fstate2, fstate3, fstate4, fstate5;
156:  boolean Clk_pstate1, Clk_pstate2, Clk_pstate3,
157:   Clk_pstate4, Clk_pstate5, Clk_fstate1,
158:   Clk_fstate2, Clk_fstate3, Clk_fstate4,
159:   Clk_fstate5, Clk_P1needf1, Clk_P1needf2,
160:   Clk_P2needf2, Clk_P2needf3, Clk_P3needf3, Clk_P3needf4,
161:   Clk_P4needf4, Clk_P4needf5, Clk_P5needf5, Clk_P5needf1,
162:   Clk_f1gP1, Clk_f2gP1, Clk_f2gP2, Clk_f3gP2, Clk_f3gP3,
163:   Clk_f1gP5, Clk_f4gP3, Clk_f4gP4, Clk_f5gP4, Clk_f5gP5;
164: end;
```

Now, by considering the above `Dining_philosophers` process, one can use the compiler to automatically generate a code to simulate different scenarios with five concurrent philosophers and five forks, based on the same approach presented previously for the `Philosopher` process.

13.3 Exercises

13.1. Compile and simulate the whole `Dining_philosophers` process with POLYCHRONY.

Expected Learning Outcomes:

This chapter illustrated how to define a solution to a complex problem in SIGNAL and how the POLYCHRONY toolset can help the user during the design. Among the important lessons to remember are the following:

- The **decomposition** of complex problems into smaller ones, which is a general principle, must always be applied here.
- The **clock properties** are in practice more easily tractable when the specifications analyzed are not huge.
- The **incremental definition and analysis of models** are the key solution to the design of complex and huge problems.

References

1. Dijkstra EW (1971) Hierarchical ordering of sequential processes. Acta Informatica 1(2): 115–138
2. Bijoy J, Gamatié A, Suhaib S, Shukla S (2007) Dining Philosopher Problem: Implementation in SIGNAL, FERMAT Lab. Technical Report n. 2007–15, Virginia Tech (VA, USA)

Appendix A
Main Commands of the Compiler

This chapter surveys the main commands of the SIGNAL compiler and their associated options. For further information, the reader can also refer directly to the documentation distributed with the compiler.

A.1 Compilation Commands

The compilation of SIGNAL programs is mainly achieved by invoking the following command:

```
signal
```

Different options are proposed together with this command for various purposes as discussed below.

A.1.1 General Synopsis

The synopsis implemented by the compiler is

```
signal -h
signal -vers
signal [options] FILE [-par[=PFILE]] [-d=dirname]
```

with the following explanations:

- `signal -h` displays the online help documentation.
- `signal -vers` displays the used compiler version.
- `signal [options] FILE [-par[=PFILE]] [-d=dirname]` compiles a SIGNAL process or module defined in `FILE` with respect to specified options. Let us call this process or module `FOO`. When `FOO` has parameters (typically, when `FOO` is a process), the list of values for these parameters is optionally given in `PFILE`; this is the meaning of the `-par` option.

The output files resulting from the compilation are created in the `dirname` directory. When no directory is specified, the output files are generated in an automatically created subdirectory, called `FOO`.

A.1.2 Commonly Used Compiler Options

In addition to the previous options, the SIGNAL compiler also provides the following options:

- `-lis`: Creates a file `FOO_LIS.SIG` containing the result of `FOO` parsing. Typically, syntax errors are indicated in this file.
- `-tra`: Creates a file `FOO_TRA.SIG` containing the result of `FOO` transformation. This file may contain the unsatisfied clock constraint annotations.
- `-v`: Explains what is being done during the compilation (verbose).
- `-war`: Displays warning messages on the standard output, or in the `FOO_LIS.SIG` file if `-lis` is present.
- `-spec`: Creates a file `FOO_ABSTRACT.SIG` containing the result of `FOO` interface abstraction.
- `-dc+`: The program is transformed such that clocks are represented as pure events and are organized in a hierarchy.
- `-bdc+`: Creates a file `FOO_bdc_TRA.SIG`; the program is transformed such that clocks are represented as Boolean expressions (pure events no longer appear).
- `-clu`: Code partitioning with respect to a cluster-based code generation.
- `-force`: Forces the code generation by adding "exceptions" for unsatisfied constraints.
- `-check`: Generates code for assertions.
- `-c[:[i][m]]` creates ANSI C code as follows:
  - `c:` creates `FOO.c`, `FOO.h`, and the files `FOO_ext.h`, `FOO_undef.log`, and `FOO_type.h`, which are needed in `FOO.c`.
  - `c:i` creates `FOO_io.c` and the files `FOO_undef.log` and `FOO_type.h`, which are needed in `FOO_io.c`.
  - `c:m` creates `FOO_main.c`.
  - `c:im` and `c:mi` create files created by `c:i` and `c:m`
  - `c` creates files created by `c:` and `c:im`.

  - `FOO_externalsProc.h`: Created if some external functions are referred to if `FOO` contains an interface of those functions.
  - `FOO_externalsUNDEF_LIS.h`: Created if some referred to types or constants are used and not defined in `FOO`; in this case the file `FOO_externalsUNDEF.h` must be provided.
  - `FOO_types.h`: Contains C types corresponding to SIGNAL types.
  - `FOO_body.c`: Contains the code associated with each step and the scheduler.

  - FOO_body.h: Contains the interface of all functions that are generated in the file FOO_body.c.
  - FOO_io.c: Contains input–output functions associated with the interface of FOO.
  - FOO_main.c: Contains the main C program.

- -c++[:[i][m]]: Same as -c[:[i][m]] but generates C++ code.
- -java [:[i][m]]: Mostly the same as -c[:[i][m]] but generates Java code.

  - FOO.java: Contains code associated with each step and the scheduler
  - FOO_io.java: Contains input–output functions associated with the interface of FOO
  - FOO_main.java: Contains main, a Java program

- -javat [:[i][m]]: Mostly the same as -java, but threads are generated for clusters.
- -z3z: Creates the file FOO.z3z, which contains SIGALI code.
- -profiling (not fully implemented): Creates files for cost–performance evaluation.

A.1.3 Examples of Command Usage

Let us consider the C language as target source code.

1. The command:

   ```
   signal -lis FOO.SIG,
   ```

   where FOO.SIG is the name of the file containing the process FOO, produces the following messages:

   ```
   ===> Program analysis
   ===> Reduction to the kernel language
   ===> Graph generation (Process FOO)
     # Annotated source program generation: FOO_LIS.SIG
   ===> Clock calculus (Process: FOO)
   ```

   The result of this command is the file FOO_LIS.SIG, which may contain possible syntax error messages.
   In the case of a syntax error, the "Clock calculus" (the last line in the above messages) step is not displayed:

   ```
   ===> Program analysis
   ===> Reduction to the kernel language
   ===> Graph generation (Process FOO)
     # Annotated source program generation: FOO_LIS.SIG
   ```

2. The command

   ```
   signal -tra FOO.SIG,
   ```

where FOO.SIG is the name of the file containing the process FOO, produces the following messages:

```
===> Program analysis
===> Reduction to the kernel language
===> Graph generation (Process FOO)
===> Clock calculus (Process: FOO)
  # Hierarchized program generation: FOO_TRA.SIG
```

Here, the result is the generated FOO_TRA.SIG file.
When the process FOO contains a clock constraint, the following messages are obtained:

```
===> Program analysis
===> Reduction to the kernel language
===> Graph generation (Process FOO)
===> Clock calculus (Process: FOO)
  # Clocks constraints (Process: FOO)
  # Hierarchized program generation: FOO_TRA.SIG
```

The presence of any null clocks in the process is also indicated explicitly:

```
===> Program analysis
===> Reduction to the kernel language
===> Graph generation (Process FOO)
===> Clock calculus (Process: FOO)
  # program with null clock signals
  # Clocks constraints (Process: FOO)
  # Hierarchized program generation: FOO_TRA.SIG
```

3. The command

```
signal -c FOO.SIG,
```

where FOO.SIG is the name of the file containing the process FOO, produces the following messages:

```
===> Program analysis
===> Reduction to the kernel language
===> Graph generation (Process FOO)
===> Clock calculus (Process: FOO)
 ------------ Events -> Booleans     .... BEGIN -------
 ------------ Events -> Booleans     .......END -------
===> Graph processing(Process : FOO)
 ------------ Sequentializing        .... BEGIN -------
 ------------ Sequentializing        .......END -------
===> C generation (Process : FOO)
      * Externals Declarations  : FOO/FOO_externalsProc.h
      * Externals Declarations  : FOO/FOO_externalsUNDEF_LIS.h
     **** WARNING : external definitions must be defined in :
            FOO_externalsUNDEF.h (Using
                        FOO_externalsUNDEF_LIS.h file )
      * Externals Declarations  : FOO/FOO_externals.h
      * Types Declarations      : FOO/FOO_types.h
      * Main Program            : FOO/FOO_main.c
      * Instant Execution       : FOO/FOO_body.c
```

```
      * Header file (body)       : FOO/FOO_body.h
      * Input/Output procedures : FOO/FOO_io.c
```

Here, several C code files are generated for the simulation. Observe the warning message that indicates in which appropriate files external definitions must be provided (SIGNAL enables to import external programs, in the form of external processes).

4. The command

```
signal -java FOO.SIG,
```

where `FOO.SIG` is the name of the file containing the process `FOO`, produces the following messages:

```
===> Program analysis
===> Reduction to the kernel language
===> Graph generation (Process FOO)
===> Clock calculus (Process: FOO)
 ------------ Events -> Booleans     .... BEGIN -------
 ------------ Events -> Booleans     .......END -------
===> Graph processing(Process: FOO)
 ------------ Sequentializing        .... BEGIN -------
 ------------ Sequentializing        .......END -------
===> Java generation (Process: FOO)
      * Main Program             : FOO/FOO_main.java
      * Header file (body)       : FOO/FOO.java
      * Input/Output procedures : FOO/FOO_io.java
```

The generated code files can be used for a Java-based simulation.

5. The command

```
signal -z3z FOO.SIG,
```

where `FOO.SIG` is the name of the file containing the process `FOO`, produces the following messages:

```
===> Program analysis
===> Reduction to the kernel language
===> Graph generation (Process FOO)
===> Clock calculus (Process: FOO)
 ------------ Z3Z Generation .... BEGIN --------------
  # Equations over Z/3Z generation: FOO.z3z
 ------------ Z3Z Generation .......END --------------
```

The above command enables one to automatically generate an encoding of the `FOO` process in $\mathbb{Z}/3\mathbb{Z}$. The resulting `FOO.z3z` file is typically usable for model-checking with the SIGALI tool.

A.2 Automatic Makefile Generation

Beyond the previous compilation commands, the SIGNAL compiler offers the means to automatically generate `Makefile` files to produce executable forms of the simulation code.

A.2.1 Synopsis

The proposed synopsis for the `Makefile` file generation is as follows:

```
genMake -h display this help.
genMake <TARGET_LANGUAGE> FOO

genLink -h display this help.
genLink FOO1 [FOO2,....,FOOn]
```

These commands have the following meaning:

- `genMake` is a Makefile program generator for a SIGNAL program with

  ```
  <TARGET_LANGUAGE> ::= C | C++ | Java.
  ```

- `genLink` is a Makefile program generator for the linking of SIGNAL processes that have been separately compiled (useful for C and C++ languages). The main program must be specified as the first parameter (`FOO1`).

A.2.2 Examples of Command Usage

Let us consider again the C language for the target source code.

1. The command

   ```
   genMake C FOO
   ```

   produces the following messages:

   ```
   Makefile generated in the file: Makefile_FOO
   To produce executable, run the command: make -f Makefile_FOO
   To produce separate object, run the command:
                                   make separate -f Makefile_FOO
   ```

2. The command

   ```
   genLink FOO FOO2 FOO3,
   ```

   where `FOO` is the main program (first parameter) and `FOO2` and `FOO3` are other processes, produces the following messages:

   ```
   Makefile generated in the file: link_FOO
   To produce executable, run the command: make -f link_FOO.
   ```

Appendix B
The Grammar of SIGNAL

This chapter presents the grammar[1] of the SIGNAL language in Backus–Naur form syntax. The grammar covers the whole language beyond the notions introduced in this book.

In its description, the following notational conventions are considered:

- SIGNAL keywords as well as accepted character(s) are shown as underlined boldfaced words and characters, e.g.,

  my-keyword

  Note that in a SIGNAL program, the keywords may appear either totally in lowercase or totally in uppercase letters.
- To facilitate the understanding of the rules of the grammar, additional contextual information is sometimes used as a suffix of nonterminal strings. The contextual information is set in italics in the following form:

  non-terminal-*contextual-information*

  In the grammar, this is, for instance, the case for the nonterminal Name, which allows one to designate a signal (or a group of signals), a parameter, a constant, a type, a model (e.g., a process), a module, or a directive.
- Optional elements are shown between square brackets, as follows:

  [my-optional-elements]

- A possibly empty list of elements is represented between braces, in the following way:

  { my-possibly-empty-list-of-elements }.

- The set notation

  set1\set2

  denotes any element obtained from set1 which is not an element of set2, i.e., the difference between two sets.

[1] This grammar can be found online at http://www.irisa.fr/espresso/Polychrony.

The grammar of SIGNAL version 4

```
MODULE ::=
   module Name-model =
          [ DIRECTIVES ]
          DECLARATION { DECLARATION }
   end ;

DECLARATION ::=
   MODEL
   | DECLARATIO            [ DIRECTIVES ]
          [ BODY ] ;
   | private action Name-model =
          INTERFACE-DEF
          [ DIRECTIVES ]
          [ BODY ] ;

NODE ::=
   node Name-model =
          INTERFACE-DEF
          [ DIRECTIVES ]
          [ BODY ] ;
   | private node Name-model =
          INTERFACE-DEF
          [ DIRECTIVES ]
          [ BODY ] ;

FUNCTION ::=
   function Name-model =
          INTERFACE-DEF
          [ DIRECTIVES ]
          [ BODY ] ;
   | private function Name-model =
          INTERFACE-DEF
          [ DIRECTIVES ]
          [ BODY ] ;

INTERFACE-DEF ::=
   INTERFACE
   | Name-model-type

INTERFACE ::=
   [ PARAMETERS ] ( INPUTS OUTPUTS ) EXTERNAL-GRAPH

PARAMETERS ::=
   { { FORMAL-PARAMETER } }

FORMAL-PARAMETER ::=
   S-DECLARATION
   | DECLARATION-OF-TYPES
   | FORMAL-MODEL
```

```
FORMAL-MODEL ::=
   process Name-model-type Name-model
   | action Name-model-type Name-model
   | node Name-model-type Name-model
   | function Name-model-type Name-model

INPUTS ::=
   ? { S-DECLARATION }

OUTPUTS ::=
   ! { S-DECLARATION }

EXTERNAL-GRAPH ::=
   [ PROCESS-ATTRIBUTE ] [ SPECIFICATION-OF-PROPERTIES ]

PROCESS-ATTRIBUTE ::=
   safe
   | deterministic
   | unsafe

SPECIFICATION-OF-PROPERTIES ::=
   spec GENERAL-PROCESS

DIRECTIVES ::=
   pragmas PRAGMA { PRAGMA } end pragmas

PRAGMA ::=
   Name-pragma [ ( PRAGMA-OBJECT { , PRAGMA-OBJECT } ]
   [ Pragma-stm ]

PRAGMA-OBJECT ::=
   Label
   | Name

Pragma-stm ::=
   String-cst

BODY ::=
   DESCRIPTION-OF-MODEL

DESCRIPTION-OF-MODEL ::=
   GENERAL-PROCESS
   | EXTERNAL-NOTATION

EXTERNAL-NOTATION ::=
   external [ String-cst ]

DECLARATION-BLOCK ::=
   where DECLARATION { DECLARATION } end

DECLARATION-OF-TYPES ::=
   type DEFINITION-OF-TYPE { , DEFINITION-OF-TYPE } ;
   | private type DEFINITION-OF-TYPE { , DEFINITION-OF-TYPE } ;
```

```
DEFINITION-OF-TYPE ::=
   Name-type
   | Name-type = DESCRIPTION-OF-TYPE
   | process Name-model-type = INTERFACE-DEF [ DIRECTIVES ]
   | action Name-model-type = INTERFACE-DEF [ DIRECTIVES ]
   | node Name-model-type = INTERFACE-DEF [ DIRECTIVES ]
   | function Name-model-type = INTERFACE-DEF [ DIRECTIVES ]

DESCRIPTION-OF-TYPE ::=
   SIGNAL-TYPE
   | EXTERNAL-NOTATION [ TYPE-INITIAL-VALUE ]

TYPE-INITIAL-VALUE ::=
   init Name-constant

SIGNAL-TYPE ::=
   Scalar-type
   | External-type
   | ENUMERATED-TYPE
   | ARRAY-TYPE
   | TUPLE-TYPE
   | Name-type

Scalar-type ::=
   Synchronization-type
   | Numeric-type
   | Alphabetic-type

Synchronization-type ::=
   event
   | boolean

Numeric-type ::=
   Integer-type
   | Real-type
   | Complex-type

Integer-type ::=
   short
   | integer
   | long

Real-type ::=
   real
   | dreal

Complex-type ::=
   complex
   | dcomplex

Alphabetic-type ::=
   char
   | string
```

```
External-type ::=
   Name-type

ENUMERATED-TYPE ::=
   enum ( Name-enum-value { , Name-enum-value } )

ARRAY-TYPE ::=
   [ S-EXPR { , S-EXPR } ] SIGNAL-TYPE

TUPLE-TYPE ::=
   struct ( NAMED-FIELDS )
   | bundle ( NAMED-FIELDS ) [ SPECIFICATION-OF-PROPERTIES ]

NAMED-FIELDS ::=
   S-DECLARATION { S-DECLARATION }

DECLARATION-OF-CONSTANTS ::=
   constant SIGNAL-TYPE CONSTANT-DEF { , CONSTANT-DEF } ;
   | constant CONSTANT-DEF { , CONSTANT-DEF } ;
   | private constant SIGNAL-TYPE CONSTANT-DEF
                    { , CONSTANT-DEF } ;
   | private constant CONSTANT-DEF { , CONSTANT-DEF } ;

CONSTANT-DEF ::=
   Name-constant
   | Name-constant = DESCRIPTION-OF-CONSTANT

DESCRIPTION-OF-CONSTANT ::=
   S-EXPR
   | EXTERNAL-NOTATION

S-DECLARATION ::=
   SIGNAL-TYPE SEQUENCE-DEF { , SEQUENCE-DEF } ;
   | SEQUENCE-DEF { , SEQUENCE-DEF } ;

SEQUENCE-DEF ::=
   Name-signal
   | Name-signal init S-EXPR

DECLARATION-OF-SHARED-VARIABLES ::=
   shared SIGNAL-TYPE SEQUENCE-DEF { , SEQUENCE-DEF } ;
   | shared SEQUENCE-DEF { , SEQUENCE-DEF } ;

DECLARATION-OF-STATE-VARIABLES ::=
   statevar SIGNAL-TYPE SEQUENCE-DEF { , SEQUENCE-DEF } ;
   | statevar SEQUENCE-DEF { , SEQUENCE-DEF } ;

DECLARATION-OF-LABELS ::=
   label Name-label { , Name-label } ;

REFERENCES ::=
   ref Name-signal { , Name-signal } ;
```

```
P-EXPR ::=
   GENERAL-PROCESS
   | ELEMENTARY-PROCESS
   | LABELLED-PROCESS
   | HIDING

GENERAL-PROCESS ::=
   CONFINED-PROCESS
   | COMPOSITION
   | CHOICE-PROCESS
   | ASSERTION-PROCESS
   | ITERATION-OF-PROCESSES

ELEMENTARY-PROCESS ::=
   INSTANCE-OF-PROCESS
   | DEFINITION-OF-SIGNALS
   | CONSTRAINT
   | DEPENDENCES

LABELLED-PROCESS ::=
   Label :: P-EXPR

Label ::=
   Name

HIDING ::=
   GENERAL-PROCESS / Name-signal { , Name-signal }
   | HIDING / Name-signal { , Name-signal }

CONFINED-PROCESS ::=
   GENERAL-PROCESS DECLARATION-BLOCK

COMPOSITION ::=
   (| [ P-EXPR { | P-EXPR } ] |)

CHOICE-PROCESS ::=
   case Name-signal in CASE { CASE } [ ELSE-CASE ] end

CASE ::=
   ENUMERATION-OF-VALUES : GENERAL-PROCESS

ELSE-CASE ::=
   else GENERAL-PROCESS

ENUMERATION-OF-VALUES ::=
   { S-EXPR { , S-EXPR } }
   | [. [ S-EXPR ] , [ S-EXPR ] .]
   | [. [ S-EXPR ] , [ S-EXPR ] [.
   | .] [ S-EXPR ] , [ S-EXPR ] .]
   | .] [ S-EXPR ] , [ S-EXPR ] [.

ASSERTION-PROCESS ::=
   assert (| [ CONSTRAINT { | CONSTRAINT } ] |)
```

```
ITERATION-OF-PROCESSES ::=
   array ARRAY-INDEX of P-EXPR [ ITERATION-INIT ] end
   | iterate ITERATION-INDEX of P-EXPR [ ITERATION-INIT ] end

ARRAY-INDEX ::=
   Name to S-EXPR

ITERATION-INDEX ::=
   DEFINED-ELT
   | ( DEFINED-ELT { , DEFINED-ELT } )
   | S-EXPR

ITERATION-INIT ::=
   with P-EXPR

INSTANCE-OF-PROCESS ::=
   EXPANSION
   | Name-model ( )
   | PRODUCTION
   | assert ( S-EXPR )

EXPANSION ::=
   Name-model { S-EXPR-PARAMETER { , S-EXPR-PARAMETER } }

PRODUCTION ::=
   MODEL-REFERENCE ( S-EXPR { , S-EXPR } )

MODEL-REFERENCE ::=
   EXPANSION
   | Name-model

S-EXPR-PARAMETER ::=
   S-EXPR
   | SIGNAL-TYPE
   | Name-model

DEFINITION-OF-SIGNALS ::=
   DEFINED-ELT := S-EXPR
   | ( DEFINED-ELT { , DEFINED-ELT } ) := S-EXPR
   | DEFINED-ELT ::= S-EXPR
   | ( DEFINED-ELT { , DEFINED-ELT } ) ::= S-EXPR
   | DEFINED-ELT ::= defaultvalue S-EXPR
   | ( DEFINED-ELT { , DEFINED-ELT } ) ::= defaultvalue S-EXPR

DEFINED-ELT ::=
   COMPONENT
   | COMPONENT [ S-EXPR { , S-EXPR } ]

COMPONENT ::=
   Name-signal
   | Name-signal . COMPONENT

CONSTRAINT ::=
   S-EXPR { ^= S-EXPR }
   | S-EXPR { ^< S-EXPR }
```

```
    | S-EXPR { ^> S-EXPR }
    | S-EXPR { ^# S-EXPR }
    | S-EXPR :=: S-EXPR

DEPENDENCES ::=
    SIGNALS { --> SIGNALS }
    | { SIGNALS --> SIGNALS } when S-EXPR

SIGNALS ::=
    ELEMENTARY-SIGNAL
    | { ELEMENTARY-SIGNAL { , ELEMENTARY-SIGNAL } }

ELEMENTARY-SIGNAL ::=
    DEFINED-ELT
    | Label

S-EXPR ::=
    INSTANCE-OF-PROCESS
    | CONVERSION
    | S-EXPR-DYNAMIC
    | S-EXPR-TEMPORAL
    | S-EXPR-CLOCK
    | S-EXPR-BOOLEAN
    | S-EXPR-ARITHMETIC
    | S-EXPR-CONDITION
    | S-EXPR-TUPLE
    | S-EXPR-ARRAY
    | S-EXPR-ELEMENTARY
    | S-EXPR \\ S-EXPR
    | ( S-EXPR )

CONVERSION ::=
    Type-conversion ( S-EXPR )

Type-conversion ::=
    Scalar-type
    | Name-type

S-EXPR-DYNAMIC ::=
    SIMPLE-DELAY
    | WINDOW
    | GENERALIZED-DELAY

SIMPLE-DELAY ::=
    S-EXPR $ [ init S-EXPR ]

WINDOW ::=
    S-EXPR window S-EXPR [ init S-EXPR ]

GENERALIZED-DELAY ::=
    S-EXPR $ S-EXPR [ init S-EXPR ]

S-EXPR-TEMPORAL ::=
    MERGING
    | EXTRACTION
```

```
    | MEMORIZATION
    | VARIABLE
    | COUNTER

MERGING ::=
   S-EXPR default S-EXPR

EXTRACTION ::=
   S-EXPR when S-EXPR

MEMORIZATION ::=
   S-EXPR cell S-EXPR [ init S-EXPR ]

VARIABLE ::=
   var S-EXPR [ init S-EXPR ]

COUNTER ::=
   S-EXPR after S-EXPR
    | S-EXPR from S-EXPR
    | S-EXPR count S-EXPR

S-EXPR-CLOCK ::=
   SIGNAL-CLOCK
    | CLOCK-EXTRACTION
    | ^0
    | S-EXPR ^+ S-EXPR
    | S-EXPR ^- S-EXPR
    | S-EXPR ^* S-EXPR

SIGNAL-CLOCK ::=
   ^ S-EXPR

CLOCK-EXTRACTION ::=
   when S-EXPR

S-EXPR-BOOLEAN ::=
   not S-EXPR
    | S-EXPR or S-EXPR
    | S-EXPR and S-EXPR
    | S-EXPR xor S-EXPR
    | RELATION

RELATION ::=
   S-EXPR = S-EXPR
    | S-EXPR /= S-EXPR
    | S-EXPR > S-EXPR
    | S-EXPR >= S-EXPR
    | S-EXPR < S-EXPR
    | S-EXPR <= S-EXPR
    | S-EXPR == S-EXPR
    | S-EXPR <<= S-EXPR

S-EXPR-ARITHMETIC ::=
   S-EXPR + S-EXPR
```

```
    | S-EXPR - S-EXPR
    | S-EXPR * S-EXPR
    | S-EXPR / S-EXPR
    | S-EXPR modulo S-EXPR
    | S-EXPR ** S-EXPR
    | + S-EXPR
    | - S-EXPR
    | DENOTATION-OF-COMPLEX

DENOTATION-OF-COMPLEX ::=
   S-EXPR @ S-EXPR

S-EXPR-CONDITION ::=
   if S-EXPR then S-EXPR else S-EXPR

S-EXPR-TUPLE ::=
   TUPLE-ENUMERATION
   | TUPLE-FIELD

TUPLE-ENUMERATION ::=
   ( S-EXPR { , S-EXPR } )

TUPLE-FIELD ::=
   S-EXPR . Name-field

S-EXPR-ARRAY ::=
   ARRAY-ENUMERATION
   | CONCATENATION
   | ITERATIVE-ENUMERATION
   | INDEX
   | MULTI-INDEX
   | ARRAY-ELEMENT
   | SUB-ARRAY
   | ARRAY-RESTRUCTURATION
   | SEQUENTIAL-DEFINITION
   | TRANSPOSITION
   | ARRAY-PRODUCT
   | REFERENCE-SEQUENCE

ARRAY-ENUMERATION ::=
   [ S-EXPR { , S-EXPR } ]

CONCATENATION ::=
   S-EXPR |+ S-EXPR

ITERATIVE-ENUMERATION ::=
   [ PARTIAL-DEFINITION { , PARTIAL-DEFINITION } ]

PARTIAL-DEFINITION ::=
   ELEMENT-DEF
   | ITERATION

ELEMENT-DEF ::=
   [ S-EXPR { , S-EXPR } ] : S-EXPR
```

```
ITERATION ::=
   { PARTIAL-ITERATION { , PARTIAL-ITERATION } } : ELEMENT-DEF
   | { PARTIAL-ITERATION { , PARTIAL-ITERATION } } : S-EXPR

PARTIAL-ITERATION ::=
   [ Name ] [ in S-EXPR ] [ to S-EXPR ] [ step S-EXPR ]

INDEX ::=
   S-EXPR .. S-EXPR [ step S-EXPR ]

MULTI-INDEX ::=
   << S-EXPR { , S-EXPR } >>

ARRAY-ELEMENT ::=
   S-EXPR [ S-EXPR { , S-EXPR } ]
   | S-EXPR [ S-EXPR { , S-EXPR } ] ARRAY-RECOVERY

ARRAY-RECOVERY ::=
   \\ S-EXPR

SUB-ARRAY ::=
   S-EXPR [ S-EXPR { , S-EXPR } ]

ARRAY-RESTRUCTURATION ::=
   S-EXPR : S-EXPR

SEQUENTIAL-DEFINITION ::=
   S-EXPR next S-EXPR

TRANSPOSITION ::=
   tr S-EXPR

ARRAY-PRODUCT ::=
   S-EXPR *. S-EXPR

REFERENCE-SEQUENCE ::=
   S-EXPR [ ? ]

S-EXPR-ELEMENTARY ::=
   CONSTANT
   | Name-signal
   | Label
   | Name-state-variable ?

CONSTANT ::=
   ENUM-CST
   | Boolean-cst
   | Integer-cst
   | Real-cst
   | Character-cst
   | String-cst

ENUM-CST ::=
   # Name-enum-value
```

```
   | Name-type # Name-enum-value
Boolean-cst ::=
   true
   | false

Integer-cst ::=
   numeral-char { numeral-char }

Real-cst ::=
   Simple-precision-real-cst
   | Double-precision-real-cst

Simple-precision-real-cst ::=
   Integer-cst Simple-precision-exponent
   | Integer-cst . Integer-cst [ Simple-precision-exponent ]

Double-precision-real-cst ::=
   Integer-cst Double-precision-exponent
   | Integer-cst . Integer-cst Double-precision-exponent

Simple-precision-exponent ::=
   e Relative-cst
   | E Relative-cst

Double-precision-exponent ::=
   d Relative-cst
   | D Relative-cst

Relative-cst ::=
   Integer-cst
   | + Integer-cst
   | - Integer-cst

Character-cst ::=
   ' Character-cstCharacter '

Character-cstCharacter ::=
   Character \ character-spec-char

character-spec-char ::=
   '
   | long-separator

String-cst ::=
   " { String-cstCharacter } "

String-cstCharacter ::=
   Character \ string-spec-char

string-spec-char ::=
   "
   | long-separator

Name ::=
```

```
   begin-name-char { name-char }
begin-name-char ::=
   name-char \ numeral-char

name-char ::=
   letter-char
   | numeral-char
   | _

letter-char ::=
   upper-case-letter-char
   | lower-case-letter-char
   | other-letter-char

upper-case-letter-char ::=
   A | B | C | D | E | F | G | H | I | J | K | L | M
   | N | O | P | Q | R | S | T | U | V | W | X | Y | Z

lower-case-letter-char ::=
   a | b | c | d | e | f | g | h | i | j | k | l | m
   | n | o | p | q | r | s | t | u | v | w | x | y | z

other-letter-char ::=
   À | Á | Â | Ã | Ä | Å | Æ | Ç | È | É | Ê | Ë | Ì
   | Í | Î | Ï | Ð | Ñ | Ò | Ó | Ô | Õ | Ö | Ø | Ù
   | Ú | Û | Ü | Ý | Þ | ß | à | á | â | ã | ä | å
   | æ | ç | è | é | ê | ë | ì | í | î | ï | ð | ñ
   | ò | ō | ô | õ | ö | ø | ù | ú | û | ü | ý | þ
   | ÿ

numeral-char ::=
   0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Comment ::=
   % { CommentCharacter } %

CommentCharacter ::=
   Character \ comment-spec-char

comment-spec-char ::=
   %

Character ::=
   character
   | CharacterCode

character ::=
   name-char
   | mark
   | delimitor
   | separator
   | other-character
mark ::=
   . | ' | " | % | : | = | < | > | + | - | * | / | @
```

```
   | $ | ^ | # | | | \
delimitor ::=
   ( | ) | { | } | [ | ] | ? | ! | , | ;

separator ::=
   \x20
   | long-separator

long-separator ::=
   \x9 | \xA | \xC | \xD

CharacterCode ::=
   OctalCode
   | HexadecimalCode
   | escape-code

OctalCode ::=
   \ octal-char [ octal-char [ octal-char ] ]

octal-char ::=
   0 | 1 | 2 | 3 | 4 | 5 | 6 | 7

HexadecimalCode ::=
   \x hexadecimal-char [ hexadecimal-char ]

hexadecimal-char ::=
   numeral-char
   | A | B | C | D | E | F | a | b | c | d | e | f

escape-code ::=
   \a | \b | \f | \n | \r | \t | \v | \\ | \" | \' | \? | \%

prefix-mark ::=
   \
```

Glossary

Abstract clock. Given a logical time reference, the abstract clock of a signal consists of the set of logical instants at which this signal occurs and holds a value.

ARGOS. A variant of the STATECHARTS language, which adopts the semantics of synchronous languages.

Clock calculus. The clock calculus is a static analysis phase during the compilation of SIGNAL programs that aims (1) to determine the existence of a clock hierarchy with a single root (or master) clock from which all the other clocks of a process can be extracted and (2) to verify the consistency of clock relations between the signals involved in this process.

Clock hierarchy. The set inclusion relation expresses the fact that a clock is a subset of another clock in terms of sets of logical instants. The clock hierarchy is a hierarchical structure resulting from the ordering of all the clocks in a process, following the clock inclusion relation.

Conditional dependency graph. The conditional dependency graph consists of a labeled directed graph, associated with a process, such that (1) vertices are signals or clock variables, (2) arcs represent dependency relations, and (3) labels are clock expressions specifying when the associated dependency relations are valid.

Embedded system. An embedded system is a special-purpose computer system formed of software and hardware components that are subject to physical constraints from the system's environment and execution platform.

Endochrony. Endochrony is a property that characterizes a process for which the clock calculus produces a clock hierarchy with a unique master clock.

Endo-isochrony. Endo-isochrony is a constructive variant of the isochrony criterion, which requires that the communication part in the composition of two endochronous processes itself be endochronous. It is a sufficient but not necessary criterion to have an isochronous pair of processes.

ESTEREL. A textual imperative synchronous language, dedicated to control-dominated applications.

Exochrony. Exochrony is a property that characterizes a process for which the clock calculus produces a clock hierarchy with several local master clocks that cannot be hierarchized under a single root clock.

Hierarchized conditional dependency graph. The hierarchized conditional dependency graph is a conditional dependency graph enriched with clock hierarchy information. It is used as an efficient internal representation of SIGNAL processes for optimized code generation.

Isochrony. Isochrony is a property that characterizes the fact that the synchronous composition of two processes is equivalent to their asynchronous composition (i.e., involving *send*/*receive* communications). It is a useful notion in the synchronous design of distributed (asynchronous) systems.

Logical time. The logical time is a discretization of the operational behavior of a system resulting from the synchrony hypothesis, where the behaviors associated with each fragment are executed instantaneously. It is an abstraction of the usual asynchronous vision, which specifies the behavior of a system as a continuous succession of computations and synchronization phases.

LUCID SYNCHRONE. A textual higher-order functional variant of the LUSTRE language.

LUSTRE. A textual declarative and functional synchronous language, dedicated to dataflow-oriented applications.

Monoclocked system. A monoclocked system is a system such that the activations of all its components are governed by a single global clock, also referred to as its master clock.

Multiclocked system. A multiclocked system is a system with several components, where each component holds its own activation clock and no master clock exists in the system.

Reactive system. A reactive system is a system that continuously interacts with its environment, and the rhythm at which the system reacts is dictated by the environment.

Real-time system. A real-time system is a system whose correctness depends not only on the logical results of its associated computations, but also on the delay after which these results are produced. In particular, this delay must satisfy the temporal constraints imposed by the environment of the system.

Polychrony. In the jargon of SIGNAL, polychrony is an abstract mathematical time model in terms of partial order relations, used to describe multiclocked systems. The same word is used to refer the design environment of SIGNAL; in that case the name syntax usually includes only capital letters such as POLYCHRONY.

POLYCHRONY. The design environment of the SIGNAL language, composed of a graphical user interface, a compiler, and a model checker (SIGALI).

Process. A process is a set of equations for signals specifying relations between values, on the one hand, and clocks, on the other hand, of the signals involved.

Program. A program in SIGNAL is a process in which all required signal definitions are described so that this process can be executed.

SCADE. A graphical and industrial version of the LUSTRE design environment.

SIGALI. A model-checking and discrete controller synthesis tool connected to POLYCHRONY.

Signal. A signal is a totally ordered sequence of typed values. This sequence of values is observed over a logical time reference such that a value may occur or not occur at a logical instant.

SIGNAL. A textual declarative and relational synchronous language, dedicated to dataflow oriented applications.

Sildex. A graphical and industrial version of the SIGNAL language (also referred to as RT-Builder).

Specification. A SIGNAL specification is a process that expresses data dependencies and/or clock constraints on signals. However, in contrast to a program, a specification is not necessarily an executable process since its signals may be only partially defined.

STATECHARTS. A graphical state diagram based formalism, dedicated to control-oriented applications.

SYNCCHARTS. A variant of STATECHARTS with the same semantics as synchronous languages (also used as a graphical version of the ESTEREL language).

Synchronous composition (of processes). The synchronous composition of two processes is an operation that yields a new process such that for all signals common to the composed processes their carried values must be identical at the same occurring logical instants. In other words, a value agreement is imposed for common signals when they occur at the same logical instants.

Synchrony hypothesis. Given a reactive system, the synchrony hypothesis states that in the presence of input events, the system reacts fast enough to produce the corresponding output events before its next input events arrive. Thus, the computations and communications implied by such a reaction can be considered as instantaneous; i.e., the durations of computations and communications are abstracted.

Solutions to Exercises

Exercises in Chapter 3

3.1

Indicate whether or not the following affirmations are true and justify your answer:

1. A signal is necessarily present whenever it holds a value different from $\perp$.
2. A constant signal is always present.
3. When a signal becomes absent, it implicitly keeps its previously carried value.
4. A signal of `event` type and a signal of `boolean` type are exactly the same.
5. The abstract clock of a signal defines the set of instants at which the signal occurs.
6. SIGNAL assumes a reference clock that enables one to always decide the presence/absence of any defined signal.
7. In the expression s_n `:= R(`s_1`,...,`s_{n-1}`)`, where `R` is an instantaneous relation, if the signal s_1 is absent while all other arguments of `R` are present, s_n is calculated by considering some default value depending on the type of s_1.
8. In the expression s_2 `:=` s_1 `$ 1 init c`, the signal s_2 may occur with the latest value of s_1 while s_1 is absent.
9. In the expression s_3 `:=` s_1 `default` s_2, the signals s_1 and s_2 must have exclusive clocks.
10. In the expression s_3 `:=` s_1 `or` s_2,
    a. s_3 is true when s_1 is true and s_2 is absent.
    b. s_3 is true when s_1 is true and s_2 is false.
    c. s_3 is false when s_1 is absent and s_2 is false.
    d. s_3 is false when s_1 is absent and s_2 is absent.

▷

1. Yes, a signal that is present necessarily carries a value different from $\perp$.
2. No, this depends on the context.
3. No, an absent signal holds no value.
4. No, a signal of `event` type is always *true* whenever present.
5. Yes, see the definition given in Sect. 3.2.2, on page 54.

6. No, SIGNAL adopts a multiclocked vision of systems.
7. No, all signals involved must have the same clock.
8. No, all signals involved must have the same clock.
9. No, there could be instants at which both s_1 and s_2 are present.
10. a. No, all signals involved must have the same clock.
    b. Yes.
    c. No, all signals involved must have the same clock.
    d. No, all signals involved must have the same clock.

3.2

Indicate the possible type for each signal in the following expressions and simplify each expression:

1. (E_1) : `(s when s) default s`
2. (E_2) : `(s1 when b) default (s2 when b)`
3. (E_3) : `if s1 > s2 then true else false`

▷

1. `s` and the result of (E_1) : are either of `boolean` or of `event` types. (E_1) : is simplified into `s`.
2. `s1` and `s2` are of any type, `b` is of `boolean` type, and (E_2) : has the same type as `s1`. (E_2) : is simplified to `s1 when b`.
3. `s1` and `s2` are of numeric type and (E_3) : is of `boolean` type. (E_3) : is simplified to `s1 > s2`.

3.3

In the following expressions, where `s1` and `s2` are two signals independent of each other, `f` and `g` are instantaneous functions and `cond` is a Boolean function.

- (E_1) : `(f(s1) when cond(s2)) default g(s1)`
- (E_2) : `(f(s2) when cond(s1)) default g(s1)`

Determine the clock of expressions E_1 and E_2.

▷

- (E_1) : clock of `s1`
- (E_2) : clock of `s1`

3.4

Let us consider the following expression:

```
(A when B) default C.
```

Its evaluation can lead to different results. For instance, if signals `A` and `B` are absent but signal `C` is not absent, the expression takes the value of `C`. Given that `B` is a Boolean signal and `A` and `C` are of any type, enumerate all possible cases for the evaluation of that expression.

▷

- If `A` is present and `B` is *true*, the result is `A` (this scenario includes two cases: `C` is present or absent).

- Otherwise if C is present, the result is C (this scenario includes five cases: A is absent and B is either *true* or *false* or absent; A is present and B is either *false* or absent).
- Otherwise the result is ⊥ (five cases: the previous five cases where C is also absent).

3.5

Let us consider the following expression:

```
b := true when c default false.
```

- When is this expression correct regarding clock constraints?
- When can it be simplified to `b := c`?

▷

- It is valid only (1) when the clock of b is defined elsewhere (in a system of equations containing this expression) and (2) if it can be proved that the clock of b is at least equal to the clock of c.
- The simplification `b:= c` is only possible when b and c have the same clock.

3.6

Let s1 and s2 be some Boolean signals. They are assumed to have independent clocks. Define a signal of integer type whose value is determined as follows:

- 1 if s1 is *true*
- 2 if s2 is *true*
- 3 if s1 and s2 are both *true*
- Absent otherwise

▷

```
s3 := (3 when s1 when s2) default
      (2 when s2) default
      (1 when s1)
```

Or alternatively,

```
s3 := (3 when s1 when s2) default
      (1 when s1) default
      (2 when s2).
```

3.7

Indicate the possible type for the signal s in the following expressions and give the clock of each expression:

1. `((not s) when (not s)) default s`
2. `(s when s) default (not s)`

▷

The following types are possible for the signal s: `event` and `boolean`.
The clock of the above expressions is as follows:

1. (E_1) : clock of s
2. (E_2) : clock of s

Exercises in Chapter 4

4.1

Define a process `Sum` in which on each occurrence of its unique input `a` of `real` type, the sum of the values of `a` that have occurred until now is computed in the output signal `s`.
▷

```
process Sum =
 ( ? real a;
   ! real s; )
 (| s := (s $ init 0.0) + a |);
```

4.2

Define a process `Average` in which on each occurrence of its unique input `a` of `real` type, the average of the values of `a` that have occurred until now is computed in the output signal `avg`.
▷

```
process Average =
 ( ? real a;
   ! real avg; )
 (| s := (s $ init 0.0) + a
  | cnt := (cnt $ init 0) + 1
  | avg := s / cnt %synchronizes cnt with s and a %
  |)
 where
   integer cnt; real s;
 end;
```

4.3

Given a constant parameter `N` of `integer` type, define a process `AverageN` in which on each occurrence of its unique input `A` of `real` type, the average of the `N` previous values of `A` is computed in the output signal `avg`.
Note: `a$N` may be initialized to 0.0 by using the following expression: `init[to N:0.0]`.
▷

```
process AverageN =
 { integer N; }
```

```
 ( ? real a;
   ! real avg; )
 (| s := (s $ init 0.0) - (a $ N init [{to N}: 0.0]) + a
  | avg := s / N
  |)
 where
  real s;
end;
```

4.4

Let a be a positive integer signal. Define a signal max in a process Maximum which takes at the current logical instant the most recent maximum value held by a.

▷

```
process Maximum =
 ( ? integer a;
   ! integer max; )
 (| zmax := max$ init 0
  | max := a when(a > zmax) default zmax
  |)
 where
  integer zmax;
 end;
```

4.5

Define a subtraction between A and B of event types which gives instants where A is present but B is not.

▷

```
process Subtraction =
 ( ? event a, b;
   ! event sub; )
 (| not_b_a := not b default a
  | sub := a when not_b_a
  |)
 where
  event not_b_a;
 end;
```

4.6

Indicate among all the above processes which ones are also functions.

▷

- Exercise 4.1: Process Sum is a function.
- Exercise 4.2: Process Average is a function.
- Exercise 4.3: Process AverageN is a function.
- Exercise 4.4: Process Maximum is not a function.
- Exercise 4.5: Process Subtraction is not a function.

Exercises in Chapter 5

5.1

Discuss the difference that may exist between the following two statements:

- `when ((c=1) default (c=2) default (c=3))`
- `when (c=1) default when (c=2) default when (c=3)`

▷

In the first statement, the fact that `c` can take the values 2 and 3 is never taken into account. In fact, all arguments of the `default` operators have the same clock. If `c` is present whatever its value is, only the expression `(c=1)` is evaluated since it has the same clock as `c`!

To enable an evaluation of expressions `(c=2)` and `(c=3)`, one must consider the second statement, in which the different arguments of the `default` operators do not have the same clock.

5.2

Simplify the following SIGNAL expressions into equivalent expressions:

1. `when (not ^s) default ^s`
2. `s when (^s)`

▷

1. `^s`
2. `s`

5.3

Let `s` be a signal of `event` type.

1. Simplify the expression `true when (^s)`.
2. Is it possible to detect the instants at which `s` is absent by considering the expression `when not (^s)`?

▷

1. `^s`.
2. No. The expression `not (^s)` is always evaluated to *false*; it follows that `when not (^s)` is always evaluated to $\perp$. As a result, it should not be considered to check the status of `s`.

   A way to check the absence of `s` may consist in defining a Boolean intermediate signal that is *true* when `s` is present, and is *false* otherwise. The clock of this Boolean signal must be greater than that of `s`.

5.4

Let us consider a very simple model of a microwave oven, which has a unique button. This button enables one to start and stop the oven. A signal `button` of `event` type is associated with this button.

Define a process that emits a Boolean output signal `busy`, which has the same clock as `button`. The value of `busy` reflects the state of the oven: *true* for active and *false* for idle.

▷

```
process Oven =
 ( ? event button;
   ! boolean buzy; )
 (| button ^= buzy
  | buzy := not(busy$ init false)
  |)
```

5.5

Let us consider a positive `integer` signal denoted by `s`.

1. Define a Boolean signal `first` which holds the value *true* only on the first occurrence of `s` and *false* otherwise (try to find a solution without using a counter).
2. Define a signal `up` of `event` type that is present whenever the current value carried by `s` is greater than the previous one. Any two consecutive values of `s` are assumed to be different. For the initial comparison, suppose that the value preceding the initial value of `s` is 0.
3. Define a signal `top` of event type which is present whenever a local maximum is reached among values of `s`. This can be detected on the occurrence of the first decreasing value of `s` after the values had previously been increasing. Any two consecutive values of `s` are still assumed to be different.
4. Let us assume that `s` may hold consecutive identical values. There is a local maximum when the values of `s` decrease after previously increasing and before they become stable (i.e., identical values). Define the local maximal signal `top_bis` in this case.
5. Define a signal `local_max` that only takes the values of `s`, which represents local maxima.

▷

- ```
  process First_True =
    ( ? integer s;
      ! boolean first; )
    (| first := b $ init true
     | b := not (^s)
     |)
   where
     boolean b;
   end;
  ```
- ```
  process Up_Events =
    ( ? integer s;
      ! event up; )
    (| pre_s := s $ init 0
     | up := when (s > pre_s)
     |)
   where
     event pre_s;
  end;
  ```

- ```
  process Top_Events
    (? integer s;
     ! event top; )
    (| (| pre_s := s $ init 0
       | up := (s > pre_s)
       | pre_up := up$ init true
       |)
     | top := when(pre_up and (not up))
     |)
    where
     boolean up, pre_up;
     integer pre_s;
  end;
  ```
- ```
  process Top_Event_Bis
    (? integer s;
     ! event top_bis; )
    (| (| pre_s := s $ init 0
       | up := (s > pre_s) or ((s = pre_s) and pre_up)
       | pre_up := up$ init true
       |)
     | top_bis := when(pre_up and (not up))
     |)
    where
     boolean up, pre_up;
     integer pre_s;
  end;
  ```
- ```
  process Local_Maximum
    (? integer s;
     ! integer local_max; )
    (| (| pre_s := s $ init 0
       | up := (s > pre_s) or (s = pre_s) and pre_up
       | pre_up := up$ init true
       |)
     | summit := when(pre_up and (not up))
     | local_max := pre_s when summit
     |)
    where
     event summit;
     boolean up, pre_up;
     integer pre_s;
  end;
  ```

5.6

Let `s1` and `s2` be two signals of any type and independent of each other. Define a signal `present` of `event` type that occurs whenever `s1` and `s2` are present at the same time. Does the expression `^s1 and ^s2` adequately define such a signal?
▷

```
present := ^s1 when (^s2)
```

The alternative expression `^s1 and ^s2` constrains `s1` and `s2` to have the same clock.
```

5.7

Given an input signal s, define a process that extracts the values carried by s every N steps as follows: $s_0, s_{N-1}, s_{2N-1}, \ldots$ (the modulo operator could be used).

▷

```
(| cnt ^= s
 | cnt := ((cnt$ init 0) modulo N) + 1
 | extracted_values := s when(cnt = 1)
 |)
```

5.8

Define an event only_one that provides notification that either signal s1 or signal s2 is present and that both signals are not present at the same time.

▷

```
(| together := ^s1 when ^s2
 | only_one := when((not together)
                    default (^s1 default ^s2))
 |)
```

5.9

As an addition, integer signals s1 and s2 must be synchronous. Now, suppose they can arrive at "slightly" different logical instants; we want nevertheless to add them! Define a process Addition that describes their addition.

s1	:	·	·	2	·	4	9	·	…
s2	:	5	7	·	·	·	6	3	…
sum	:	·	·	7	·	11	15	·	…

▷

```
process Addition =
 ( ? integer s1, s2;
   ! integer sum;
 )
(| sum := (ss1 + ss2) when sec
 | ss1 := s1 cell ^s2
 | ss2 := s2 cell ^s1
 | sec ^= s1 ^+ s2
 | sec := (^s1 when ^s2) default not (sec $ init true)
 |)
where
  integer ss1, ss2;
  boolean sec;
end;
```

5.10

Define a vector v3 denoting the product of two vectors v1 and v2 of the same length.

▷

```
array i to size - 1 of
  v3[i] := v1[i] * v2[i]
end
```

5.11

Define an integer unit matrix u.

▷

```
array i to size - 1 of
  array j to size - 1 of
     u[i,j] := if i=j then 1 else 0
  end
end
```

Exercises in Chapter 6

6.1

Let us consider the Counting_Instants process defined in Chap. 5(Sect. 5.1.3).

1. Generate the corresponding C code
2. Simulate the trace scenario illustrated in the example

▷

1. Generation of the corresponding C code,

```
signal -c Counting_Instants.SIG
```

2. Simulation of the trace scenario illustrated in the example.
   a. Generate a Makefile:

```
genMake C Counting_Instants
```

   b. Generate an executable file:

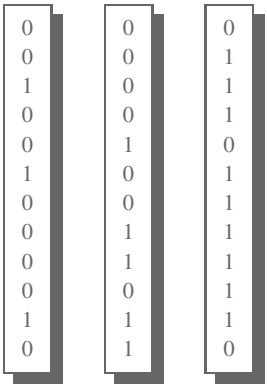
```
make -f Makefile_Counting_Instants
```

   c. Define input data files:[2]

RC_start.dat, RC_finish.dat, and RC_clk.dat,

[2] Note that when an input parameter is of event type, the corresponding data file is denoted as RC_<parameterIdentifier>.dat. The "C_" refers to "clock," which means that RC_<parameterIdentifier>.dat also represents the clock of the signal.

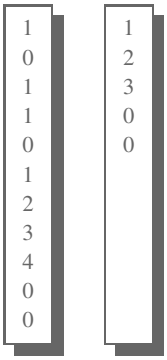
They are represented, respectively, from left to the right, as follows:



d. Execute the generated executable file, and check the generated output data files:

`Wcnt.dat` and `Wduration.dat`

respectively from the left to the right, as follows:



6.2

Define a signal `present` that indicates the number of items stored in a box. The insertion and removal of items are, respectively, denoted by event signals `in` and `out` (both signals may occur at the same instant).

▷

```
(| present ^= in default out
 | pre_present := present$ init 0
 | present := pre_present when in when out default
              (pre_present + 1) when in default
              (pre_present - 1) when out
 |)
```

6.3

Let us consider an image processing context. The pixels obtained from a sequence of pictures are represented as Boolean data, received line by line, and for each line, column by column. A picture has NL lines and NC columns. NL and NC are constant parameters. The last pixel of a picture is followed by the first pixel of the next picture.

Define for each pixel the corresponding line and column number. For instance, the first and second pixels are, respectively, associated with the following couples of rank values: $(0, 0)$ and $(0, 1)$.

▷

```
process Picture
 { integer NL, NC; }
 ( ? boolean pixel;
   ! integer line, column;
 )
(| pixel ^=column
 | column := ((column$ init 0) modulo NC) + 1
 | line := (((line$ init 0) modulo NL + 1)
            when (column = 1)) cell ^pixel
 |)
```

6.4

A two-track railway crossing is protected by a barrier. Each track is closed or opened depending on the value indicated in a Boolean signal $\texttt{close}_i$ ($i \in \{1, 2\}$): *true* means to close and *false* means to open.

Define the barrier's controller as a Boolean signal `close` that indicates *true* when both tracks are closed and *false* when they are opened, as illustrated in the trace given below:

close1 :	t	·	·	t	f	t	f …
close2 :	t	·	f	f	f	t	· …
close :	t	·	·	·	f	t	· …

▷

```
process Barrier =
 ( ? boolean close1, close2;
   ! boolean close;
 )
(| state := (close1 cell (^close2) init false) or
            (close2 cell (^close1) init false)
 | close := state when(state /= (state$ init false))
 |)
where
  boolean state;
end
```

6.5

Resynchronizing, let s be a signal, and clk be a clock faster than that of s. The resynchronization of s on clk consists in delaying its occurrences (without repetition) at the next occurrences of clk whenever clk is absent. When s occurs simultaneously with clk, this resynchronization operation is not necessary. An illustrative trace is as follows:

s : 1 · · 2 · · 3 . . .
clk : · · *t* · *t* · *t* . . .
s on clk : · · 1 · 2 · 3 . . .

Define such a behavior.

▷

```
process Resync =
  ( ? integer s; event clk;
    ! integer s_on_clk;
  )
 (| s_on_clk := (s cell first_clk) when clk
  | first_clk := on_s $ init false
  | on_s := (not clk) default ^s
  |)
 where
   boolean first_clk, on_s;
 end;
```

6.6

Given a matrix *M*, define the vector D that extracts the diagonal elements of *M*.

▷

```
array i to size - 1 of
   D[i] := M[i,i]
end
```

Exercises in Chapter 7

7.1

Propose both operational and denotational interpretations for the semantics of the following statements:

- Clock extraction (| clk := ^s |)
- Clock extraction from a Boolean signal: (| clk := when c |)
- Clock intersection (| clk := s_1 ^* s_2 |)
- Clock union: (| clk := s_1 ^+ s_2 |)

▷

- Clock extraction.
  - Denotational semantics: $[\![(\!|\ \texttt{clk := \^{}s}\ |\!)]\!] =$
    $\{b \in \mathcal{B}_{|clk,s} \mid tags(b(clk)) = tags(b(s)) = C \in \mathcal{C}$
    $\forall t \in C, b(clk)(t) = tt\}$
  - Operational semantics:

$$p \xrightarrow{\text{s}(\bot),\ \text{clk}(\bot)} p,$$
$$p \xrightarrow{\text{s}(v),\ \text{clk}(true)} p.$$

- Clock extraction from a Boolean signal.
  - Denotational semantics: $[\![(\!|\ \texttt{clk := when c}\ |\!)]\!] =$
    $\{b \in \mathcal{B}_{|clk,c} \mid tags(b(clk)) = \{t \in tags(b(c)) | b(c)(t) = tt\} = C \in \mathcal{C}$
    $\forall t \in C, b(clk)(t) = tt\}$
  - Operational semantics:

$$p \xrightarrow{\text{c}(\bot),\ \text{clk}(\bot)} p,$$
$$p \xrightarrow{\text{c}(false),\ \text{clk}(\bot)} p,$$
$$p \xrightarrow{\text{c}(true),\ \text{clk}(true)} p.$$

- Clock intersection.
  - Denotational semantics: $[\![(\!|\ \texttt{clk := s1 \^{}* s2}\ |\!)]\!] =$
    $\{b \in \mathcal{B}_{|clk,s_1,s_2} \mid tags(b(clk)) = tags(b(s_1)) \cap tags(b(s_2)) = C \in \mathcal{C}$
    $\forall t \in C, b(clk)(t) = tt\}$
  - Operational semantics:

$$p \xrightarrow{\text{s}_1(\bot),\ \text{s}_2(\bot),\ \text{clk}(\bot)} p,$$
$$p \xrightarrow{\text{s}_1(\bot),\ \text{s}_2(v_2),\ \text{clk}(\bot)} p,$$
$$p \xrightarrow{\text{s}_1(v_1),\ \text{s}_2(\bot),\ \text{clk}(\bot)} p,$$
$$p \xrightarrow{\text{s}_1(v_1),\ \text{s}_2(v_2),\ \text{clk}(true)} p.$$

- Clock union.
  - Denotational semantics: $[\![(\!|\ \texttt{clk := s1 \^{}+ s2}\ |\!)]\!] =$
    $\{b \in \mathcal{B}_{|clk,s_1,s_2} \mid tags(b(clk)) = tags(b(s_1)) \cup tags(b(s_2)) = C \in \mathcal{C}$
    $\forall t \in C, b(clk)(t) = tt\}$
  - Operational semantics:

$$p \xrightarrow{\text{s}_1(\bot),\ \text{s}_2(\bot),\ \text{clk}(\bot)} p,$$
$$p \xrightarrow{\text{s}_1(\bot),\ \text{s}_2(v_2),\ \text{clk}(true)} p,$$
$$p \xrightarrow{\text{s}_1(v_1),\ \text{s}_2(\bot),\ \text{clk}(true)} p,$$
$$p \xrightarrow{\text{s}_1(v_1),\ \text{s}_2(v_2),\ \text{clk}(true)} p.$$

7.2

Give an operational interpretation for the semantics of the following processes:

- Process P1

```
1: process P1 =
2:   { integer N }
3:   ( ? integer s1;
4:     ! boolean cond; integer s2)
5:   (| s2 := N * s1
6:    | cond := s1 > s2
7:    |);%process P1%
```

- Process P2

```
1: process P2 =
2:    ( ? boolean b1; integer s1;
3:      ! boolean b4;)
4:    (| b2 := when b1
5:     | s2 := s1 $ 1 init 0
6:     | b3 := s2 < 5
7:     | b4 := b2 default b3
8:     |)
9:  where
10:    event b2; boolean b3; integer s2;
11: end;%process P2%
```

▷

- Process P1.
  - Equation at line 5:

$$\texttt{s2 := N * s1} \xrightarrow{\text{s1}(\perp),\ \text{N}(n),\ \text{s2}(\perp)} \texttt{s2 := N * s1},$$
$$\texttt{s2 := N * s1} \xrightarrow{\text{s1}(v_1),\ \text{N}(n),\ \text{s2}(n\times v_1)} \texttt{s2 := N * s1}.$$

  - Equation at line 6:

$$\texttt{cond := s1 > s2} \xrightarrow{\text{s1}(\perp),\ \text{s2}(\perp),\ \text{cond}(\perp)} \texttt{cond := s1 > s2},$$
$$\texttt{cond := s1 > s2} \xrightarrow{\text{s1}(v_1),\ \text{s2}(v_2),\ \text{cond}(v_1>v_2)} \texttt{cond := s1 > s2}.$$

  Then, by substitution, the following is deduced:

$$\texttt{P1} \xrightarrow{\text{s1}(\perp),\ \text{N}(n),\ \text{s2}(\perp),\ \text{cond}(\perp),} \texttt{P1},$$
$$\texttt{P1} \xrightarrow{\text{s1}(v_1),\ \text{N}(n),\ \text{s2}(n\times v_1),\ \text{cond}(v_1>n\times v_1),} \texttt{P1}.$$

- Process P2.
  - Equation at line 4:

$$\texttt{b2 := when b1} \xrightarrow{\text{b1}(\perp),\ \text{b2}(\perp)} \texttt{b2 := when b1},$$
$$\texttt{b2 := when b1} \xrightarrow{\text{b1}(\mathit{false}),\ \text{b2}(\perp)} \texttt{b2 := when b1},$$
$$\texttt{b2 := when b1} \xrightarrow{\text{b1}(\mathit{true}),\ \text{b2}(\mathit{true})} \texttt{b2 := when b1}.$$

– Equation at line 5:

$$\texttt{s2 := s1 \$ 1 init 0} \xrightarrow{\texttt{s1}(\bot),\texttt{s2}(\bot)} \texttt{s2 := s1 \$ 1 init 0},$$
$$\texttt{s2 := s1 \$ 1 init 0} \xrightarrow{\texttt{s1}(\texttt{v}_1),\texttt{s2}(0)} \texttt{s2 := s1 \$ 1 init } \texttt{v}_1.$$

More generally, we have

$$\texttt{s2 := s1 \$ 1 init c} \xrightarrow{\texttt{s1}(\texttt{v}_1),\texttt{s2}(c)} \texttt{s2 := s1 \$ 1 init } \texttt{v}_1,$$

and for short the above rule is rewritten as follows:

$$\texttt{P2}\{c\} \xrightarrow{\texttt{s1}(\texttt{v}_1),\texttt{s2}(c)} \texttt{P2}\{\texttt{v}_1\}.$$

– Equation at line 6:

$$\texttt{b3 := s2 < 5} \xrightarrow{\texttt{s2}(\bot),\ \texttt{b3}(\bot)} \texttt{b3 := s2 < 5},$$
$$\texttt{b3 := s2 < 5} \xrightarrow{\texttt{s2}(v_2),\ \texttt{b3}(v_2<5)} \texttt{b3 := s2 < 5}.$$

– Equation at line 7:

$$\texttt{b4 := b2 default b3} \xrightarrow{\texttt{b2}(\bot),\texttt{b3}(\bot),\texttt{b4}(\bot)} \texttt{b4 := b2 default b3},$$
$$\texttt{b4 := b2 default b3} \xrightarrow{\texttt{b2}(vb_2),\texttt{b3}(\bot),\texttt{b4}(vb_2)} \texttt{b4 := b2 default b3},$$
$$\texttt{b4 := b2 default b3} \xrightarrow{\texttt{b2}(vb_2),\texttt{b3}(vb_3),\texttt{b4}(vb_2)} \texttt{b4 := b2 default b3},$$
$$\texttt{b4 := b2 default b3} \xrightarrow{\texttt{b2}(\bot),\texttt{b3}(vb_3),\texttt{b4}(vb_3)} \texttt{b4 := b2 default b3}.$$

Then, by combining the different transition scenarios for each equation, one obtains globally the following transition rules:

$$\texttt{P2}\{0\} \xrightarrow{\texttt{s1}(\bot),\ \texttt{s2}(\bot),\ \texttt{b1}(\bot),\ \texttt{b2}(\bot),\ \texttt{b3}(\bot),\ \texttt{b4}(\bot)} \texttt{P2}\{0\},$$
$$\texttt{P2}\{0\} \xrightarrow{\texttt{s1}(\bot),\ \texttt{s2}(\bot),\ \texttt{b1}(\mathit{false}),\ \texttt{b2}(\bot),\ \texttt{b3}(\bot),\ \texttt{b4}(\bot)} \texttt{P2}\{0\},$$
$$\texttt{P2}\{0\} \xrightarrow{\texttt{s1}(v_1),\ \texttt{s2}(0),\ \texttt{b1}(\bot),\ \texttt{b2}(\bot),\ \texttt{b3}(0<5 \equiv \mathit{true}),\ \texttt{b4}(\mathit{true})} \texttt{P2}\{v_1\},$$
$$\texttt{P2}\{0\} \xrightarrow{\texttt{s1}(\bot),\ \texttt{s2}(\bot),\ \texttt{b1}(\mathit{false}),\ \texttt{b2}(\bot),\ \texttt{b3}(0<5 \equiv \mathit{true}),\ \texttt{b4}(\mathit{true})} \texttt{P2}\{v_1\},$$
$$\texttt{P2}\{c\} \xrightarrow{\texttt{s1}(v_1),\ \texttt{s2}(c),\ \texttt{b1}(\bot),\ \texttt{b2}(\bot),\ \texttt{b3}(c<5),\ \texttt{b4}(c<5)} \texttt{P2}\{v_1\},$$
$$\texttt{P2}\{c\} \xrightarrow{\texttt{s1}(v_1),\ \texttt{s2}(c),\ \texttt{b1}(\mathit{false}),\ \texttt{b2}(\bot),\ \texttt{b3}(c<5),\ \texttt{b4}(c<5)} \texttt{P2}\{v_1\},$$
$$\texttt{P2}\{0\} \xrightarrow{\texttt{s1}(\bot),\ \texttt{s2}(\bot),\ \texttt{b1}(\mathit{true}),\ \texttt{b2}(\mathit{true}),\ \texttt{b3}(\bot),\ \texttt{b4}(\mathit{true})} \texttt{P2}\{0\},$$
$$\texttt{P2}\{0\} \xrightarrow{\texttt{s1}(v_1),\ \texttt{s2}(0),\ \texttt{b1}(\mathit{true}),\ \texttt{b2}(\mathit{true}),\ \texttt{b3}(0<5 \equiv \mathit{true}),\ \texttt{b4}(\mathit{true})} \texttt{P2}\{v_1\},$$
$$\texttt{P2}\{c\} \xrightarrow{\texttt{s1}(v_1),\ \texttt{s2}(c),\ \texttt{b1}(\mathit{true}),\ \texttt{b2}(\mathit{true}),\ \texttt{b3}(c<5),\ \texttt{b4}(\mathit{true})} \texttt{P2}\{v_1\}.$$

Exercises in Chapter 8

8.1

Give the encoding in $\mathcal{F}_3$ of the statement: `clk := ^s`.

▷

$clk = s^2$

8.2

Let $\mathtt{b}_1$ and $\mathtt{b}_2$ be two Boolean signals. Propose a $\mathbb{Z}/3\mathbb{Z}$ encoding of the statement: $\mathtt{b} := \mathtt{b}_1 \neq \mathtt{b}_2$.

▷

- Value part: $b = -b_1 b_2$
- Clock part: $b^2 = b_1^2 = b_2^2$

Exercises in Chapter 9

9.1

Give the analysis result of the following processes. If there are any problems, suggest a solution to solve them:

1.
```
process S_OccurrenceNumber =
  ( ? integer s;
    ! integer cnt; )
  (| cnt := (cnt $ 1 init 0) + 1 when ^s |);
```
2.
```
process Bi_ValuedState =
  (? event up, down;
   ! integer s)
  (| s1 := up when (pre_s /= 1)
   | s2 := down when (pre_s /= 2)
   | s := (1 when s1) default (2 when s2)
   | pre_s := s$1 init 1
   |)
  where
    integer pre_s;
    event s1, s2;
  end;
```

▷

1. In the process `S_OccurrenceNumber`, there is a clock constraint because the specified equation leads to a recursive definition of the clock of `cnt`. Such a constraint is verified only if the clock of `cnt` is either a subset of that of `s` or is equal to that of `s`. But this information cannot be inferred by the compiler if it is not explicitly specified. To solve this clock constraint, a simple way consists in synchronizing `s` and `cnt`.
2. In the process `Bi_ValuedState`, the clock of `s` is not properly defined: the definition of signals `s1` and `s2` requires that `pre_s` (hence `s`) is present, whereas the definition of `s` requires that either `s1` or `s2` is present! To solve such a clock constraint issue, one can specify a more frequent signal that will serve as a reference clock. For that purpose, the following equations can therefore be considered in process `Bi_ValuedState`:

```
...
| s ^= up ^+ down
| s := (1 when s1) default (2 when s2) default pre_s
| pre_s := s$1 init 1
...
```

Exercises in Chapter 10

10.1

Let A be an integer signal whose value is between 0 and 99. Define a process that produces for each value of A two integer digits, one representing the *tens*, and the other one representing the *units*. As an example, for values of A of 35; 7; 10, one expects 3; 5, 0; 7, 1; 0.

▷

```
process Digits =
  ( ? integer s;
    ! integer digit;
  )
(| tens := not (tens$ init false)
 | s ^= when tens
 | digits := (s/10)
               default ((s cell (not tens)) modulo 10)
 |)
where
  boolean tens;
end;
```

10.2

Let `n` be a positive or null integer signal. Define a process `N_Events` that generates a sequence of `n` events after each occurrence of `n`, and before its next occurrence. A decreasing counter can be used for the definition of this process.

▷

```
process N_Events =
  ( ? integer n;
    ! event i;
  )
(| cnt := n default (pre_cnt - 1)
 | pre_cnt := cnt$ init 0
 | n ^= when (pre_cnt = 0)
 | i := when (pre_cnt > 0)
 |)
where
  integer cnt, pre_cnt;
end;
```

Exercises in Chapter 11

11.1

Let us consider the following processes. Discuss the determinism and the endochrony of each of them.

```
1.      (| s2:= s1-2
         | s3:= s2 $ 1 init 0
         |)
2.      (| s2:= s1*3
         | s3:= s2-4 when s2 >=0
         | s4:= s3 when s3>0
         |)
3.      (| s2:= s1 $ 1 init 0
         | s1:= s3 default s2-1
         | s3 ^= when s2>0
         |)
4.      (| s1:= s2 default s3 |)
5.      (| s2 ^< s1 |)
```

▷

1. Deterministic and endochronous, with s_2^2 as master clock
2. Deterministic and endochronous, with s_2^2 as master clock
3. Deterministic and endochronous, with s_2^2 as master clock
4. Deterministic but not endochronous
5. Neither deterministic nor endochronous

Exercises in Chapter 12

12.1

Define a complete SIGNAL specification of the FSM example presented in Sect. 12.3.1 as an abstraction of a 2-FIFO queue.

▷

```
1: (| s0 := (true when prev_s1 when (^out))
2:           default (false when prev_s0 when(in ^+ out))
3:           default prev_s0
4:  | prev_s0 := s0 $ 1 init true
5:  | s1 := (true when ((prev_s2 when (^out))
6:           default (prev_s0 when (^in))))
7:           default (false when prev_s1 when(in ^+ out))
8:           default prev_s1
9:  | prev_s1 := s1 $ 1 init false
10: | s2 := (true when prev_s1 when (^in))
11:        default (false when prev_s2 when(in ^+ out))
12:        default prev_s2
13: | prev_s2 := s2 $ 1 init false
14: | ERR_empty := (true when prev_s0 when (^out))
15:               default prev_ERR_empty
```

```
16: | prev_ERR_empty := ERR_empty $ 1 init false
17: | ERR_full := (true when prev_s2 when (^in))
18:              default prev_ERR_full
19: | prev_ERR_full := ERR_full $ 1 init false
20: | s0 ^= s1 ^= s2 ^=
21:                ERR_empty ^= ERR_full ^= in ^+ out
22: |)
```

In the same SIGNAL specification, define two Boolean signals `OK_write` and `OK_read` that, respectively, denote the fact that write and read requests are authorized or not.

▷

```
1: (| ...
2:  | OK_write := s0 or s1
3:  | OK_read := s1 or s2
4:  | ...
5:  |)
```

12.2

Let us consider the FWS system model described in Sect. 12.5.2. Is there any way to define the `FWS_System` process such that the communication between the two concurrent processes becomes clock-constraint-free?

▷

One can think of another specification of the `FW_System` model in which the confirmed alarm values `tmp` are transferred via a "clockless" memory, which is made available whenever a process needs to access this memory. This is obtained very easily by introducing a new equation in the body of `FW_System` which defines a local signal `tmp_mem` as follows: `tmp_mem := (var tmp)`.

Then, instead of putting `tmp` as an input for `Alarm_Notifier` at line 7 in Fig. 12.9, we consider `tmp_mem`. The `var` operator allows one to memorize the value of `tmp` in `tmp_mem` (see Sect. 5.2). The clock of `tmp_mem` is the one of the context in which it is used, i.e., the memorized value is available whenever required.

In this second version of the `FW_System` model, there are no longer clock constraints on the specific instants at which the concurrent processes must exchange information. Their associated master clocks are completely independent of each other, with a possible empty set of common instants. In this case, the compiler generates no clock constraints and automatically produces a code without an exception message.

Exercises in Chapter 13

13.1

Compile and simulate the whole `Dining_philosophers` process with POLYCHRONY.

▷

Execute the following commands:

1. Compile and generate, e.g., the corresponding C code:

```
signal -c <fileName>.SIG -par=<fileName>.PAR
```

2. Generate a Makefile:

```
genMake C <fileName>
```

3. Generate an executable file:

```
make -f Makefile_<fileName>
```

4. Define input data files:

```
R<inputName>.dat
```

5. Execute the executable file generated, and analyze the output data files generated identified as follows:

```
W<outputName>.dat
```

Index

$\mathcal{F}_3$ encoding space, 110
  polynomial dynamical equations, 114
  primitive constructs, 110
  some extended constructs, 112

abstract clock, 24, 26–28, 33, 36, 38, 51
  absence (of events), 52
  definition, 54
  empty clock, 75
  operator
    extraction, 74
    intersection, 75
    lower bound, 75
    relative complement, 75
    synchronization, 74
    union, 75
    upper bound, 75
  relation
    equality, 77
    exclusion, 77
    inferiority, 77
    superiority, 77
  role, 53
  type, 54
    event, 54
abstraction, 150
  black box, 152
  external process, 150
  gray box, 153
assignment, 55

behavioral simulation
  input data file, 89, 201
  output data file, 89, 202
  philosophers model, 201
  watchdog, 88

Chronos (mythology), 52
clock calculus
  LUCID SYNCHRONE, 36
  SIGNAL, 123, 144
    clock hierarchy, 123, 130
    example, 124–126, 129, 139
clock hierarchy, 130
  downsampling, 130
  free condition, 130
clock-driven execution, 26
code generation, 141
  philosophers model, 197
  watchdog example, 86
combinatorial process, 98
comments (syntax), 66
compilation, 122
  dining philosophers example, 195
  principles, 122
  watchdog example, 86
conditional dependency graph, 116
  primitive constructs, 116

design patterns, 171
  bottom-up approach (incremental), 176
  endo-isochrony, 185
  finite state machines, 180
  oversampling, 183
  preemption, 182
  top-down approach (refinement), 171
distribution methodology, 167

embedded systems, 4
  complexity, 11
  determinism, 10
  distributed, 10
  heterogeneous, 10
  large scale, 10
  modularity, 11

predictability, 10
safety criticality, 9
endochrony, 131, 144
Euclid's greatest common divisor algorithm, 183
event-driven execution, 26
example
black box specification, 152
external process, 150
gray box specification, 154
model of a tank, 156
examples
ABRO, 29
application executive library, 150
assertion on Booleans, 157
assertion on clock constraints, 157
blackboard (read), 173
checking event presence, 76
contradiction in process specification, 124
counting event occurrences, 75
cyclic definitions in a process, 126
dependency cycle, 117
dining philosophers, 191
endochronization of a process, 136
endochronous program, 132, 135
endoisochronous processes, 166, 185
Euclid's greatest common divisor algorithm, 184
FIFO queue, 177
finite state machine, 181
generated code, 142
hierarchical statechart, 31
operational semantics of a process, 100
overheating prevention, 6
process with constrained inputs, 124
resettable counter, 34, 67, 138
satisfaction of a specific property, 129
static parameters, 67
system-on-chip, 4
temporal determinism, 128
the cardiologist metaphor, 18
the Stop_Self service, 69
timely control of a physical process, 8
vector reduction, 79
watchdog, 8, 84
exochrony, 131, 144

globally asynchronous, locally synchronous systems, 53, 159

modularity, 149
module, 149
library, 150
syntax, 149
monoclocked system, 24, 38
multiclocked semantic model, 102
asynchronous composition, 165
behavior, 103
determinism, 163
endo-isochrony, 165
endochrony, 162
flow equivalence, 162
flow invariance, 165
observation point, 102
relaxation, 162
stretch closure, 105
stretch equivalence, 104
stretching, 104
tag, 102
multiclocked system, 25, 38

oversampling, 155, 184

physical clock, 8, 10, 24, 25
polychrony, 52, 102
process, 64
array of processes, 79
frame, 66
function, 64
intrinsic process, 156
assertion, 156
label, 69
local declaration, 65
notation, 66
body, 66
interface, 66
operators
composition, 65
local declaration, 65
program, 64
processes
hierarchy, 69
sub-processes, 69

reactive systems, 6
real-time programming models, 12
asynchronous, 13
preestimated time, 14
synchronous, 15

real-time systems, 7
  hard real-time systems, 9
  soft real-time systems, 9

semantics
  denotational, 102
  operational, 95
separation of concerns, 17
  functional properties, 17
  nonfunctional properties, 17
signal, 43
  abstract clock, 51
  constant, 51
  declaration, 50
  extended operator
    memorization, 78
    sliding window, 78
  monoclock operators
    delay, 57
    relations/functions, 56
  multiclock operators
    merging, 59
    undersampling, 59
  types, 44
    array, 47
    boolean, 46
    bundle, 47
    character, 46
    comparison, 48, 50
    complex, 45
    conversion, 48, 50
    definition of, 44
    enumerated, 46
    integer, 44
    real, 45
    string, 46
    structure, 47
specification, 150, 152, 153, 155, 173–175
synchronous languages
  ARGOS, 32
  ESTEREL, 27
  LUCID SYNCHRONE, 35
    clock calculus, 36
  LUSTRE, 33
  SIGNAL, 36
  STATECHARTS, 30
  SYNCCHARTS, 32
  goal, 21
synchronous technology
  POLYCHRONY, 83
  SIGALI, 84
  Esterel Studio, 30
  SCADE, 35
synchrony hypothesis, 16, 22, 38