HAL
open science

high-level component-based models for functional verificationof systems-on-a-chip

Yuliia Romenska

To cite this version:

Yuliia Romenska. high-level component-based models for functional verificationof systems-on-a-chip. Computer science. Université Grenoble Alpes, 2017. English. NNT : 2017GREAM019 . tel-01681425v2

HAL Id: tel-01681425

https://theses.hal.science/tel-01681425v2

Submitted on 12 Jan 2018

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Communauté UNIVERSITÉ Grenoble Alpes

THÈSE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ GRENOBLE ALPES

Spécialité : **Mathématiques et Informatique**

Arrêté ministériel : 25 mai 2016

Présentée par

Yuliia ROMENSKA

Thèse dirigée par **Florence MARANINCHI**, Professeur, Grenoble INP

préparée au sein du **Laboratoire Verimag**
dans **l'École Doctorale Mathématiques, Sciences et Technologies de l'Information, Informatique**

Composants Abstraits pour la Vérification Fonctionnelle des Systèmes sur Puce

Thèse soutenue publiquement le **10 mai 2017**,
devant le jury composé de :

Laurence PIERRE
Professeur, UNIVERSITÉ GRENOBLE ALPES, Président
Erika ABRAHAM
Professeur, UNIVERSITÉ D'AIX-LA-CHAPELLE, Rapporteur
Franco FUMMI
Professeur, UNIVERSITÉ DE VÉRONE, Rapporteur
Kim GRÜTTNER
Chercheur principal à OFFIS, UNIVERSITÉ CARL VON OSSIETZKY D'OLDENBOURG, Examinateur
Laurent MAILLET-CONTOZ
Directeur de recherche, STMICROELECTRONICS, Examinateur
Florence MARANINCHI
Professeur, GRENOBLE INP, UNIVERSITÉ GRENOBLE ALPES, Directrice de thèse

This document is licensed under a Creative Commons Attribution 4.0 International (CC BY 4.0) license.

Yuliia Romenska: *High-Level Component-Based Models for Functional Verification of Systems-on-a-Chip,*
© 2017

To my teachers

Acknowledgements

The work presented in this thesis would not be possible without the help of many people. First of all, I would like to thank my supervisor *Florence Maraninchi* for all our fruitful discussions, for the knowledge she gave me, for teaching me to be a researcher, for support and her patience, for faith in me and my work, for the freedom she gave in developing my ideas, for availability and readiness to help despite the busy schedule.

I would like to thank the members of my jury who paid attention to this work. Thanks to *Erika Abraham* (RWTH-Aachen, Germany) and *Franco Fummi* (University of Verona, Italy) for the time they spent reviewing the manuscript; to *Laurent Maillet-Contoz* (STMicroelectronics, France) and *Kim Grüttner* (OFFIS, Germany) for their role as examiners; to *Laurence Pierre* (TIMA, France), for accepting to be a president of the jury. Their questions and insightful comments contributed the quality of this work. I would like to thank *Laurent* also for the evaluation of this thesis from the industrial perspective, and for the management of the OpenES project which made this thesis possible.

I have been lucky in doing a Ph.D. at the Verimag laboratory. I thank all my colleagues for the creative environment they make, for our seminars and events. They always inspired me to discover and to learn things beyond my field, to improve and to move forward in my scientific inquiries.

Thanks to the members of the *Synchrone* team for the good atmosphere, for their feedback on some parts of the presented work, for their readiness to help and support. In particular, I would like to thank: *Matthieu Moy* for showing me the "SystemC world"; *Nicolas Halbwachs* and *Pascal Raymond* for their feedback on my attempts to encode order non-determinism into PSL, *Erwan Jahier* for showing me the Lurette tool. I also thank *Claire Maiza*, *Susanne Graf*, *Karine Altisen*, *Fabienne Carrier*, *Catherine Parent-Vigouroux* and *Stéphane Devismes* for the short and long talks we had on various scientific and non-scientific topics.

I would not be able to complete the PhD without the fundamental knowledge I got at V. N. Karazin Kharkiv National University. I thank *Grygoriy N. Zholtkevych*, *Lilia P. Belova*, *Irina T. Zaretska*, ... for being my professors.

In addition I thank *Frédéric Mallet* who gave me the first taste of what research is.

Many thanks are addressed to my friends: to those I met at Verimag (*Irini*, *Abhinav*, *Mihail*, *Irina*, *Thomas*, *Lijun*, *Wei-Tsun*, *Anais*, *Maxime*, *Denis*, *Moustapha*, *Amaury*, *Souha*, ...) for all nice time we spent together and for the cultural exchange; to *Vera* for all the sweets she made for me and for her positiveness; to *Atefeh* for our trips and small discoveries; to my *Ubinet* friends.

I am endlessly grateful to *Hamza Rihani* who was always by my side during this adventure and shared with me good and less good times. It is hard to overestimate the importance of his support for the success of this PhD. He proofread entirely the manuscripts and helped me to improve the quality of text.

Finally, I would like to thank my family, who always supported and encouraged me in what I undertook.

Contents

I Preliminaries **1**

1 Introduction **3**
1.1 Context . . . 3
1.2 Summary of Contributions . . . 4
1.3 Outline of the Document and Suggestions for Readers . . . 5
1.4 Dissemination Activities . . . 6

2 Background **9**
2.1 Systems-on-Chip . . . 9
2.1.1 Definition . . . 9
2.1.2 Design Flow . . . 11
2.1.3 Functional Verification . . . 12
2.1.4 SystemC . . . 15
2.1.5 Transaction Level Modeling . . . 21
2.2 Formal Models for Discrete Concurrent Systems . . . 24
2.2.1 Synchronous Models . . . 24
2.2.2 Asynchronous Models . . . 28
2.3 Formal Specifications . . . 29
2.3.1 Intuition . . . 29
2.3.2 Defining Properties (The Discrete-Time Case) . . . 30
2.3.3 Specification Languages for Hardware Designs . . . 33
2.4 Recognition of Regular Languages, Continuous Recognizers . . . 34
2.5 Components and Contracts . . . 36

II Running Example **37**

3 The Running Example **39**
3.1 The External View and Functionality of the System . . . 39
3.2 SystemC/TLM Virtual Prototype . . . 41
3.2.1 TL Models of the Components . . . 41
3.2.2 SystemC/TLM-2.0 Implementation of the Components . . . 45
3.2.3 Communication Mechanism . . . 47
3.3 The Embedded Software and the CPU . . . 50
3.3.1 The Control Algorithm . . . 50
3.3.2 Interrupt Handling . . . 50
3.3.3 Execution Modes . . . 52
3.4 Formal Specification of Components . . . 53
3.4.1 The Embedded Software and the CPU . . . 54
3.4.2 The Image Processing Unit (IPU) . . . 57
3.4.3 The Liquid Crystal Display Controller (LCDC) . . . 58
3.4.4 The Image Sensor (SEN) . . . 59
3.4.5 Other Components . . . 59
3.5 Synchronization Bugs . . . 60

III Efficient Monitoring of Loose-Ordering Properties for SystemC/TLM 63

4 Introducing the Notion of Loose-Ordering 65
4.1 Introduction . . . 65
4.2 Motivation for Loose-Orderings . . . 67
4.2.1 Over-Constraining . . . 67
4.2.2 Robustness of the Embedded Software . . . 68
4.3 Proposal . . . 69
4.3.1 Loose-Orderings . . . 69
4.3.2 Loose-Ordering Patterns . . . 69
4.4 Definitions . . . 70
4.4.1 Input/Output Interface . . . 71
4.4.2 Loose-Orderings . . . 71
4.4.3 An Antecedent Requirement . . . 72
4.4.4 A Timed Implication Constraint . . . 74
4.5 Encoding Loose-Ordering Properties . . . 75
4.5.1 The Encoded Subset . . . 75
4.5.2 Encoding Using SERE Operators . . . 76
4.5.3 Encoding Using LTL Operators . . . 78

5 Compositional Building of Recognizers 89
5.1 Introduction . . . 89
5.2 Recognition Principle . . . 90
5.3 Recognition Context . . . 91
5.4 Building Recognizers . . . 92
5.4.1 Elementary Recognizers of Ranges . . . 93
5.4.2 Composite Recognizers . . . 99
5.4.3 Recognizers of Loose-Ordering Properties . . . 102
5.5 Validation . . . 103
5.5.1 Implementing Recognizers in Lustre . . . 103
5.5.2 Obtaining Monitors from the Lustre Implementation . . . 104

6 Monitoring Principles and SystemC Implementation 107
6.1 Introduction . . . 107
6.1.1 Challenges in Monitoring SystemC/TLM Models . . . 107
6.1.2 Monitoring Loose-Ordering Properties . . . 108
6.2 A Library of Primitive Monitors: SystemC Implementation . . . 110
6.2.1 Attribute Grammar of the Language of Loose-Orderings . . . 110
6.2.2 SystemC Implementation . . . 113
6.3 Monitors of Loose-Ordering Properties . . . 118
6.4 Monitorable SystemC/TLM Channels . . . 119
6.4.1 Observable Signal Channels . . . 119
6.4.2 Observable SystemC/TLM Sockets . . . 120

7 Experiments 125
7.1 Comparing Complexities . . . 125
7.1.1 Experimental settings . . . 125
7.1.2 The ViaPSL Strategy: Complexities of the PSL Monitors . . . 126
7.1.3 The Drct Strategy: Complexities of the Direct SystemC Monitors . . . 126
7.1.4 Comparison . . . 127
7.2 Monitoring Loose-Ordering Properties . . . 128

IV Towards Generalized Stubbing with Contracts 131

8 Towards Generalized Stubbing with Sequence Properties 133
8.1 Introducing the Notion of Generalized Stubbing . . . 133
8.1.1 Inspiration . . . 134
8.1.2 Generalized Stubbing for SoCs . . . 135

8.1.3 Constraints on Property Languages and Semantic Choices . . . 135
8.1.4 Expected Benefits of Generalized Stubbing and Contributions . . . 136
8.2 Runs of Components, Prefixes, Fragments . . . 136
8.3 Expressivity of Contracts vs Implementability of Stubs . . . 137
8.3.1 Semantics of the Implication Operator . . . 138
8.3.2 Efficient Implementation . . . 138
8.3.3 Productive Runs and a Necessary Condition for Bounds . . . 138
8.4 Implementation of an Assume Clause . . . 138
8.5 Restrictions on Property Languages of a Guarantee Clause and Semantic Choices . . . 139
8.5.1 Semantic Choices . . . 139
8.5.2 Continuous Recognition of π . . . 140
8.5.3 Defining Several Guarantee Clauses . . . 140
8.5.4 Bounds of a Cycle-Free Set of Guarantee Clauses . . . 141
8.5.5 Summary on Constraints . . . 142
8.6 Semantics of a System of Components . . . 142

9 Implementation by Encoding into Mealy Machines 145
9.1 Principles . . . 145
9.2 Encoding of a Contract . . . 146
9.2.1 Encoding of an Assume Clause . . . 147
9.2.2 Encoding of a Guarantee Clause . . . 148
9.2.3 Non-Deterministic Choice Between Inputs and Outputs and Its Interpretation . . . 150
9.3 Encoding of the Semantics of a Stub-Based System . . . 151
9.3.1 Definitions . . . 151
9.3.2 Semantics . . . 151
9.4 Choices on Non-Determinism . . . 153

10 Execution Mechanics: Implementation with the SystemC Scheduler 155
10.1 Implementation Principles . . . 155
10.1.1 Enabling Communication . . . 156
10.1.2 Implementing a Stub . . . 156
10.2 Simulation with the SystemC Scheduler . . . 157
10.2.1 Recall: the Semantics of a Stub-Based System . . . 157
10.2.2 A Scheduling Algorithm . . . 157
10.3 Experiments . . . 160
10.3.1 Experimental Settings . . . 160
10.3.2 Observations and Results . . . 160

V Related Work and Conclusions 165

11 Related Work 167
11.1 Functional Verification of SystemC/TLM . . . 167
11.1.1 Specification Languages and TLM assertions . . . 167
11.1.2 Monitoring SystemC/TLM . . . 168
11.1.3 Verification Methodologies and Libraries . . . 172
11.2 Functional Verification Through the Design Flow of SoCs . . . 173
11.2.1 Transactor-Based Verification: Properties Reuse . . . 173
11.2.2 Properties Refinement . . . 173
11.3 Contracts for SystemC/TLM . . . 174
11.3.1 Contracts for Hardware Designs . . . 174
11.3.2 Other Uses of Formally-Defined Contracts . . . 174
11.4 Virtual Prototyping of SoCs . . . 174
11.4.1 Early Detection of Bugs . . . 174
11.4.2 Raising Abstraction Level Beyond TLM . . . 175

12 Conclusions and Prospects **177**
12.1 Summary and Contributions . . . 177
12.1.1 Context of the Work . . . 177
12.1.2 Contributions . . . 178
12.2 Prospects . . . 179
12.2.1 Integrating Loose-Orderings into Existing Standards . . . 179
12.2.2 Direct Prospects for the Academic World . . . 180
12.2.3 Long Term Prospects . . . 180

Appendices **180**

Appendix A LUSTRE Implementation of Loose-Orderings **181**

Appendix B Encoding of Loose-Orderings into PSL **187**
B.1 Encoding of an Antecedent Requirement into LTL . . . 187
B.2 Encoding of a Timed Implication Constraint into LTL . . . 199

Bibliography **217**

Part I

Preliminaries

Chapter 1

Introduction

Contents

1.1 Context 3
1.2 Summary of Contributions 4
1.3 Outline of the Document and Suggestions for Readers 5
1.4 Dissemination Activities 6

1.1 Context

Applications of Systems-on-a-Chip (SoCs) include most of the electronic devices used nowadays. A SoC is an electronic system embedded within a larger electronic device, such that hardware components necessary for the system's operation are grouped onto a single integrated circuit. SoCs are extremely complex; they consists of *hardware*, made of blocks, and embedded *software*, meant to be executed on top of the hardware. Functional correctness of such systems depends on the proper interaction of these heterogeneous parts; it is required to check that the software operates correctly on the assumed hardware. Moreover, it is also required to check that the hardware operates in the "expected" way.

Due to the enormous cost and intrinsic complexity of hardware components, *virtual prototyping* is used to ensure the correctness of a SoC. A virtual prototype is a very abstract executable model of the hardware. It has the same interface for the embedded software as the real hardware such that the software can be executed on the virtual prototype. The embedded software implemented by means of virtual prototyping at the later stages of the design flow can be executed on the hardware. The degree of details relevant to the actual hardware, which is presented by virtual prototypes, defines their abstraction level. Virtual prototypes at the lower abstraction levels (e.g., *Register Transfer Level – RTL*) can be used for synthesis of the hardware chip. Their simulation is very slow because the models contain too much details, specially on timing aspects. Simulation with virtual prototypes at higher abstraction levels (e.g., *Transaction Level – TL*) can be very fast; these models abstract away many details. For instance, wires which can be explicitly modeled at RTL are abstracted by chunks of data, called *transactions*, at TLM. Since such virtual prototypes are more abstract, they require less effort to build.

SystemC-based Transaction-Level Modeling (TLM) [Sys] is the industrial standard for defining high-level executable models of SoCs. Due to their simulation speed, TL models are used for the development of the embedded software and checking software/hardware interfaces. Simulation with SystemC/TLM virtual prototypes can help to detect functional bugs in a system's design early, (potentially) saving time are effort that are needed for their fixing. Therefore, it is very important to check that:

- TL models are correct and do not have functional bugs,
- they are representative, or faithful, to the real hardware.

For SystemC/TLM virtual prototypes *Assertion-Based Verification (ABV)* allows property checking early in the design cycle [Eck+06b; Tom+09; PF08]. Properties (or assertions) define aspects of the expected behavior of TL models (e.g., a communication protocol of the components). There are dedicated languages that can be used to define the assertions (e.g., PSL [18505; EF06]). ABV defines how properties are interpreted and evaluated when SystemC/TLM virtual prototypes are simulated. Although ABV helps to find malfunctions in the virtual prototype early, TL models can be *over-constrained*, which means that

they do not represent all behaviors of the hardware. Due to that some malfunctions can be discovered too late in the design flow, which, in the extreme case, can cause re-spin of a chip [Bor]. For instance, in [Cor08], it has been established that exact delays in SystemC/TLM models are sources of over-constraints and spurious synchronizations in models. The problem has been solved by introducing a notion of *loose-timing* which replaces exact delays with intervals. TL models can (potentially) have other over-constraints, which are not identified yet. Those over-constraints should be found and removed from the models. Moreover, ABV support for capturing those over-constraints should be proposed.

Early simulation of SoCs with SystemC/TLM virtual prototypes is fast and allows early development of the embedded software. To simulate the system, the TL model should be *executable*. Executability means that certain details on interactions of components, exchanged data and/or components' internal behavior are defined. Although the effort spent to build TL models is negligible comparing to RTL models, it takes time to define TL models of all hardware components and to implement a TLM virtual prototype. To check the software, many of the details about the components' implementation can be irrelevant (e.g., the actual implementation of computation algorithms); only their border behavior, i.e., inputs/outputs they consume/provide, matters. To enable very early simulation with SystemC/TLM virtual prototypes, when the implementation of some of the components is missing and only a very abstract definition of their input/output behavior is provided, the idea is to use the specification instead of those components. It requires the understanding of what it means to "run" the specification, and what types of the specification can be used in place of TL components.

Both the identification and the removal of over-constraints in TL models, and early simulation of SoCs with SystemC/TLM virtual prototypes are closely related to the problem of the raise of the abstraction level above TLM. Adding non-determinism to the model as a way to remove over-constraints is a natural step towards raising the abstraction level of the models. The specification capturing this non-determinism can be used in early simulation of the system, which in turn allows early detection and fixing of functional bugs.

The work presented in this document was carried out during a Ph.D. at the team *Synchronous Language and Reactive Systems (Synchrone)* of the Verimag laboratory, in the context of the OpenES European CATRENE project, in a collaboration with STMicroelectronics. The focus has been made on a family of SoCs that includes one or more CPUs running embedded software, one or several accelerators (e.g., hardware blocks), potentially complex interconnects, memories, and input/output devices. A part of this thesis is about examining and testing TL models. It attempts to answer the questions: Are there other sources of over-constraints in TL models? Why do they appear and what are the consequences for the quality of the software and the hardware? How to remove those over-constraints? Finally, how to specify and test obtained models? Another part of this work is the investigation of the fundamental question on abstraction level: if one wants to start with models above TLM, what kind of information can be accepted to be not available yet? To answer the questions, it was required to master virtual prototyping of SoCs with SystemC/TLM, to understand what is a "bug" in these models, to check types of bugs which often happen and to analyze their causes. The work also required to become proficient in Assertion-Based Verification for TL models; in particular, to learn different specification styles, specification languages used in the domain, testing approaches for hardware designs, the industrial standards of verification methodologies used in the design flow of SoCs, etc. Apart from that, the work at Verimag with the Synchrone team reinforced and augmented the knowledge about formal models and specifications, which was the inspiration for this thesis.

1.2 Summary of Contributions

The main contributions of this thesis consist of three parts.

- **Design of a small case study of a system-on-a-chip and implementation of its TLM virtual prototype.** The case study is made of components frequently used in industrial case studies. It is characterized by complex synchronization between the components. The virtual prototype is implemented in SystemC/TLM; it is used for experiments.

Two other parts are orthogonal and complementary. First, this work contributes results related to the specification and the monitoring of TL models:

- **Identification of the notion of *loose-ordering*.** This notion is very much in the same spirit as loose-timing, which helps avoiding over-constraints in transaction-level models of systems-on-a-chip due to order of interactions between components.

- **Definition of a set of primitive constructs to capture loose-ordering properties.** The properties are identified by reviewing industrial case studies. These constructs could be integrated in a language like PSL to allow expressing loose-ordering properties.
- **Translation of the primitives into efficient SystemC monitors.** The monitors can be used for on-line testing of SystemC/TLM models. They facilitate detection and localization of bugs caused by erroneous synchronization of components.

The third part of the contributions is about the very early simulation with SystemC/TLM virtual prototypes:

- **Definition of a generalized stubbing mechanics for systems-on-a-chip.** The mechanics enables the early simulation of systems in which some of the components have very abstract specifications only, in the form of constraints between inputs and outputs. A *contract-like* specification style with an *assume clause* that specifies what the component expects from its environment, and a *guarantee clause* that specifies what the component promises to produce if used properly, is adopted. The stubbing mechanics provides a facility to expose bugs, and blame faulty components (if the assume clause of A complains, then the components that provide inputs to A are to be blamed). The mechanics is *generic*; the focus is made on key concepts, principles and rules which make the stubbing mechanics implementable and applicable for real, industrial case studies. Any specification language which conforms the requirements (e.g., loose-orderings), can be plugged in the stubbing framework.
- **Implementation of the generalized stubbing mechanics in SystemC.** The SystemC implementation should be immediately usable by people used to SystemC/TLM. It allows integration of stubs into SystemC/TLM virtual prototypes.

1.3 Outline of the Document and Suggestions for Readers

The thesis is made of five parts and two appendices:

I. Preliminaries

- *Chapter 2 "Background"* is a collection of different notions the reader may or may not know about. This chapter can be skipped; we refer to its sections when needed.

II. Running Example

- *Chapter 3 "Running Example"* introduces the first contribution of this work. The chapter describes a SystemC/TLM virtual prototype used through the document to motivate and illustrate new notions, and to perform experiments. The big part of this chapter describes the implementation details of the model in SystemC/TLM. For those readers, who are either not familiar with or not interested in SystemC/TLM, we suggest to read Sections 3.1 and 3.5; they provide respectively an overview of the system's functionality and a list of bugs the system may have.

The following two parts can be read independently, in any order:

III. Efficient Monitoring of Loose-Ordering Properties for SystemC/TLM

- *Chapter 4 "Introducing the Notion of Loose-Ordering"* motivates and defines a notion of loose-orderings and loose-ordering patterns. Section 4.5 presents encoding of loose-ordering properties into PSL; if the reader is not familiar with PSL, this section can be skipped. The rest of the chapter should be read entirely. When reading this chapter the reader is suggested to consult Section 3.4, where (s)he can find a list of the loose-ordering properties of our running example.
- Chapters 5 and 6 focus on implementation aspects. *Chapter 5 "Compositional Building of Recognizers"* defines recognizers of loose-ordering properties and shows their implementation in Lustre for validation purposes. *Chapter 6 "Monitoring Principles and SystemC Implementation"* presents a monitoring principle of SystemC/TLM models and a SystemC implementation of the recognizers. These chapters are suggested to the readers interested, in particular, in the technical aspects related to the checking of loose-ordering properties.

- *Chapter 7 "Experiments"* compares complexities of our SystemC monitors with monitors of PSL properties. It also demonstrates on the running example the benefits of the SystemC monitors in detecting and finding bugs of SystemC/TLM virtual prototypes.

IV. Towards Generalized Stubbing with Contracts

- *Chapter 8 "Towards Generalized Stubbing with Sequence Properties"* introduces a generalized stubbing mechanics, identifies related semantical, implemental and schedular problems and proposes their solution. This chapter should be read entirely; it provides all the concepts, needed to understand the generalized stubbing mechanics.
- Chapters 9 and 10 describe implementation and the early simulation with stubs. *Chapter 9 "Implementation by Encoding into Mealy Machines"* defines the operational semantics of stubs and a system made of stubs. Then, *Chapter 10 "Execution Mechanics: Implementation with the SystemC Scheduler"* provides a SystemC implementation of the stubbing mechanics and describes a simulation of a stub-based system with the SystemC scheduler. This chapter also shows the experiments.

V. Related Work and Conclusions The document concludes with the following chapters:

- *Chapter 11 "Related Work"* presents an overview of related work.
- *Chapter 12 "Conclusions"* ends this document with a summary of the various contributions of the thesis, as well as the prospects of this work.

There are two appendices:

- *Appendix A "*Lustre *Implementation of Loose-Orderings"* provides a Lustre implementation of the recognizers of the loose-ordering properties.
- *Appendix B "Encoding of Loose-Orderings into PSL"* contains examples of loose-ordering properties and their encoding into PSL.

1.4 Dissemination Activities

The first part of this work has been published as a DATE'16 paper:

- Yuliia Romenska and Florence Maraninchi. "Efficient Monitoring of Loose-Ordering Properties for SystemC TLM". in: *Design, Automation, and Test in Europe (DATE)*. Dresden, Germany, 2016

Parts of the contributions were presented at conferences and workshops:

- Yuliia Romenska. "High-Level Component Based Models for Functional and Non-Functional Properties of Systems-On-Chip". *A talk given at the 21st International Open Workshop on Synchronous Programming (SYNCHRON)*. Aussois, France. December 2, 2014.
- Yuliia Romenska. "Efficient Monitoring of Loose-Ordering Properties for SystemC TLM". *An interactive presentation given at the DATE'16 conference*. Dresden, Germany. March, 2016.
- Yuliia Romenska. "Efficient Monitoring of Loose-Ordering Properties for SystemC TLM". *A talk given at the OpenES and CONTREX Projects Workshop*. Dresden, Germany. March, 2016.

Various aspects of the work were presented during internal seminars of the VERIMAG laboratory:

- Yuliia Romenska. "High-Level Executable Component-Based Models for Functional and Non-Functional Properties of SoCs". *A talk given at the seminar of the Verimag laboratory*. Grenoble, France. February, 2015.
- Yuliia Romenska. "Encoding of Loose-Orderings into Linear Temporal Logics". *A talk given at the meeting of the SYNCHRONE/Verimag team*. Grenoble, France. October, 2015.
- Yuliia Romenska. "Efficient Monitoring of Loose-Ordering Properties for SystemC TLM". *A talk given at the seminar of the SYNCHRONE/Verimag team*. Grenoble, France. November, 2015.

Other dissemination activities are:

- Yuliia Romenska. "Very Early Simulation with Component-Based Virtual Prototypes for Systems-on-a-Chip". *A talk given at the 4th edition of the PhD day organized by the MSTII Doctoral School of the universitity Grenoble-Alpes (UGA).* Grenoble, France. April 30, 2015. *Best Poster Award.*
- Yuliia Romenska. "Contract-Based Specifications for Systems-on-Chip". *A talk given at the 5th Halmstad Summer School on Testing.* Halmstad, Sweden. June, 2015.

Chapter 2

Background

Contents

2.1 Systems-on-Chip **9**
2.1.1 Definition 9
2.1.2 Design Flow 11
2.1.3 Functional Verification 12
2.1.4 SystemC 15
2.1.5 Transaction Level Modeling 21
2.2 Formal Models for Discrete Concurrent Systems **24**
2.2.1 Synchronous Models 24
2.2.2 Asynchronous Models 28
2.3 Formal Specifications **29**
2.3.1 Intuition 29
2.3.2 Defining Properties (The Discrete-Time Case) 30
2.3.3 Specification Languages for Hardware Designs 33
2.4 Recognition of Regular Languages, Continuous Recognizers **34**
2.5 Components and Contracts **36**

To make the document self-contained, in this chapter we provide the minimum background information required for understanding of the first and the second parts of this work. The reader familiar with the presented topics may skip this chapter, or check only its specific sections. When defining our work in the following chapters, we will refer to related sections of the background provided here.

2.1 Systems-on-Chip

2.1.1 Definition

A *System-on-a-Chip (SoC)* refers to the integration of different computing elements and/or other electronic subsystems into a single-integrated circuit (chip) [Ris11]. A SoC is heterogeneous: in addition to classical digital components (e.g., processor, memory, bus, etc.) it may contain analog, mixed-signal, and other radio-frequency functions - all on one chip. These systems range from portable devices such as MP3 players, videogame consoles, digital cameras, or mobile phones to large stationary installations like traffic lights, factory controllers, engine controllers for automobiles, or digital set-top boxes [Pop+10]. It consists of the hardware architecture and the embedded software meant to be executed on its *programmable* part.

The hardware architecture of a SoC (e.g., Fig. 2.1) is defined by the set of hardware blocks (in the sequel also referred to as *components*, *modules*) and subsystems made of them. Most of those blocks are pre-designed, they are referred to as *Intellectual Property* (IP). Components constituting the architecture can be divided into two categories: (i) hardware, and (ii) communication.

2.1.1.1 Hardware Components

Programmable Components The programmable hardware components are also called *processor nodes* of the architecture. They include *computing resources* represented by the processing units or CPUs, intra-subsystem communication, and other hardware components, such as local memories, I/O components,

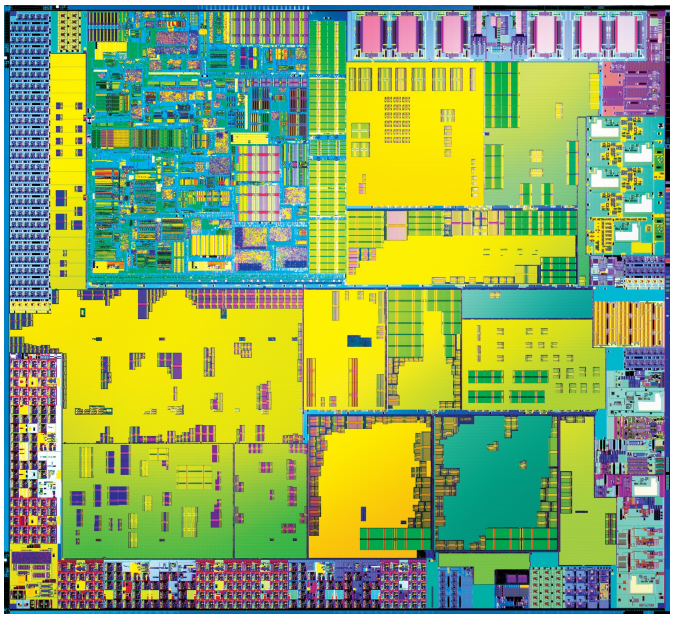


Figure 2.1 Intel EP80579 "Tolapai" SoC [Int].

or hardware accelerators. The *CPU* (Central Processing Unit) also known as processor core, processing element, or shortly processor, executes programs stored in the memory by fetching their instructions, examining them, and then executing them one after another [TA12]. Depending on the number of processors, one may distinguish *single core* and *multi-core* processor nodes of a SoC.

Non-Programmable Components The non-programmable hardware components add functionalities to a SoC. They are peripherals, co-processors (hardware accelerators), memories. A *peripheral* component senses state (status) or get data from devices externally connected to a SoC. When the component detects an event, it can generate an interrupt request, which triggers the execution of a service routine on the processor node. The typical peripherals are General Purpose Input/Output (GPIO), Timers (TMR), Interrupt Controllers (INTC), LCD Controller (LCDC), etc. A *co-processor* is designed and optimized to quickly perform a specific set of computations to accelerate the system performance. Examples of hardware accelerators are cryptoprocessors, DSPs performing a digital signal processing from a data stream (voice, image, etc.). A *memory* is a digital storage device which is commonly used to allocate the program executed by a processor.

Peripherals, hardware accelerators implement a set of *internal registers*; writing or reading those registers allow to control or monitor the components. A basic classification of the registers is:

- *control register*: a write-only register which configures and controls the component (e.g., a register setting parameters in an LCDC);
- *status register*: a read-only register used to get the status of the component (e.g., the register used in the INTC to store the owner of the transmitted interrupt);
- *data register*: a read-write register which sets or gets data (e.g., the result of the computation performed by a hardware accelerator).

2.1.1.2 Inter-Component Communication

Inter-block communication enables transmission of data from a *source* hardware component to a *destination*. It can be implemented by means of a *bus* or a *network on chip* (NoC). A bus is a collection of parallel wires for transmitting address, data and control signals. A standard on-chip bus may connect a CPU and standard components like memories, peripherals, interrupt units, or some application-specific components (e.g., see Fig. 2.2). The most widely-spread on-chip connection standard is the Advanced Microcontroller Bus Architecture, AMBA [Fur00] specified by ARM. To improve both performance and scaling with the rapid increase in the number of hardware components to be connected, buses can be linked through bridges forming complex *hierarchies* (e.g., AMBA-based Advanced High-performance Bus (AHB), Advanced System Bus (ASB), Advanced Peripheral Bus (APB) [Fur00]).

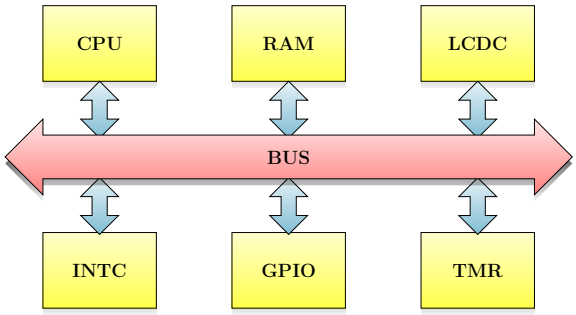


Figure 2.2 A standard block diagram of a SoC architecture based on a system bus.

When bus-based solutions reach their limit, packet-switched networks are used instead [Fur05]. A network-on-chip (NoC) is constructed from point-to-point data links interconnected by switches, which provide means to route the data messages from the source module to the destination module (e.g., Quarc NoC [MVS08], Spidergon NoC [Cop+04], Hermes NoC [Mor+04]).

2.1.2 Design Flow

The complexity of SoCs comes from the need to ensure that the embedded software runs correctly on the hardware. Functional bugs are a large, if not the largest, cause of chip respins [Bor]. To tackle the complexity, ensure functional correctness and increase productivity at the SoC design stage, several strategies are used along the design flow. Among them (i) the abstraction of the hardware, (ii) the early hardware/software co-development, (iii) the specification and verification of the design (these will be discussed in Sections 2.3 and 2.1.3 respectively).

Models of the Hardware: Levels of Abstraction There are common models used in the hardware design written with more or less details. These levels of details are commonly called *abstraction levels*. The traditional hardware design process relies classically on three different levels of abstraction:

- The **Layout** is the most precise description of the chip. The location and design of each transistor is precisely known.
- The **Gate Level** abstracts away a lot of details by focusing only on describing the logical gates (AND, OR, flip-flops, etc.) and their connections. The actual implementation of the gates is provided with the hardware libraries.
- At the **Register Transfer Level** (RTL) the gate-level instantiations of independent flip-flops and logical operators are replaced by registers and a data-flow description of the transfers between registers. Still, each wire is represented, but its precise value is known only at each clock tick.

There are tools which allow automatic translation of the RTL models to the gate level. This process is called *synthesis*. It optimizes the circuit with respect to its surface and timing. The translation of the gate level models to the layout level can be done automatically by the use of place-and-route software optimizing the locations and connections of the various elements.

The RTL models are very slow in simulation. Moreover, they appear too late in the design flow to be used for the hardware/software co-design. To achieve better simulation speeds while benefiting of the early availability of the models, one may create models with less details before the RTL one:

- **Cycle-accurate (CA)** models require that at component bounds the wires are the same as at RTL, and exhibit the same value at each clock cycle. The internals of the component are left free to the implementer.
- **Transaction Level Models (TLM)** are created after hardware/software partitioning, i.e., after it has been decided for each processing if it would be done using a specific hardware block or by software. The main idea of TLM is to abstract away communication on the buses by so-called *transactions*: instead of modeling all the bus wires and their state changes, only the logical operations (reading, writing, etc.) carried out by the buses are considered in the model. The TL models do not use clocks. They are asynchronous by nature, with synchronization occurring during the communication between components. These abstractions allow simulations multiple orders of magnitude faster than RTL. TLM is discussed in more details in Section 2.1.5.

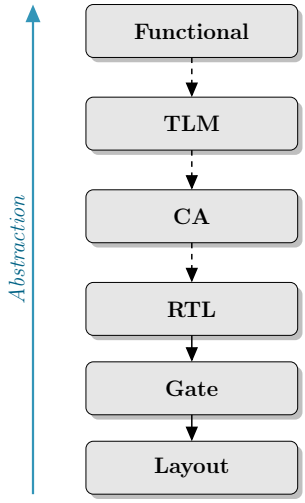


Figure 2.3 Different abstraction levels for describing the hardware. (Source [Cor08])

- **Function** or **algorithmic** models can be sequential C/C++ programs or MATLAB descriptions. These models can serve as a reference implementation of processing algorithms (e.g., face recognition), however they do not contain any details with respect to the final hardware and software.

There is no automatic translation of the cycle-accurate models to RTL, or of the TL models to the cycle-accurate one. One needs to encode respective models manually. This can be tedious and error-prone. To ensure that the design at any level of abstraction is complaint with the specification, functional verification is used along the design flow.

2.1.3 Functional Verification

The reasons for functional errors in a system's design can be (i) ambiguities in the design specification, (ii) implementation errors in the design. Formal specification of the design is defined in Section 2.3. This section provides the (simplified) view on the mechanisms enabling the functional verification of hardware designs.

2.1.3.1 The Design Under Verification (DUV)

The *Design Under Verification*[1] (DUV) can be defined at different abstraction levels (e.g., TLM, RTL). The DUV can present either a module (like an IP block), of the whole system (like a SoC). The verification phase defines the type of properties of the DUV which are checked during the simulation, and the way those properties are checked (see below).

2.1.3.2 Simulation-Based Functional Verification

The primary goal of *functional verification* is to ensure that the initial design implementation, i.e., DUV, is functionally equivalent to the design specification [Vas06]. The functional verification concerns only the intended *behavior* of the design; the non-functional properties such as power consumption or temperature are not considered. Time is classified as a functional property if it affects the correctness of the system's behavior.

One can distinguish two fundamental approaches in verifying the functionality of the mixed software/hardware systems: static and dynamic. The former is defined by formal model checking or other methods. This approach relies on state space exploration based on abstract representations of system level models [GD03; Per+04; Wei+05]. The formal methods are very successfully applied to formal verification of the hardware, and less successfully to formal verification of the embedded software. Due to state explosion problems, they are not widely used for verification of SoCs. In the sequel we discuss only dynamic verification approaches.

[1] In the literature the Design Under Verification (DUV) is sometimes referred to as the Design Under *Test* (DUT). In this document we stick to the DUV abbreviation.

The dynamic *simulation-based verification* exercises the DUV at runtime. Its fundamental operation is the process of the DUV *state activation* followed by the DUV *response checking* (Fig. 2.4). All functional verification can be described by a series of such fundamental steps. Applying sequences of *input stimuli* to the DUV one may put it in the *goal state* (the state of interest), and then check that the DUV's output or its internal execution in that state is correct with respect to the specification. A state of the DUV may refer to the content of its registers, the state of its state machine, etc. For instance, suppose we want to ensure that the GPIO sends an interrupt when a press of a button occurs. To do that, it would be necessary to apply a keystroke to the GPIO (a stimulus), and then to check if the GPIO produces an interrupt (an output). Sometimes the simulation-based verification is also referred to as *testing* of the design due to its runtime nature.

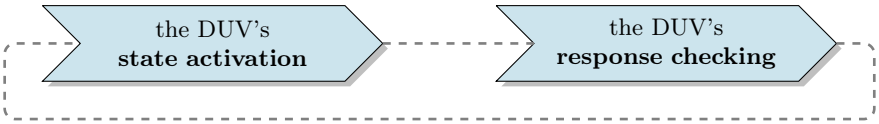


Figure 2.4 Fundamental steps of the simulation-based verification.

2.1.3.3 Verification Progress

The aim of verification is to generate the necessary stimuli to put the DUV into all *goal states* and ensure that the DUV's outputs, or its internal behavior, are compliant with the design specification in those states. Goal states are defined at the very beginning of the verification process. Usually they represent states of the DUV where malfunctions are likely to occur. The verification progress is measured by the covered sequences of states of the DUV leading to the goal states. These sequences are referred to as *scenarios*. For instance, one might be interested in checking that the interrupt controller (INTC) sends an interrupt to the CPU whenever it gets an interrupt either from the LCDC or the GPIO. The goal state here would be the state of the INTC with one pended interrupt to be sent. The sequence of input stimuli leading to this state would include (i) receiving an interrupt from the LCDC, (ii) receiving an interrupt from the GPIO, etc. The set of scenarios to reach all goal states of the DUV defines the *functional coverage of the design* [YWD14; AMZ04; Hel+06; LGD10; HMMC09]. Using functional coverage one may check the verification progress and define the set of scenarios which should be exercised further.

2.1.3.4 Structure of a Testbench

To automate the verification process of the DUV at runtime, testbenches[2] are constructed. The basic components of a testbench are shown in Fig. 2.5 as it is defined by the *Universal Verification Methodology* (UVM) [AO14b], the widely accepted Accellera[3] standard. The components of the testbench are (i) a stimuli generator, (ii) a coverage improver, (iii) an assertion checker.

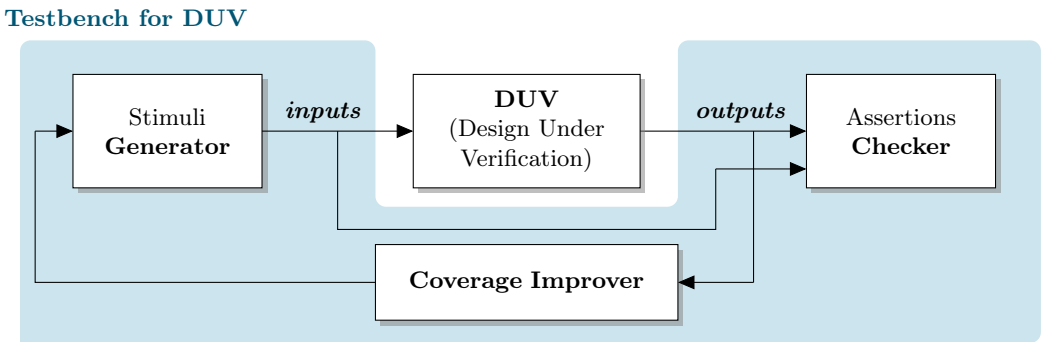


Figure 2.5 Simplified view of a testbench for testing hardware design.

[2]In the literature a testbench is also called a *verification environment* of the DUV (see for instance [AO14a; AO14b; IJ04]).

[3]Accellera Systems Initiative is an independent, not-for profit organization dedicated to create, support, promote, and advance system-level design, modeling and verification standards for use by the worldwide electronics industry [Acc].

The stimuli *generator* (also known as *traffic generator*) provides input stimuli (or simply inputs) for the DUV. One may distinguish *direct*, *constrained random*, and *coverage driven stimuli generation*. In direct stimuli generation specific inputs are created for each scenario. It is usually used to cover very specific corner cases which are difficult to exercise in any other way.

Constrained random stimuli generation relies on randomization to automatically generate constrained random sequences of inputs. In this approach a list of inputs (e.g., atomic transactions) and their corresponding valid orderings, valid data content, and valid parameter content are defined by *constraints*. The constraints are defined by means of languages with support of random variables and ranges (e.g., *e* [IJ04], CRAVE [Hae+12; LD14], SCV [GED07; Wil+09; Opeb]). Provided the set of constraints, the generation of random stimuli relies on different solving approaches such as Binary Decision Diagrams (BDDs) [Ake78], Boolean Satisfiability Theory (SAT) [ES03] or Satisfiability Module Theories (SMT) [Bar+09]. Usually different solvers implementing one of those approaches are used to get inputs (e.g., Boolector [BB09], SWORD [Wil+07], Z3 [DMB08], etc.). When the set of constraints is defined, they may *contradict* each other. The occurring contradictions should be detected and resolved before generation is started [Gro+08; Gro+09]. Random generation of inputs leads to far more covered scenarios of the DUV than it could possibly be done with direct inputs. At the same time there is very low probability to reach corner cases. It is a reason why constrained random generation is usually used together with directed stimuli generation. Due to automation of stimuli generation a huge set of scenarios can be executed, however some of the generated inputs can be repeated. Moreover, it is difficult to measure verification progress, since the set of scenarios being generated is known only during simulation.

Coverage driven stimuli generation is the unification of coverage collection performed by the **coverage improver** (see Fig. 2.5) and the constrained random stimuli generation described above. In this case the results of coverage collection is used to guide random generation of inputs.

The assertion *checker* ensures that the DUV's responses are compliant with its specification. If the DUV is defined at the Register Transfer (RT) of Cycle Accurate (CA) level of abstraction, specification properties (assertions) are interpreted on discrete time scales defined by *clocks*. If the DUV is defined at the TL level, the unique discrete time scale is recovered from sequences of *events*. Checking of the DUV's assertions is usually referred to as *Assetions Based Verification* (ABV) [Bom+07; Fos09; Fos09].

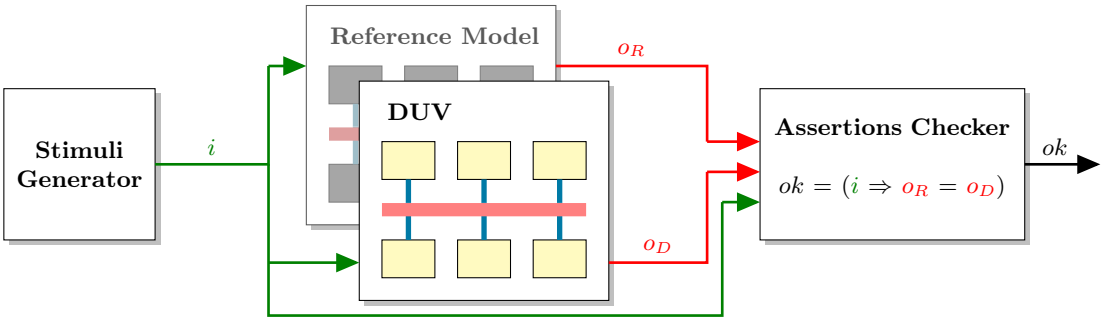


Figure 2.6 Black-box assertion checking for system level verification.

One may distinguish two types of checking: *black-box* and *white-box*. The *black-box* checking verifies only the boundary (i.e., inputs and outputs) of the DUV. If a property of a device cannot be verified through its ports, it is either not controllable (i.e., cannot be activated), or not observable. Usually black-box checking requires a *reference model* to check that responses generated by the DUV are in fact the expected ones. As a reference model one may use either the model of the DUV at a higher abstraction level than the one being verified (e.g., the TL model of the DUV for the verification at the RT level), or the properties of the DUV expressed in languages with support of temporal constructs (e.g., PSL [18505], SVA [Cer+14]). The work of the assertion checker then consists in verifying that the outputs of the DUV are the same as the outputs of the reference model, triggered with the same inputs as the verified DUV (Fig. 2.6). Such type of checking ensures consistency of the design with the reference model. Since only boundaries of the DUV are observed, some bugs can be detected with the delay which makes it difficult to localize their source. Due to that the black-box testing is dominant during system level verification when the majority of the errors are detected in port interfaces and in the communication between modules.

During the module design phase most of the erros are in the function of the module. To enhance debugability of the DUV and localization of bugs *inside* the DUV, monitors and checkers are used. They track internals of the DUV with the potential to become sources of malfunctions. Such an approach is

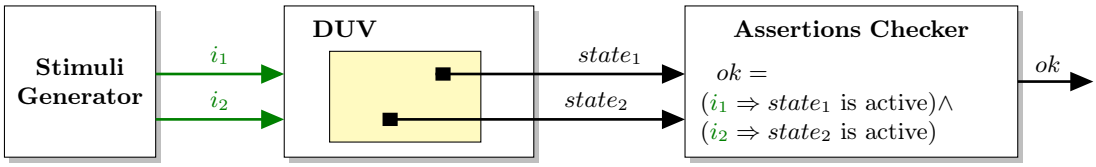


Figure 2.7 White-box assertion checking for module level verification.

referred to as *white-box* verification. It does not use a reference design (Fig. 2.7). White-box functional verification is usually used for smaller modules in the early stages of the design process, or for RTL design where checkers are embedded into the design and/or the system boundary. This assertion technique provides effective checks to detect the design problems at exact timing of the false behavior, although it can be hard to reuse and manage [IJ04].

2.1.4 SystemC

The information provided in this section is inspired by [Sys; DCBK10]. SystemC is a system design and modeling language based on C++. Strictly speaking SystemC is a C++ library [DCBK10]. It was standardized by the Open SystemC Initiative (OSCI) and approved as IEEE 1666 in 2006. It consists of the language itself and potential methodology-specific libraries, e.g., SCV[4]. SystemC can be used to model SoCs at different levels of abstraction from RTL up to TLM. In this section we briefly cover features of SystemC necessary for the understanding of the present work. For more complete information, the reader is invited to consult the SystemC Language Reference IEEE 1666-2011 [Sys].

2.1.4.1 Overview

SystemC addresses the modeling of both software and hardware using C++. It implements the structures necessary for hardware modeling enabling concepts of *hierarchy*, *time*, *concurrency*, *communications* and *hardware data types*. Figure 2.8 shows the structure of the library. The basic unit of SystemC design is a *module* which encapsulates a design component (e.g., IP). Modules may contain other modules (sub-components), simulation processes referred to as *threads*, channels and ports for connectivity. Simulation processes are executed *only* by the event-driven non-preemptive SystemC *scheduler* (the simulation kernel). They appear to execute concurrently. The user can indirectly control the execution of processes by the kernel by means of *events* and *notifications*. For instance, to make the scheduler to resume a thread P1 waiting for an event `e`, another thread P2 should notify `e`. Few seconds of the wall-clock time can be needed to simulate several milliseconds of the design.

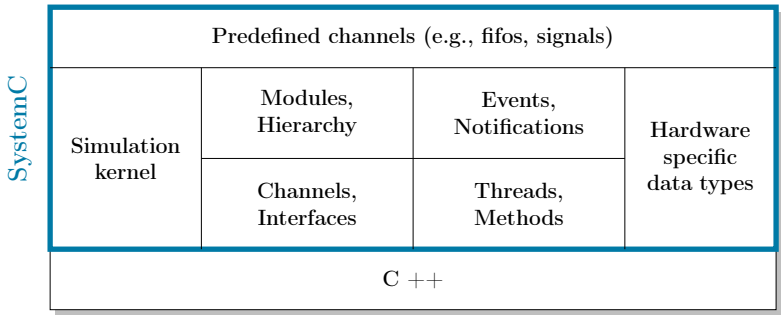


Figure 2.8 Simplified structure of SystemC library.

Channels can represent both very simple communications such as a wire or a FIFO, or complex communications schemes that eventually map to significant hardware such as the AMBA bus. SystemC provides several predefined channels common to software and hardware design (Fig. 2.8). These built-in channels include locking mechanisms like mutex and semaphores, as well as hardware concepts like FIFOs, signals and others. SystemC modules connect to channels and other modules via *ports*.

The following sections provide technical and syntactical details about the concepts introduced above. Their understanding is the prerequisite for Chapters 3, 6 and 12. The reader already familiar with the material may skip the rest of this section and move directly to Section 2.1.5.

[4]SystemC Verification Library

2.1.4.2 Modules

A SystemC *module* (component) is a C++ class definition which can be defined with a macro `SC_MODULE` (Fig. 2.9). It may contain constructors (`SC_CTOR` macro in Fig. 2.9(a)), destructors, ports, member channels and data instances, member module instances, simulation process member functions (threads and methods), and other functions. All of these items are explained in the sections below.

```
#ifndef NAME_H
#define NAME_H
SC_MODULE(Name) {
  // port declaration
  // channel/submodule instances
  SC_CTOR(Name) {
    // connectivity
    // process registrations
  }
  // process declarations
  void thread();
  ...
  // function declarations
  void func(int arg);
  ...
};
#endif
```

(a) `Name.h`

```
#include <systemc>
#include "Name.h"



...
/* Implementation of processes
   and functions */
void Name::thread(){
   // Implementation
}
...
void Name::func(int arg){
   // Implementation
}

...
```

(b) `Name.cpp`

Figure 2.9 Syntax of the `SC_MODULE`.

2.1.4.3 Simulation Processes

A basic unit of concurrent execution of SystemC is the SystemC *simulation process* registered within the SystemC simulation kernel (discussed below). The registration of processes is done by the constructs `SC_THREAD` or `SC_METHOD` (Fig. 2.12(a)). The simulation kernel then schedules and calls each of the registered processes as needed (see Sec. 2.1.4.8 below).

When a simulation process runs, its small segment of code is executed and then control is returned to the simulation kernel. The SystemC simulator is *non-preemptive*, i.e., it cannot force a running process to return control. Thus, it is required for the processes to suspend themselves (i.e., to return control to the kernel) to allow SystemC to proceed to execute other processes. An `SC_THREAD` may suspend itself calling the `wait()` function specifying an event or a time-out (see below). When `wait()` returns, the process is resumed. For instance, Figure 2.12(b) provides an example of a process where the function `wait(TIME)` is used to simulate the passage of time. `SC_METHOD` processes never suspend internally, i.e., they can never invoke `wait()`. Instead, they execute without interrupts and return to the caller (the kernel); simulation time does not pass between the invocation and return of `SC_METHOD`s.

```
//FILE: blinker.h ------------
SC_MODULE(Blinker) {
  bool blinker;

  SC_CTOR(Blinker) {
    SC_THREAD(T);
  }
  void T();
  ...
};
```

(a) Registration of the process with `SC_THREAD` in the kernel

```
//FILE: blinker.cpp ------------
void Blinker::T(){
  while(true){
    blinker = true;
    cout << "Blink ON" << endl;
    wait(200, SC_MS);
    cout << "Blink OFF" << endl;
    blinker = false;
    wait(200, SC_MS);
}}
```

(b) Example usage of `wait(TIME)`

Figure 2.10 A process using `wait()`.

An `SC_THREAD` can be invoked by the kernel only once; if it terminates, it cannot be restarted. An `SC_THREAD` typically begins execution at the start of simulation and continues in an endless loop until the simulation ends (e.g., see Fig. 2.12(b)). Each time `SC_THREAD` returns control to the simulation kernel calling `wait()`, its execution state is saved (variables of the process are persistent), which lets the process

be resumed when the `wait()` returns. On the contrary, an `SC_METHOD` must initialize variables each time the method is invoked.

Only the SystemC scheduler is allowed to call both `SC_THREAD`s and `SC_METHOD`s. The user can indirectly control the execution of processes by the kernel by means of *events*, *notification* and *sensitivity*.

2.1.4.4 Events, Notification, Sensitivity

SystemC is an event-driven simulator. For instance, the time-out of a `wait(TIME)` is an *event*. Events are modeled with the `sc_event` SystemC class. The class `sc_event` allows explicit triggering (causing) of events by means of a notification method `notify()` (Fig. 2.11). There are three types of notifications: (i) immediate, (ii) delayed and (iii) timed. They are specified as shown in Fig. 2.11. To make simulation processes `SC_THREAD`s and `SC_METHOD`s react on occurrences of events, one should define the *sensitivity* of those processes to appropriate events. It can be of any of the following types:

1. *Static* sensitivity is implemented by applying SystemC `sensitive` command on `SC_METHOD` or `SC_THREAD` within the constructor (Fig. 2.12(a)).

2. *Dynamic* sensitivity lets a simulation process change its sensitivity on the fly. The `SC_METHOD` implements dynamic sensitivity with a `next_trigger(arg)` command; the `SC_THREAD` implements it with a `wait(arg)` command (Fig. 2.12(b)).

```
sc_event e;
/* notification of event */
e.notify();              // immediate
e.notify(SC_ZERO_TIME);// delayed
e.notify(100, SC_NS);   // in 100 nonaseconds
```

Figure 2.11 Syntax of `sc_event`.

```
// FILE:  Server.h ------------
SC_MODULE(Server)
{int tasks_N;
 sc_event e_task_arrival;
 sc_event e_service_request;
 sc_event e_service_finish;

 SC_CTOR(Server)
 {tasks_N = 0;
  SC_THREAD(task_generator_T);

  /* static sensitivity */
  SC_METHOD(queue_M);
  sensitive<<e_task_arrival;

  SC_THREAD(server_T);
  sensitive<<e_service_finish
           <<e_task_arrival;

  SC_METHOD(service_M);
  sensitive<<e_service_request;
 }
 void task_generator_T();
 void queue_M();
 void server_T();
 void service_M();
};
```

(a) Static sensitivity in a module's constructor

```
1  // FILE:  Server.cpp ------------
2  void Server::task_generator_T(){
3    sc_time ARRIVAL_T(400, SC_MS);
4    while(true){
5      wait(ARRIVAL_T);
6      cout<<"New task arrived\n";
7      e_task_arrival.notify();
8  }}
9  void Server::queue_M(){
10   tasks_N +=1;
11 }
12 void Server::server_T(){
13   while(true){
14     while (tasks_N != 0){
15       e_service_request.notify();
16       wait(e_service_finish);
17       cout<<"Task departs\n";
18       tasks_N -= 1;
19     }
20   wait(e_task_arrival);
21 }}
22 void Server::service_M(){
23   sc_time SERVICE_T(400, SC_MS);
24   cout<<"Task is accomplished\n";
25   next_trigger(SERVICE_T);
26   e_service_finish.notify();
27 }
```

(b) Examples of dynamic sensitivity: lines 5, 25

Figure 2.12 Sensitivity of processes.

2.1.4.5 Communication

Communication between concurrent `SC_THREAD`s can be done by exchanging events and ordinary module member data. It is unsafe because the processes may miss events[5]. To ensure safe data communication between simulation processes, it is important to update a handshake variable indicating when a request for data is made, and clear it when the request is acknowledged. This communication principle is encapsulated in SystemC built-in *channels* such as `sc_signal<T>` .

All channels inherit from and implement one or more SystemC *interface* classes[6]. For instance, the interface `sc_signal_inout_if<T>` lets a module have access to the value of a channel through the `read()` and `write()` methods. It is inherited by `sc_signal<T>` (Fig. 2.19). Simply by referring to interfaces, one may implement modules independently of the implementation details of the communication channels. For instance, in Figure 2.13 the implementation of the modules `modA`, `modB`, `modC` and `modD` does not depend on the implementation of the `complex_bus_channel`; only the interface of the latter is known.

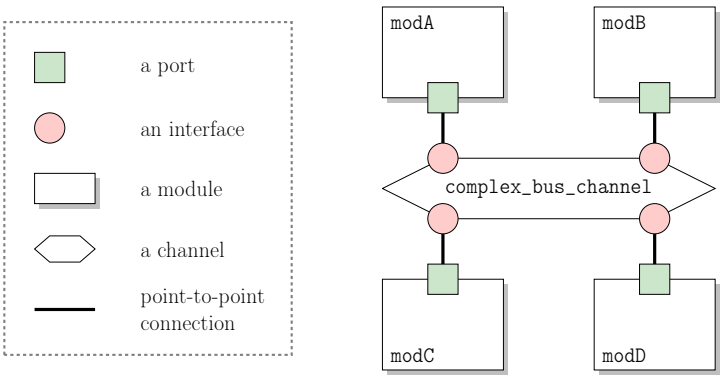


Figure 2.13 The power of interfaces in SystemC. (Source [DCBK10])

Channels are connected to modules by means of *ports* referring to respective interfaces of those channels (Fig. 2.13). There are two types of ports in SystemC. A port `sc_port<T>` is a pointer to the channel *outside* the respective module. For instance, Figure 2.14(b) shows an example where two processes `A_thread` and `B_thread` of two respective distinct modules `modA` and `modB` communicate through the FIFO channel `c`. `A_thread` in module `modA` sets a value to a local variable `v` of `modB` by calling the `write` method of the channel `c`. `B_thread` in module `modB` retrieves a value via the `read` method of `c`. The access is accomplished by means of the pointers `pA` and `pB`.

The idea of `sc_export<T>` is to move the channel *inside* the defining module, thus hiding some of the connectivity details and using the port externally as if it were a channel. The `sc_export<T>` allows control over the interface. The concept is illustrated in Figure 2.14(c). Ports are always defined within the module class (Fig. 2.15).

2.1.4.6 The Entry Point of the SystemC Simulation

The starting point of execution of a SystemC model (program) is called `sc_main()` (the analog of `main()` in C/C++). Within `sc_main()` the code executes in three distinct stages (see Fig. 2.16):

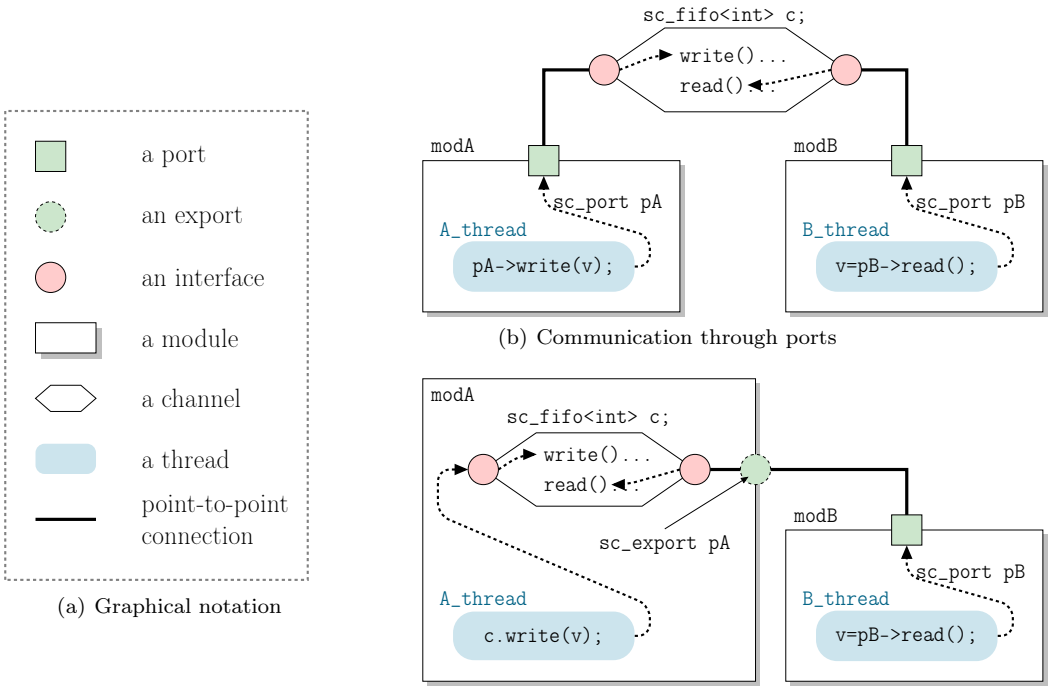
1. During the *elaboration* phase SystemC components are instantiated and connected to create a model ready for simulation.

2. *Simulation* is started with the call to the `sc_start()` method. The simulation starts with the initialization: the kernel identifies all simulation processes and places them in either the runnable or waiting process sets (more details below). When the initialization is finished, the scheduler starts to execute the processes (potentially) advancing the simulation time.

3. The *post-processing* phase is optional, it handles results of the simulation.

[5] When processes P1 and P2 communicate via an event `e`, the process interested in the event, let say P2, can miss `e` notified by P1 due to implementation of the SystemC scheduler. To capture an occurrence of `e`, P2 needs to be executed by the simulation kernel in *the same evaluation phase* of the simulation as P1 when it produces `e` (see Sec. 2.1.4.8)

[6] The SystemC concept of an interface class is the one of C++, i.e., it is a class which contains no data members and only pure virtual methods like `virtual void foo()=0` [Mey14].



(a) Graphical notation

(b) Communication through ports

(c) Communication through exports

Figure 2.14 Communication mechanism in SystemC. (Source [DCBK10])

```
// FILE: INTC.h-----------------
...
SC_MODULE(INTC){
 //sc_port<sc_signal_in_if<bool> >
 sc_in<bool> irq_in0;
 sc_in<bool> irq_in1;

 //sc_port<sc_signal_inout_if<bool> >
 sc_out<bool> irq_out;
 ...
};
```

(a) Definition of ports

```
// FILE: main.cpp-----------------
#include "INTC.h"
...
int sc_main(int argc, char** argv){
  INTC intc("Interrupt Controller");
  sc_signal<bool> intr_channel;
  intc.irq_out(intr_channel);
  ...
  sc_start();
  return 0;
}
```

(b) Binding at elaboration phase

Figure 2.15 Example of defining ports within a module class definition.

```
int sc_main(int argc, char** argv){
  /* elaboration */
  sc_start(); /* <--simulation starts here*/
  /* post-processing */
  return 0; }
```

Figure 2.16 Syntax of the `sc_main()`.

2.1.4.7 Simulation Time

Simulation time is tracked by the kernel by means of the data type `sc_time` . SystemC defines time units like `SC_NS` for nanoseconds, `SC_US` for microseconds, etc. The current simulated `time_value` can be obtained by means of the method `sc_time_stamp()` (Fig. 2.17).

```
sc_time current_time = sc_time_stamp();
```

Figure 2.17 Getting the current simulation time in SystemC.

2.1.4.8 SystemC Simulation: The Scheduling Algorithm

In this section we briefly examine operations of the simulation kernel which are schematically presented in Figure 2.18. The simulation is started with the call to `sc_start()` which invokes the simulation kernel. Being activated the kernel starts the *initialization* phase: it identifies all simulation processes P_is and places them in either the set of *runnable* processes R, if they can start immediately, or in the set of *waiting* processes W, if they need events to be activated. When initialization is finished, the kernel starts the simulation scheduling the processes to run and advancing simulation time. The simulation can be described as a state machine with three states (phases) (see. Fig. 2.18):

1. *Evaluation.* All runnable processes $R = \{P_1, \dots P_n\}$ are run one at a time. Each process P_i runs until either it executes a `wait()` (or `wait(TIME)`), or returns. If `wait(TIME)` is executed, the time information is stored as an upcoming time event in the scheduling priority queue. The phase continues until there are no runnable processes left, i.e., $R = \varnothing$. During evaluation a waiting process can become runnable if it is waiting for an event `e` which is *immediately* notified with `e.notify()` in the same evaluation phase. In this case, the process is immediately added to R during the evaluation phase.

2. *Update.* All delayed events `e`s, notified in the preceding evaluation phase with `e.notify(SC_ZERO_TIME)`, take place. Processes waiting for those events become runnable. If after update there are runnable processes (i.e., $R \neq \varnothing$), the kernel moves back to the evaluation phase[7]; otherwise it advances time.

3. *Advancement of Time.* Once the set of all runnable processes has been emptied, the simulation time is advanced. Simulated time is moved forward to the closest time with a scheduled event. All processes waiting for that particular time are moved into the runnable set, allowing the evaluation phase to resume.

The alternation between evaluation, update and time advancement continues until one of three things occurs: (i) all processes have yielded and there are no delayed events (i.e., there is nothing in the runnable set), (ii) a process has executed the function `sc_stop()`, (iii) maximum simulation time is reached (i.e., internal 64-bit time variable runs out of values). At this point simulation stops.

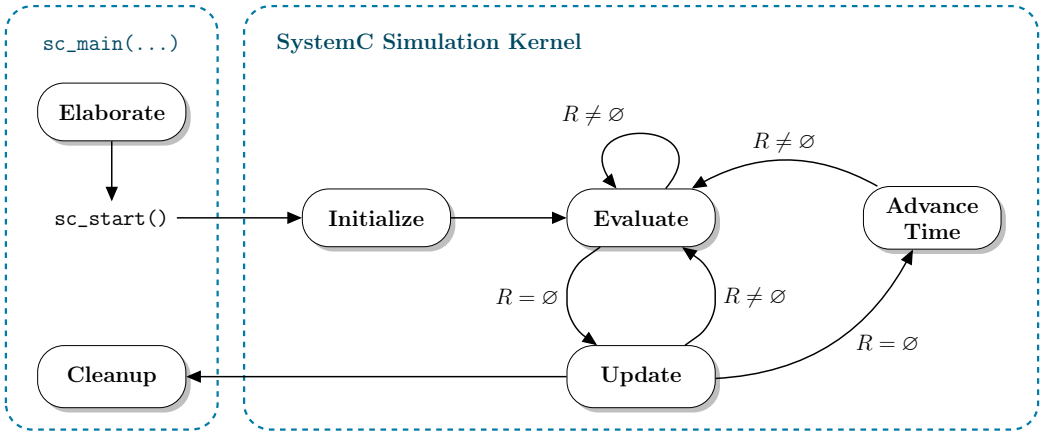


Figure 2.18 The SystemC scheduling algorithm: R is the set of runnable SystemC processes. (Source [DCBK10])

Revisiting `sc_signal<T>` The `sc_signal<T>` uses the update phase as a point of data synchronization. To accomplish the synchronization, it has two storage locations: the current and the new value. When a process writes to a `sc_signal<T>` calling its method `write()`, the process stores into the new value. By calling the method `request_update()` the process notifies the kernel. When the update phase occurs, the simulation kernel calls the `update()` method of the channel which copies the new value into the current value of the channel. The `value_changed_event()` method returns a reference to an `sc_event` which is notified any time the update causes a change in value. If any process is waiting on changes in the channel, it becomes runnable and executes during next evaluation. The principle is illustrated in Figure 2.19.

[7] An evaluation followed by an update is referred to as a *delta cycle*.

```
sc_sginal<string> msg_sig;

cout<<"Initialize during 1st delta cycle: ";
msg_sig.write("Hello World!");
cout<<"msg_sig is '"<<msg_sig<<"'\n";
wait(SC_ZERO_TIME);

cout<<"2nd delta cycle: ";
msg_sig.write("Bye!");
cout<<"msg_sig is '"<<msg_sig<<"'\n";

cout<<"3rd delta cycle: ";
msg_sig.write("Have a nice day!");
cout<<"msg_sig is '"<<msg_sig<<"'\n";

/* OUTPUT ----------------------------------------
Initialize during 1st delta cycle: msg_sig is ''
2nd delta cycle: msg_sig is 'Hello World!'
3rd delta cycle: msg_sig is 'Bye!'
-----------------------------------------------------*/
```

Figure 2.19 Example of the `sc_signal<T>` update.

2.1.4.9 Process Execution and Time Advancement: an Example

To better understand how time and execution interact, consider a design with three hypothetical processes as illustrated in Figure 2.20. We assume that t_1, t_2 and t_3 are non-zero. Each process contains statements ($stmt_{A1}, stmt_{A2}, \ldots$), and wait methods (`wait(`$t_1$`)`, `wait(`$t_2$`)`). From the modeling perspective statements of processes take some time and are evenly distributed along simulation time (Fig. 2.21(a)); however actual simulated activity is such that statements execute in zero time (Fig. 2.21(b)). At intervals $[t_i, t_{i+1}]$ no simulated time elapses; all statements executed at $[t_i, t_{i+1}]$ (e.g., $B4$, $B5$, $C4$, $C5$ at $[t_3, t_4]$) are executed during the same evaluate phase in random order[8].

2.1.5 Transaction Level Modeling

TLM was introduced in 2003 [CG03]. It defines the principles of high-level modeling of component-based systems. TLM is a concept independent of any language. TLM virtual prototypes have become a de-facto standard in today's SoC design. For instance, at STMicroelectronics they are used for (i) early development of the embedded software, (ii) architectural analysis, (iii) functional verification of the hardware, where they serve as the reference model (see Sec. 2.1.3.4). The TLM-2.0 library implemented in SystemC was integrated in the IEEE 1666-2011 SystemC standard [Sys].

Figure 2.22 illustrates a standard block diagram of a TL model. Models typically consist of a set of *asynchronous components* communicating by means of *transactions*. Transactions are passed between *initiator* components (e.g., LCDC) and *target* components (e.g., RAM) through interconnects (e.g., a bus). Initiators create transaction objects; targets execute received transactions; interconnects are neither initiators nor targets of transactions, they simply pass on transactions between TLM components.

2.1.5.1 Abstraction Levels of TL Models

The abstraction level of a TL model is defined based on the granularity of time, the function and communication abstraction. TL models can be defined at any of the following abstraction levels.

1. For *untimed* TL models the notion of simulation time is unnecessary. Processes of TLM components yield at explicit pre-determined synchronization points. For instance, if the model is implemented in SystemC, the synchronization point could be the call to `wait()`.

2. When the model is *loosely-timed*, each transaction has two timing points marking the start and the end of the transaction. Simulation time is used but processes may be temporally decoupled from simulation time. Each process knows its *time quantum*, i.e., how far it can run ahead of simulation time. A process may yield either because it reaches an explicit synchronization point (e.g., the SystemC's `wait()`) or because it has consumed its time quantum.

[8] It is possible to impose the order of processes execution by means of *delta-cycle*.

```
Process_A(){
   /* t0 */
   stmtA1;
   stmtA2;
   wait(t1);
   stmtA3;
   stmtA4;
   wait(t2);
   stmtA5;
   wait(t3); }
```

```
Process_B(){
   /* t0 */
   stmtB1;
   wait(t1);
   stmtB2;
   stmtB3;
   wait(t3);
   stmtB4;
   stmtB5;
   wait(t4); }
```

```
Process_C(){
   /* t0 */
   stmtC1;
   stmtC2;
   stmtC3;
   wait(t3);
   stmtC4;
   stmtC5;
   wait(t4);
}
```

Figure 2.20 Three SystemC processes. (Source [DCBK10])

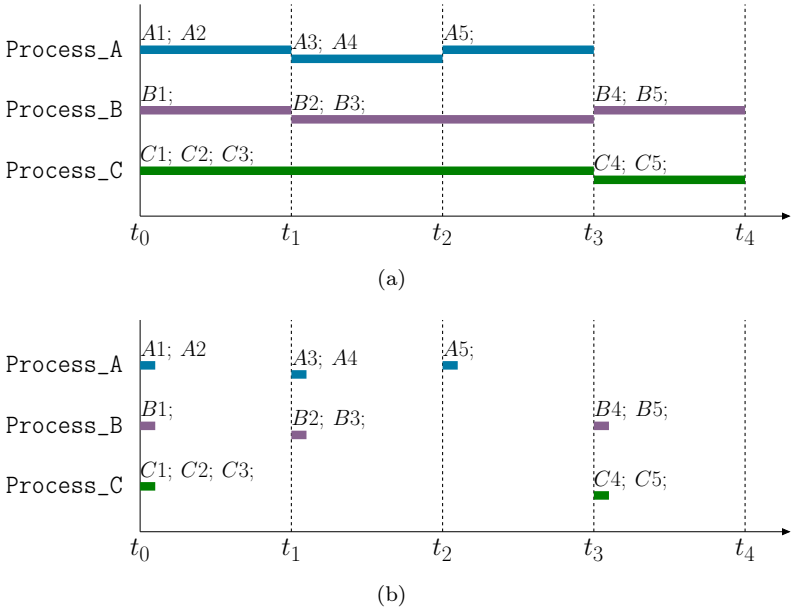


Figure 2.21 Simulated activities of three SystemC processes: a) perceived, b) actual. Solid line portions indicate program activity. Vertical discontinuities indicate a wait. (Source [DCBK10])

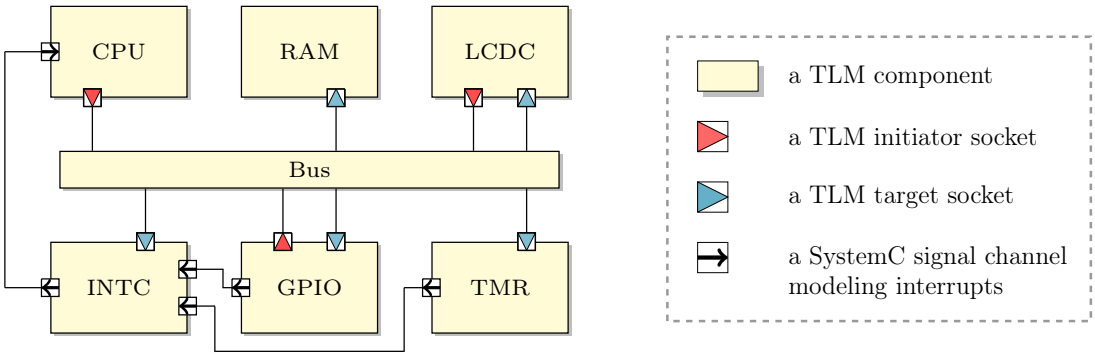


Figure 2.22 Example of a TLM virtual prototype.

3. When the model is *approximately-timed* each transaction can be associated with multiple protocol-specific timing points. Processes are typically forbidden to run ahead of simulation time.

2.1.5.2 Communication Principles

To enable communication TLM components are equipped with TLM *initiator/target sockets* (see. Fig. 2.23) which implement TLM *transport interfaces* (see below). A pair of connected initiator and target sockets form a hop of the communication path. The initiator component passes on a transaction object via its

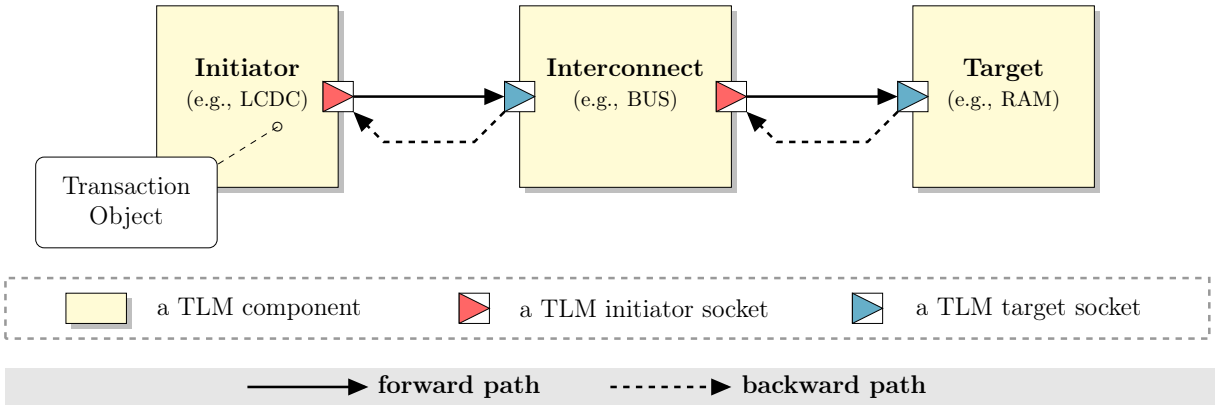


Figure 2.23 Communication via TLM sockets.

initiator socket by calling a respective method of the transport interface of the connected target socket. The target socket can belong either to the interconnect or to the target component. The interconnect component implements the transport interface; it may read attributes of the transaction object before passing it on to a further transport call. This transport call is implemented either by another interconnect component or by a target. This sequence of method calls is known as the *forward path*. The target acts as the final destination for a transaction; when the latter is executed it can be returned to the initiator either with the return from the transport method call (the *return path*), or by an explicit transport method call performed by the target back to initiator (the *backward path*).

Transport interfaces The transport interfaces define communication abstractions of TL models:

1. The *blocking* transport interface is used to model the start and end of a transaction, with the transaction being completed within a single function call. It is typically used for loosely-timed models (see Sec. 2.1.5.1).

2. The *non-blocking* transport interfaces allow a transaction to be broken down into multiple timing points, as a rule it requires multiple function calls for a single transaction. These interfaces are used to implement communication protocols of approximately-timed TL models.

2.1.5.3 TL Models Implemented in SystemC

A TLM component is implemented as a SystemC `sc_module` (Fig. 2.24). SystemC simulation processes (threads and methods) are used to model the behavior of the TLM component. When the component is an initiator (resp. a target) the `sc_module` possesses an initiator (resp. target) socket (Fig. 2.24).

```
// using namespace sc_core, tlm

SC_MODULE(TLM_Component){
  /*TLM sockets*/
  tlm_target_socket<...> target_socket;
  tlm_initiator_socket<...> initiator_socket;

  SC_CTOR(TLM_Component){
    /*register threads and methods in
      SystemC simulation kernel*/
  }
  /*threads, methods*/
  /*functions*/
  ...
};
```

Figure 2.24 Declaration of a TLM component in SystemC.

The forward and backward communication paths of TL models are enabled by means of SystemC ports . A TLM initiator (target) socket is presented by the class `tlm_initiator_socket<...>` (resp. `tlm_target_socket<...>`). The `tlm_initiator_socket<...>` inherits from the SystemC `sc_port` and

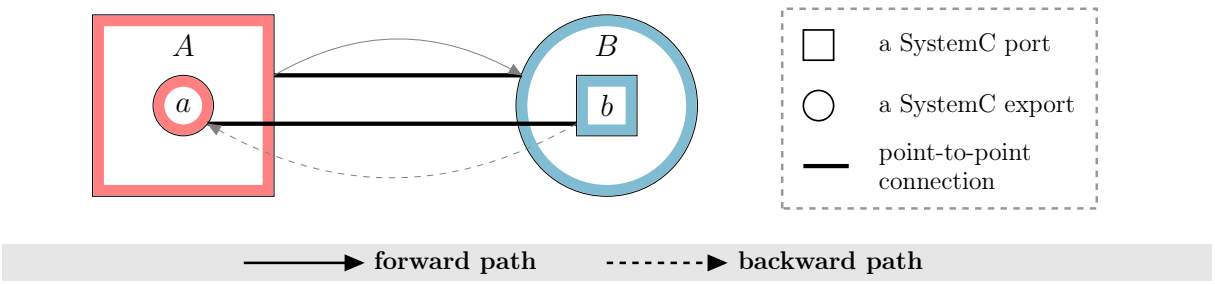


Figure 2.25 Implementation of TLM sockets in SystemC: A (resp. B) is the TLM initiator (resp. target socket).

contains a member which is the SystemC export `sc_export`. It is reverse for `tlm_target_socket<...>`. When the initiator socket A is connected to the target socket B (see Fig. 2.25), A as a SystemC port is bound to B which is a SystemC export; a SystemC port b of B is connected to a SystemC export a of A. The connected pairs of ports/exports should be compatible, i.e., exports and ports should refer to the same transport interfaces of TLM. Those interfaces can be implemented either by parent components of the sockets, or by the sockets themselves. The call from A to B (resp. from b to a) corresponds to the forward (resp. backward) communication path.

Transactions are represented by the class `tlm_generic_payload`. It allows to set parameters of a transaction such as address, data, response status, etc.

All TLM components constituting our running example (see Chapter 3) are implemented in SystemC as it is described in this section.

2.2 Formal Models for Discrete Concurrent Systems

2.2.1 Synchronous Models

The synchronous models have been proposed to describe *reactive systems*. Reactive systems are computer systems that continuously react to their environment at a speed determined by this environment (e.g., control, supervision systems) [HP85; Ber02]. Each internal or output event of the program is precisely dated with respect to the *flow of input events*. The synchronous models for reactive systems are based on the assumption that the program reacts rapidly enough to perceive all the external events. This assumption is referred to as the *perfect synchrony hypothesis*. It allows to decouple logical time on which we reason and physical time. The synchronous models are also used for modeling, simulation and validation of safety critical systems.

2.2.1.1 Synchronous Mealy Machines

Synchronous Mealy machines is a formalism that can be used to model reactive systems. One transition of an automaton corresponds to one reaction. Mealy machines has been originally proposed in [Mea55] for the synthesis of synchronous circuits. Here we define ordinary Mealy machines and their synchronous product as in [MR01] since they are extensively used in the presented work.

Definition 1: Mealy Machine — A tuple $\mathbb{M} = (\mathbb{S}, s_0, \mathbb{I}, \mathbb{O}, \mathbb{T})$ is a Mealy machine where

$$\begin{cases} \mathbb{S} & \text{is the set of states,} \\ s_o \in \mathbb{S} & \text{is the initial state,} \\ \mathbb{I} & \text{is the set of input Boolean variables,} \\ \mathbb{O} & \text{is the set of output Boolean variables,} \\ \mathbb{T} \subseteq (\mathbb{S} \times \mathbb{B}(\mathbb{I}) \times 2^{\mathbb{O}} \times \mathbb{S}) & \text{is the set of transitions.} \end{cases}$$

$\mathbb{B}(\mathbb{I})$ denotes the set of Boolean formulas with variables in $\mathbb{I}$. For $t = (s, \ell, \mathcal{O}, s') \in \mathbb{T}$, $s, s' \in \mathbb{S}$ are the source and target states, $\ell \in \mathbb{B}(\mathbb{I})$ is the triggering condition of the transition, and $\mathcal{O} \subseteq \mathbb{O}$ is the set of outputs emitted whenever the transition is triggered.

In the sequel we use the following notation to denote the triggering conditions $\ell \in \mathbb{B}(\mathbb{I})$: we use dot "." to denote the conjuction, e.g., $i.j$ is equivalent to $i \wedge j$; (ii) we use comma "," to denote disjunction, e.g., i,j is equivalent to $i \vee j$; (iii) $\bar{i}$ stands for $\neg i$.

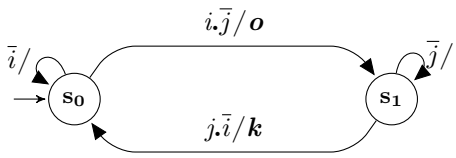


Figure 2.26 A deterministic reactive Mealy machine. s_0 is the initial state (denoted with an arrow). The inputs (resp. outputs) are $\mathbb{I} = \{i,\ j\}$ (resp. $\mathbb{O} = \{\boldsymbol{o},\ \boldsymbol{k}\}$). Arrows represent transitions; the labels are of the form *input condition/emitted outputs*. Transitions which are not defined are forbidden.

Definition 2: Determinism and Reactivity — Let $\mathbb{M} = (\mathbb{S}, s_0, \mathbb{I}, \mathbb{O}, \mathbb{T})$ be a Mealy machine. $\mathbb{M}$ is *reactive* if and only if

$$\forall s \in \mathbb{S}, \quad \left(\bigvee_{(s,\ell,\mathcal{O},s') \in \mathbb{T}} \ell \right) = \text{TRUE}.$$

$\mathbb{M}$ is *deterministic* if and only if

$$\forall s \in \mathbb{S}, \quad \forall t_i = (s, \ell_i, \mathcal{O}_i, s_i) \in \mathbb{T}, \ i \in [1,2] : \ t_1 \neq t_2 \Rightarrow \ell_1 \wedge \ell_2 = \text{FALSE}.$$

The **semantics** of a Mealy machine $\mathbb{M} = (\mathbb{S}, s_0, \mathbb{I}, \mathbb{O}, \mathbb{T})$ is given in terms of input/output/state *traces* [MR01; Alt+03].

Definition 3: Trace — Let $\mathbb{M} = (\mathbb{S}, s_0, \mathbb{I}, \mathbb{O}, \mathbb{T})$ be a Mealy machine and let $v_0, \ldots, v_n, \ldots$ be a (possibly) infinite sequence of valuations of the input variables such that $\forall i, \ v_i : \mathbb{I} \to \{\text{TRUE}, \text{FALSE}\}$. A *trace* is a sequence of tuples $t = \{(v_i, \mathcal{O}_i, s_i)\}_i$ where $\mathcal{O}_i \subseteq \mathbb{O}$ are subsets of output variables and $s_i \in \mathbb{S}$ are states, such that (i) s_0 is the initial state, (ii) each tuple $(v_n, \mathcal{O}_n, s_n)$ is followed by a tuple $(v_{n+1}, \mathcal{O}_{n+1}, s_{n+1})$ if and only if there exists a transition $(s_n, \ell, \mathcal{O}_n, s_{n+1}) \in \mathbb{T}$ such that ℓ has value TRUE on v_n.

2.2.1.2 Composition Operators

The synchronous product of Mealy machines is a construct for modeling parallelism of systems [Mar92]. The parallel composition represents the set of all possible states and transitions of the system.

Definition 4: Synchronous Product of Mealy machines — Let $\mathbb{M}_1 = (\mathbb{S}_1, s_{10}, \mathbb{I}_1, \mathbb{O}_1, \mathbb{T}_1)$ and $\mathbb{M}_2 = (\mathbb{S}_2, s_{20}, \mathbb{I}_2, \mathbb{O}_2, \mathbb{T}_2)$ be two Mealy machines. The synchronous product of $\mathbb{M}_1$ and $\mathbb{M}_2$ is the Mealy machine

$$\mathbb{M}_1 \times \mathbb{M}_2 = (\mathbb{S}_1 \times \mathbb{S}_2, (s_{10}, s_{20}), \mathbb{I}_1 \cup \mathbb{I}_2, \mathbb{O}_1 \cup \mathbb{O}_2, \mathbb{T}),$$

where $\mathbb{T}$ is defined as

$$\frac{(s_1, \ell_1, \mathcal{O}_1, s_1') \in \mathbb{T}_1, \ (s_2, \ell_2, \mathcal{O}_2, s_2') \in \mathbb{T}_2}{((s_1, s_2), \ell_1 \wedge \ell_2, \mathcal{O}_1 \cup \mathcal{O}_2, (s_1', s_2')) \in \mathbb{T}}.$$

The synchronous product of Mealy machines is both commutative and associative, it preserves both determinism and reactivity [MR01]. It does not make any synchronization between components. Figure 2.27 illustrates an example of the synchronous product of two Mealy machines. A transition in the composed state machine corresponds to exactly one transition in each of its parallel components. Some of them loops, and emit no signal (no reaction), but they do take a transition and only one.

Encapsulation When two Mealy machines have to synchronize (communicate), the parallel composition should be used together with the *encapsulation* of some dedicated signals. Encapsulation is used to enforce synchronization between parallel components by removing some of the transitions in their synchronous product. Thus, if a signal k is the output of a Mealy machine $\mathbb{P}$ and the input of a Mealy machine $\mathbb{Q}$ (see Fig. 2.27), it may serve as a synchronization signal. The intuitive semantics of the synchronization is such that $\mathbb{Q}$ should react on its input k *if and only if* it is emitted by $\mathbb{P}$. The following transitions are therefore inconsistent and should be removed from the synchronous product of $\mathbb{P}$ and $\mathbb{Q}$:

1. $\mathbb{Q}$ reacts as if k was absent, and $\mathbb{P}$ emits k to TRUE;
2. $\mathbb{Q}$ reacts as if k was present, and $\mathbb{P}$ does not emit it.

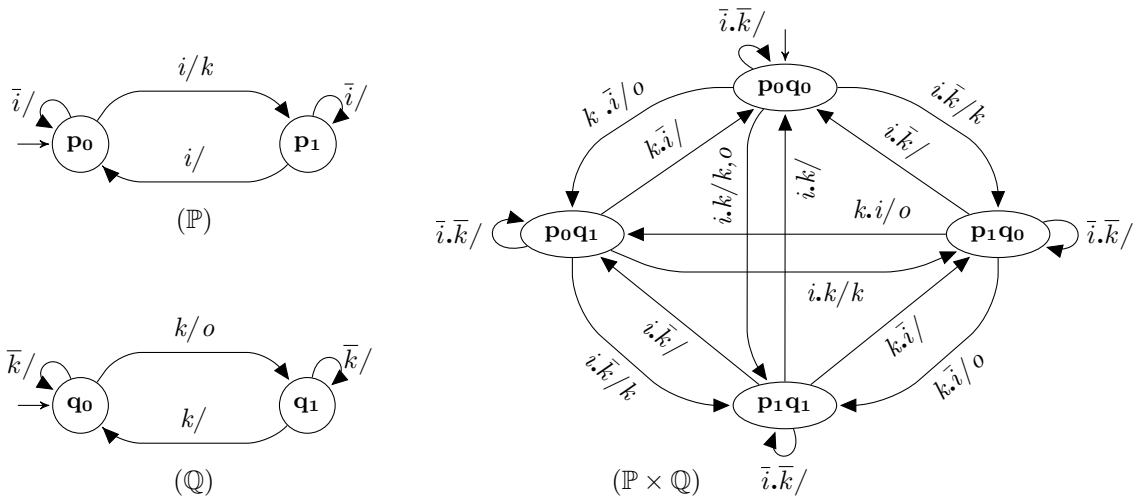
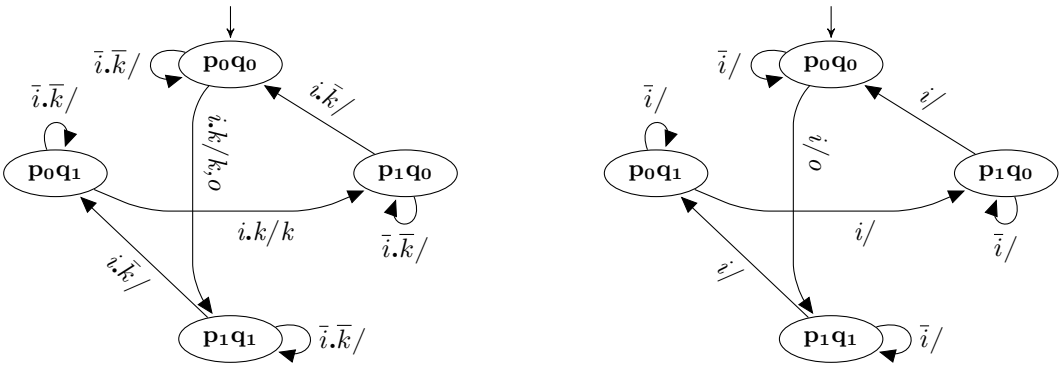


Figure 2.27 Parallel composition of two Mealy machines.



(a) Application of the encapsulation criterion for the signal k to the parallel composition $(\mathbb{P} \times \mathbb{Q})$ in Figure 2.27

(b) Hiding of the local signal k

Figure 2.28 Constructing $(\mathbb{P} \times \mathbb{Q})\backslash[k]$.

After the removal of inconsistent transitions (Fig. 2.28(a)), the synchronization signal is hidden in the labels of the remaining transitions (Fig. 2.28(b)). We denote the encapsulation of the signals $k_1, \ldots, k_n$ in the product of the Mealy machines $\mathbb{M}_1$ and $\mathbb{M}_2$ as

$$(\mathbb{M}_1 \times \mathbb{M}_2)\backslash[k_1, \ldots, k_n].$$

The formal definition of the encapsulation operator can be found in [MR01].

In general, the encapsulation operation does not preserve determinism nor reactivity. This is related to the so-called "causality" problem intrinsic to synchronous languages (see, for instance [MR01; Hal93]). However, these problems can appear only if two Mealy machines (parallel components) communicate in both directions, in the same instant. We will use encapsulation only in simple cases for which this is not necessary.

2.2.1.3 Extended Mealy Machines

A Mealy machine $\mathbb{M} = (\mathbb{S}, s_0, \mathbb{I}, \mathbb{O}, \mathbb{T})$ can be extended with a set of variables $\mathbb{V}$ of some predefined type (e.g., integers, Booleans). The transitions of the extended Mealy machine $\mathbb{M}' = (\mathbb{S}, s_0, \mathbb{I}, \mathbb{O}, \mathbb{V}, \mathbb{T})$ are of the form $(s, \phi, \ell, \mathcal{O}, \alpha, s') \in \mathbb{T}$, where $\phi \in \mathbb{B}(\mathbb{V})$ defines a condition on the values of variables in $\mathbb{V}$ ($\mathbb{B}(\mathbb{V})$ stands for the set of Boolean formulas over $\mathbb{V}$), and $\alpha : Dom(\mathbb{V}) \to Dom(\mathbb{V})$ changes the values of the variables ($Dom(\mathbb{V})$ defines the domain of the variables in $\mathbb{V}$). A transition $t = (s, \phi, \ell, \mathcal{O}, \alpha, s') \in \mathbb{T}$ is triggered if and only if both ϕ and ℓ are TRUE. In the present work we deal with Mealy machines with *counters* (e.g., see Fig. 2.29). All operators extend naturally to extended Mealy machines [MR01].

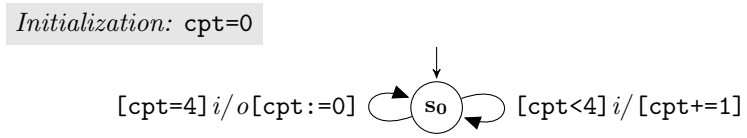


Figure 2.29 A Mealy machine with a counter `cpt`. i (resp. o) are inputs (resp. is output); `cpt` is the counter, initially `cpt = 0`. The state machine emits o each 5th occurrence of i. The transition labels are of the form $[\phi]\ell/\mathcal{O}$ $[\alpha]$, when $\phi = \text{TRUE}$ (resp. α does not change value of `cpt`) $[\phi]$ (resp. $[\alpha]$) part is omitted.

2.2.1.4 Synchronous Languages: Lustre

Synchronous models, particularly synchronous Mealy machines, are similar to the synchronous languages like Esterel [BG92], SyncCharts [And96a; And96b], Argos [MR01], Signal [BG90; LeG+91], Lustre [Hal+91]. The synchronous languages provide constructs to express the synchronous parallelism, consequent compilation into software [Ray91; HRR91; Bou+92; Bie+08], easy synthesis of hardware [RH92; RGH94; Ber92], they can also be used for verification purposes [Ver86; RDS92; HLR93; Ray10].

In this work we focus on the synchronous language Lustre. In the following chapters (e.g., see Chapters 5) we use Lustre to validate our results. Here we provide a short survey of the language and focus only on those constructs which are necessary for understanding the material presented in this work. A comprehensive description and specification of the language can be found in [Lus] or [EJ].

The language Lustre is based on the synchronous dataflow paradigm: systems are made of "nodes" and oriented wires, considered to behave in parallel. A Lustre node is a user-defined operator. A node declaration consists of: (i) an *interface specification* defining the input and output parameters with their types; (ii) declaration of *local variables*; (iii) a *system of equations* and *assertions* that defines the outputs, and possibly local variables, as functions of inputs (Fig. 2.30). The order of equations is irrelevant. Any variable x (input, output or locals) refers to a *flow*, which is a sequence of values $(x_1, x_2, \dots, x_n, \dots)$ of some elementary data types - integers, Boolean and reals. There is a notion of *global discrete clock* on which all the nodes evolve: they take one input on their input wires, and produce one output on their output wires.

```
node mealy_machine(i, j: bool) -- the interface speficication: inputs
     returns (o, k: bool);      -- and outputs
var s0, s1: bool;               -- local variables
let
   assert(not (i and j)); -- assertion:
                          -- i and j never occur simultaneously
   -- Boolean equations defining states
   s0 = true -> pre(s0 and not i) or pre(s1 and j);
   s1 = false-> pre(s1 and not j) or pre(s0 and i) ;

   -- Boolean equations defining outputs
   o = s0 and i;
   k = s1 and j;
tel
```

Figure 2.30 Lustre encoding of the Mealy machine in Figure 2.26.

Expressions `E` on right hand sides of Lustre equations are built of constants, variables, and operators. Constants are constant-valued flows (e.g., $(1, 1, \dots, 1, \dots)$). Lustre supports the following standard *data operators*: arithmetic (e.g., "+"), Boolean (e.g., `and`, `or`, `not`), conditional operators (e.g., `if-then-else`). The following *sequence operators* are available:

- the "previous" operator `pre` denotes the value of its argument at the preceding instant. For instance, if $(e_1, e_2, \dots, e_n, \dots)$ is the sequence of values of the expression `E`, `pre(E)` is a flow $(nil, e_1, e_2, \dots, e_{n-1}, \dots)$, where *nil* is an undefined value.
- the "`->`" operator (read "followed by") defines the initial value: if `E` and `F` are two expressions, with respective sequences of values $(e_1, e_2, \dots, e_n, \dots)$ and $(f_1, f_2, \dots, f_n, \dots)$, "`E -> F`" is a flow whose sequence is $(e_1, f_2, \dots, f_n, \dots)$. In other words, "`E -> F`" is initially equal to `E`, and then forever equal to `F`.

All flows evolve on the same global discrete clock.

```
-- definition of the basic Mealy machine
node mealy_machine(i: bool) returns (o: bool);
var s0, s1: bool;
let
   s0 = true -> pre(s1 and i) or pre(s0 and not i);
   s1 = false-> pre(s0 and i) or pre(s1 and not i);
   o =  s0 and i;
tel
-- encoding of the synchronous product
node product(i: bool) returns (o: bool);
var k: bool; -- the encapsulated 'wire'
let
   (k) = mealy_machine(i);
   (o) = mealy_machine(k);
tel
```

Figure 2.31 LUSTRE encoding of the synchronous product in Figure 2.27.

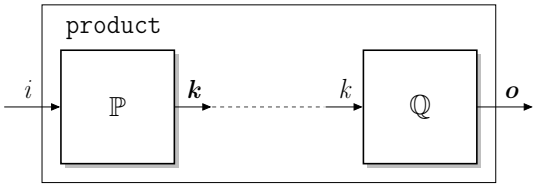


Figure 2.32 The diagrammatic view of the synchronous product of two communicating Mealy machines.

Assertions generalize equations. They are Boolean LUSTRE expressions that are assumed to be always equal to TRUE at any instant. For instance, `assert (not (i and j ))` defines the requirement that `i` and `j` do not occur simultaneously. Assertions play an essential role in program verification [RDS92; HLR93; Ray10].

The system of LUSTRE equations should be acyclic, i.e., it is forbidden to define systems of equations like `x = y + 1; y = x + 1`. These cycles are detected by a single analysis of static dependencies.

2.2.2 Asynchronous Models

Asynchronous models have been introduced in [Mil83; Mil80]. Examples of asynchronous systems are asynchronous circuits, processes on a monoprocessor system with shared memory, multiprocessor systems with shared memory, large-scale multi-computer systems, etc.

2.2.2.1 Asynchronous Parallelism

Asynchronous models model asynchronous parallelism: the parallel components of the system do not share a common clock, they can evolve independently [Gam86]. Very common modeling of asynchronous systems relies on *interleaving semantics*. It defines that an execution step of a system corresponds to a step of one of its components. If the components of the system are defined as state machines, the interleaving semantics is defined by the asynchronous product of the system's components.

Definition 5: Labeled Transition System — A tuple $\mathbb{A} = (\mathbb{S}, s_0, \mathbb{L}, \mathbb{T})$ is a labeled transition system where

$$\begin{cases} \mathbb{S} & \text{is the set of states,} \\ s_0 \in \mathbb{S} & \text{is the initial state,} \\ \mathbb{L} & \text{is the set of labels,} \\ \mathbb{T} \subseteq \mathbb{S} \times \mathbb{L} \times \mathbb{S} & \text{is the set of transitions.} \end{cases}$$

Definition 6: Asynchronous Product of Labeled Transition Systems — Let $\mathbb{A}_1 = (\mathbb{S}_1, s_{01}, \mathbb{L}_1, \mathbb{T}_1)$ and $\mathbb{A}_2 = (\mathbb{S}_2, s_{02}, \mathbb{L}_2, \mathbb{T}_2)$ be two labeled transition systems. The asynchronous product of $\mathbb{A}_1$ and $\mathbb{A}_2$ is the transition system

$$\mathbb{A}_1 \parallel \mathbb{A}_2 = (\mathbb{S}_1 \times \mathbb{S}_2, (s_{01}, s_{02}), \mathbb{L}_1 \cup \mathbb{L}_2, \mathbb{T}),$$

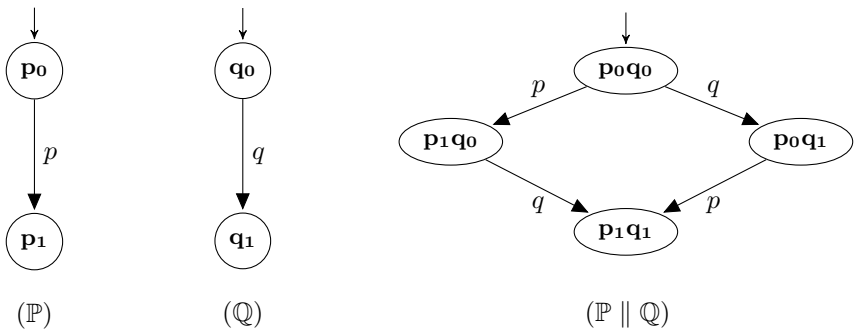


Figure 2.33 Asynchronous product of two transition systems.

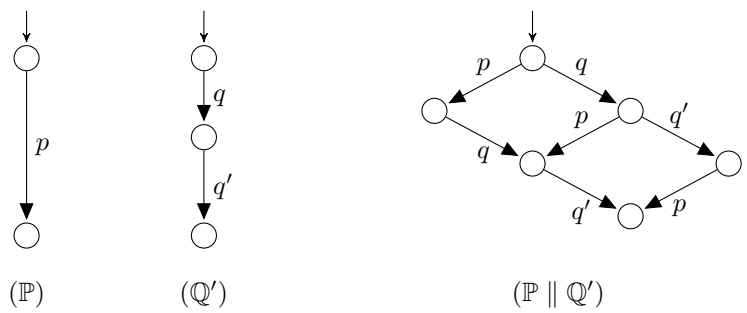


Figure 2.34 Impact of the granularity of atomic executions on the states.

where $\mathbb{T}$ is defined by

$$\frac{(s_1, \ell_1, s_1') \in \mathbb{T}_1, (s_2, \ell_2, s_2') \in \mathbb{T}_2,}{((s_1, s_2), \ell_1, (s_1', s_2)) \in \mathbb{T}, ((s_1, s_2), \ell_2, (s_1, s_2')) \in \mathbb{T}}.$$

A scheduler executing two asynchronous systems can stop them at any time between two *atomic actions* corresponding to one transition of the product. The pure asynchronous product produces the whole set of global states that are potentially reachable when two systems are executed by such a scheduler. Choosing the granularity of atomic transitions is an intrinsic modeling problem. The granularity should be faithful to reachable reality. Figure 2.34 illustrates the impact of atomicity of the behavior exposed by the asynchronous model.

2.2.2.2 Asynchronous Communication

Asynchronous modeling assumes that none of the components (processes, threads, state machines, etc.) of a system are active at the same time. Still they can exchange information using shared memory, or by means of messages. When shared memory is used the states of the global process can include the state of the memory locations that are relevant for the processes. The memory locations should be statically known. Communication by means of message passing can rely on usage of communication *channels* (e.g., FIFOs). When communication channels are used, the sending should be non-blocking: the component-sender can proceed without waiting for the component-receiver to accept the message. The global state of the model comprises states of all communication channels. Figure 2.35 shows an example of two asynchronous processes communicating through a FIFO channel. Notice, due to communication of the parallel components, some of the paths of the product are not possible.

2.3 Formal Specifications

2.3.1 Intuition

According to [MP91] a *specification* is a description of the desired behavior (*property*) or operation of the system, while avoiding references to the method or details of its implementation. For instance, in Figure 2.36 the SystemC implementation of the TLM virtual prototype has a natural language specification

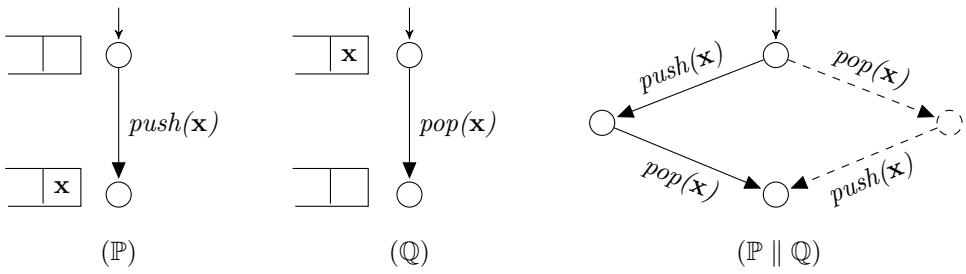


Figure 2.35 Asynchronous communication through a FIFO: dashed lines show the unfeasible path of the product.

defining the management of interrupts. A specification is typically more descriptive and less operational than the system's implementation. Using specification one can make sure that (i) the design (or code segment) functions are according to the specification, (ii) the specification is complete, (iii) the entire specification has been fully implemented. Formal specifications can be defined as follows. Let S be the implementation of the design, and let $\mathcal{C}(S)$ be a set of all *computations* (sequences) generated by S. Let ϕ be a specification property of the design S, and let $Sat(\phi)$ be the set of sequences that satisfy ϕ. Then, we say that S *implements* ϕ (or equivalently, S *satisfies* ϕ) if all its computations satisfy ϕ, i.e., $\mathcal{C}(S) \subseteq Sat(\phi)$.

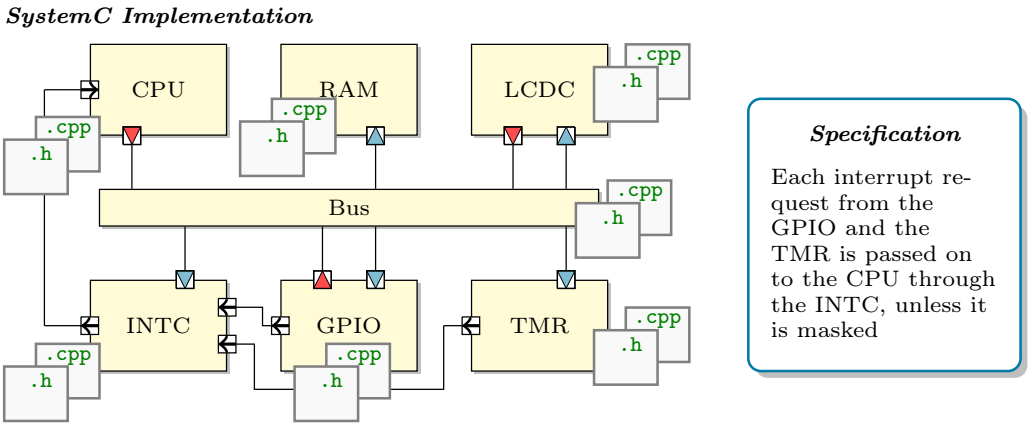


Figure 2.36 Example of a natural language specification of a TLM virtual prototype.

Properties can be defined either in plain English (like in Fig. 2.36), or by means of dedicated specification languages (e.g., PSL, SVA, see Sec. 2.3.2 below). A possible advantage of the latter is that they can provide a means to write specifications which can be easy to read and mathematically precise. Formal specification (i) are unambiguous, (ii) enable the use of formal methods for checking functional correctness of the design, (iii) can be used as a part of assertion-based verification frameworks as defined in Sec. 2.1.3. When used as a part of testbenches for hardware design, properties referred to as *assertions* are meant to be checked by the assertion checker (see Sec. 2.1.3).

2.3.2 Defining Properties (The Discrete-Time Case)

Since we deal with digital circuits, we will focus on discrete-time specification languages. There exist works on continuous-time specification logic for analog circuits [MN04; FP09; DFM13].

In this section we provide an overview of the basic *operators* used to define assertions of the hardware design. Some of those operators are implemented by industrial specification languages like *Property Specification Language* (PSL) [18505; EF06] and *SystemVerilog Assertion* (SVA) [ST12; SSF06]. Let V be a set of variables taking values in some domains (i.e., integers, Booleans, etc). Assertions are interpreted on *infinite sequences* (computations) of the design denoted here as $\sigma = \sigma^0 \dots \sigma^k \dots$, where σ^k represents the kth *valuation* of variables of V. We denote any finite subsequence of σ as $\sigma^{k \dots \ell}$, and the suffix (resp. prefix) of σ starting from (resp. ending by) the ℓth letter as $\sigma^{\ell \dots}$ (resp. $\sigma^{\dots \ell}$).

2.3.2.1 State Formulas

Properties are defined by means of *logical* and *temporal* operators, which are constructed from *state formulas* [MP91]. State formulas are defined over V using standard functions and predicates for integers (e.g., "+", "−"), Booleans (e.g., "∧", "∨"), etc. They can be combined using Boolean operators. For instance, given two integer variables $x, y \in V$ one may define a state formula $(x \leq 5) \wedge (y \neq x)$. A state formula can be evaluated at a certain position $k \geq 0$ in a sequence, and it expresses properties of the state σ^k occurring at this position. We use notation: a state formula ϕ holds at position k of σ if and only if $\sigma^k \models \phi$. For instance, in Figure 2.37 the state formula $(x \leq 5) \wedge (y \neq x)$ holds at positions 3, 5 and 7.

k	0	1	2	3	4	5	6	7	8	9	$\cdots$
x	**9**	**5**	**9**	**4**	**8**	**3**	**7**	**2**	**6**	**1**	$\cdots$
y	**9**	**5**	**1**	**5**	**5**	**1**	**10**	**5**	**1**	**1**	$\cdots$
$(x \leq 5)$	F	T	F	T	F	T	F	T	F	T	$\cdots$
$(y \neq x)$	F	F	T	T	T	T	T	T	T	F	$\cdots$
$(x \leq 5) \wedge (y \neq x)$	F	F	F	T	F	T	F	T	F	F	$\cdots$

Figure 2.37 Evaluation of state formulas.

A temporal formula is constructed from state formulas to which we apply *temporal operators*, *Boolean connectives* and *operators of extended regular expressions*. Their semantics is introduced below in respective sections.

2.3.2.2 Temporal Operators

The temporal operators presented here are operators of Linear Temporal Logic (LTL) as defined in [Pnu77; Kam68b]. In the sequel we denote the fact that a temporal formula ϕ holds at the kth position of a sequence σ by $\sigma^k \models \phi$.

The *Henceforth* Operator If ϕ is a temporal formula, then $\Box\phi$ (or $G\phi$), read as *henceforth* ϕ or *always* ϕ, is a temporal formula. Its semantics is

$$\sigma^k \models \Box\phi \iff \forall j \geq k : \sigma^j \models \phi.$$

The formula $\Box\phi$ holds at position k if and only if ϕ holds at position k and all following positions (e.g., see Fig. 2.38).

k	0	1	2	3	4	5	6	$\cdots$
x	**3**	**4**	**5**	**6**	**7**	**8**	**9**	$\cdots$
$(x > 5)$	F	F	F	T	T	T	T	$\cdots$
$\Box(x > 5)$	F	F	F	T	T	T	T	$\cdots$

Figure 2.38 Evaluation of the *henceforth* operator $\Box$. Values of x are increasing.

The *Eventually* Operator If ϕ is a temporal formula, then $\Diamond\phi$ (or $F\phi$), read as *eventually* ϕ, is a temporal formula. It holds at position k if and only if ϕ holds at some position $j \geq k$ (e.g., see Fig. 2.39):

$$\sigma^k \models \Diamond\phi \iff \exists j \geq k : \sigma^j \models \phi.$$

k	0	1	2	3	4	5	6	$\cdots$
x	**3**	**4**	**5**	**6**	**7**	**8**	**9**	$\cdots$
$(x = 5)$	F	F	T	F	F	F	F	$\cdots$
$\Diamond(x = 5)$	T	T	T	F	F	F	F	$\cdots$

Figure 2.39 Evaluation of the *eventually* operator $\Diamond$. Values of x are increasing.

The *Until* Operator The until formula $\phi\ \mathcal{U}\ \psi$ (read ϕ *until* ψ) predicts the eventual occurrence of ψ (similarly to $\Diamond\psi$) and states that ϕ holds continuously at least until the (first) occurrence of ψ (in the spirit of $\Box\phi$)[9]. The semantics is the following:

$$\sigma^k \models \phi\ \mathcal{U}\ \psi \iff \exists j \geq k : \sigma^j \models \psi\ \wedge\ \forall k \leq i < j : \sigma^i \models \phi.$$

Figure 2.40 illustrates the evaluation of the *until* formula. Notice, if position k satisfies formula ψ, it also satisfies $\phi\ \mathcal{U}\ \psi$ for any ϕ (even FALSE). Notice also that all positions satisfying $\phi\ \mathcal{U}\ \psi$ satisfy $\Diamond\psi$.

The *until* operator has a *weak* version $\phi\ \mathcal{W}\ \psi$ referred to as *week until*. It states that either ϕ always holds starting from the current position, or it holds at least until the (first) occurrence of ψ. It is defined as

$$\phi\ \mathcal{W}\ \psi \stackrel{\text{def}}{=} \Box\phi \vee \phi\ \mathcal{U}\ \psi.$$

k	0	1	2	3	4	5	6	$\cdots$
x	**4**	**5**	**5**	**5**	**6**	**6**	**7**	$\cdots$
$(x = 5)$	F	T	T	T	F	F	F	$\cdots$
$(x = 6)$	F	F	F	F	T	T	F	$\cdots$
$(x = 5)\ \mathcal{U}\ (x = 6)$	F	T	T	T	T	T	F	$\cdots$

Figure 2.40 Evaluation of the *until* operator $\mathcal{U}$.

The *Next* Operator If ϕ is a temporal formula, then $\bigcirc\phi$ (read as *next*) ϕ is a temporal formula. Its semantics is such that

$$\sigma^k \models \bigcirc\phi \iff \sigma^{k+1} \models \phi.$$

Thus, $\bigcirc\phi$ holds at position k if and only if ϕ holds at the next position $k+1$ (e.g., see Fig. 2.41).

k	0	1	2	3	4	5	6	$\cdots$
x	**5**	**4**	**3**	**5**	**4**	**3**	**5**	$\cdots$
$(x = 5)$	T	F	F	T	F	F	T	$\cdots$
$\bigcirc(x = 5)$	F	F	T	F	F	T	F	$\cdots$

Figure 2.41 Evaluation of the *next* operator $\bigcirc$. Values of x are periodic.

2.3.2.3 Boolean Operators (for Temporal Formulas)

Let ϕ and ψ be temporal formulas. They can be combined using Boolean operators of conjunction $\wedge$, disjunction $\vee$, implication $\rightarrow$, negation $\neg$, etc.

EXAMPLE 2.3.1. – Let $V = \{a, b\}$ be a vocabulary of temporal formulas ϕ, ψ and π. Let ϕ (resp. ψ) be such that $\phi = \Box(\neg(a \wedge b)) \wedge \Box(a \rightarrow \bigcirc(\neg a\ \mathcal{U}\ b))$ (resp. $\psi = \Box(\neg(a \wedge b)) \wedge \Box(b \rightarrow \bigcirc(\neg b\ \mathcal{U}\ a))$); it states that (i) a and b never occur simultaneously, (ii) a (resp. b) *never occurs two times without* b *(resp.* a*) in between*. Let π be such that $\pi = \Box(\neg(a \wedge b)) \wedge (\neg b\ \mathcal{U}\ a)$; it states that (i) a and b never occur simultaneously, (ii) a *occurs before* b. The conjunction $\phi \wedge \psi \wedge \pi$ defines alternation of a and b starting from a. $\Box$

2.3.2.4 Sequential Extended Regular Expression (SERE) Operators

Depending on the source the "SERE" acronym can be translated to different words. Thus, it stands for *"Sugar Extended Regular Expression"* in [Bee+01], *"Sequential Extended Regular Expression"* in [EF06; 18505], and *"Semi-Extended Regular Expression"* in [EF09]. In all these cases, the intent is to *extend* the usual operators of regular expressions (union "|", concatenation ";" and Kleene star "[*]") with additional operators such as *intersection*, etc. Here we provide semantics of SERE operators as it is defined in [18505]. The semantics is defined over *finite* sequences. The idea is to define a state formula ϕ which is true at one point, and then to combined state formulas using SERE operators:

[9] The until operator $\mathcal{U}$ can be used to derive the $\Diamond$ and $\Box$ operators; the reverse is not true [Kam68a]. Thus, $\Diamond\phi = \text{TRUE}\ \mathcal{U}\ \phi$ and $\Box\phi = \neg\Diamond\neg\phi$.

$$\sigma \models \{\phi\} \iff \sigma \models \phi$$
$$\sigma \models \phi \mid \psi \iff \sigma \models \phi \ \vee \ \sigma \models \psi \qquad (\textit{union})$$
$$\sigma \models \phi \,;\psi \iff \exists \sigma_1,\ \sigma_2 :\ \sigma = \sigma_1.\sigma_2 \text{ such that } \sigma_1 \models \phi \ \wedge \ \sigma_2 \models \psi \qquad (\textit{concatenation})$$
$$\sigma \models \phi[*] \iff \sigma = \epsilon \ \vee \ \exists \sigma_1,\ \sigma_2 :\ \sigma = \sigma_1.\sigma_2 \text{ such that } \sigma_1 \neq \epsilon \ \wedge \ \sigma_1 \models \phi \ \wedge \ \sigma_2 \models \phi[*] \qquad (\textit{Kleene star})$$
$$\sigma \models \phi \ \&\& \ \psi \iff \sigma \models \phi \ \wedge \ \sigma \models \psi \qquad (\textit{intersection})$$

Using these operators, one may define the following syntactic sugar:

$$\phi[*i] \stackrel{\text{def}}{=} \underbrace{\phi;\ \phi; \ldots ;\ \phi}_{i \text{ times}} \qquad (\textit{bounded ;-iteration})$$
$$\phi[*i..j] \stackrel{\text{def}}{=} \phi[*i] \mid \ \ldots \ \mid \phi[*j]$$

Example 2.3.2. – Consider a SERE $\{\text{TRUE}[*]; a; \text{TRUE}[*]; b; \text{TRUE}[*]\}[*]$. It defines a finite sequence such that (i) a and b alternate starting from a, (ii) between a and b anything else can occur. □

2.3.2.5 SERE-LTL Binding Operators

Let r be a SERE formula, and let ϕ be a temporal formula. They can be combined with the following suffix implication operator:

$$\sigma \models r \mapsto \phi \iff (\forall k \geq 0 : \sigma^{0\ldots k} \models r \rightarrow \sigma^{k\ldots} \models \phi) \qquad (\textit{suffix implication})$$

The formula $r \mapsto \phi$ states that ϕ should hold *whenever* r holds[10]. Based on suffix implication one may define the following syntactic sugar:

$$r \Mapsto \phi \stackrel{\text{def}}{=} \{r; \text{TRUE}\} \mapsto \phi \qquad (\textit{weak suffix implication})$$

The formula $r \Mapsto \phi$ states that ϕ should hold whenever r holds starting from the *next* instant.

Example 2.3.3. – Consider a formula $\{a;\ \text{TRUE}[*];\ b\} \mapsto \Box(c)$. Here, ";" is a regular expression concatenation. It states that if a occurs and then later b occurs, then c will always occur starting from the instant when b occurs. A formula $\{a;\ b\} \Mapsto \{c;\ d\}$ states that if a and b occur one after the other, c will occur at the next instant after b, and then d. □

2.3.3 Specification Languages for Hardware Designs

To define properties for hardware design (assertions), one may use either *chip design languages* (e.g., Verilog, SystemVerilog, VHDL, SystemC/C++), or *dedicated assertion languages*. The latter provide support for (a certain subset of) the operators listed in Sec. 2.3.2. Assertion languages include: (i) SystemVerilog Assertions (SVA) [ST12; SSF06], (ii) Property Specification Language (PSL) [18505; EF06], (iii) C-Asserts as a part of ANSI-C standard associated with SystemC/C++ design verification, (iv) proprietary assertions formats (e.g., Intel *ForSpec*, IBM *Sugar* languages).

SVA and PSL are IEEE standards. Both languages implement LTL, Boolean, SERE and SERE-LTL operators. The sequences on which the operators are evaluated are interpreted differently depending on the abstraction level of the hardware design. Thus, if the design is defined at synchronous RT level, each letter of a sequence σ corresponds to a *clock tick* and operators are evaluated on each clock tick. If the design is defined at asynchronous TLM level, operators are evaluated on sequences of *events*. SVA is used by about 70% of the industry to define assertions at RT level [Hog16]. PSL supports both *clocked* and *unclocked* versions of LTL and SERE operators, and it is suitable to define properties at the transaction level.

[10] We provide semantics of the formula $r \mapsto \phi$ as it is defined in PSL (see [18505]). In [DL16] its *existential* variation is proposed; it defines that r should hold *at least once*, and ϕ should hold afterwards.

PSL Operators	**Corresponding Operators of Sec. 2.3.2**
`always` f	$\Box\phi$
`eventually!` f	$\Diamond\phi$
f `until!` g	$\phi\,\mathcal{U}\,\psi$
f `until` g	$\phi\,\mathcal{W}\,\psi$
`next!` f	$\bigcirc\phi$
`next` f	$\neg\bigcirc\neg\phi$

Table 2.1 PSL syntax for temporal operators. f, g are PSL formulas.

Property Specification Language (PSL) In the whole document, PSL stands for PSL 1.1. PSL consists of four layers:

1. The *Boolean Layer* consists of Boolean expressions built with Boolean operators (conjunction "$\wedge$", disjunction "$\vee$", etc.).

2. The *Temporal Layer* is comprised of temporal expressions which describe the relationships between Boolean expressions over time. Temporal expressions are built by means of SERE, LTL and SERE-LTL operators.

3. The *Verification Layer* consists of directives which describe how the temporal expressions should be handled by the verification tool. For example, `assert` ϕ is a verification directive that tells the tools to verify that the property ϕ holds. Other verification directives include an instruction to assume, rather than verify, that a particular temporal property holds, or to specify coverage criteria for a simulation tool.

4. The *Modeling Layer* allows to model behavior of design inputs to name properties from the temporal layer.

PSL formulas can be constructed by means of LTL operators, Boolean, SERE and SERE-LTL operators. LTL based operators form so called *PSL Foundation Language*, and the respective formulas are usually referred to as *FL formulas*. Table 2.1 lists temporal operators supported by PSL as they are defined in [18505], and their expansion into corresponding LTL operators.

PSL can be evaluated on finite sequences even for the operators that look like liveness properties; it requires change of the semantics of the operators [DGV13]. For the LTL operators *next* $\bigcirc$ and *until* $\mathcal{U}$, PSL defines their *weak* versions. Thus, the PSL formula `next!` ϕ (resp. ϕ `until!` ψ) has the semantics of $\bigcirc\phi$ (resp. $\phi\,\mathcal{U}\,\psi$), and the PSL formula `next` ϕ (resp. ϕ `until` ψ) means that ϕ holds at the next instant only *if that instant exists* (resp. ϕ holds up until an instant where ψ holds *if such exists*). Intuitively, a property built on weak operators that is still pending at the end of a finite trace is considered satisfied.

2.4 Recognition of Regular Languages, Continuous Recognizers

A language L is *regular*, if it is defined by a regular expression. A *recognizer* of a regular language L is a deterministic finite automaton which accepts sequences of L [Hop+00]. For instance, Figure 2.42 shows the recognizer of a sequence 123. Here, in s_3, the recognizer accepts the sequence (the state s_3 is accepting). When the recognizer of L is started, it works till either it detects a sequence of L or fails; the recognizer cannot be re-started.

A *continuous recognizer* of a regular language L is a deterministic finite automaton, which works on *infinite sequences*. The continuous recognizer is always active. For each element of an infinite sequence, it can decide if L occurred at the end of the prefix of the sequence observed thus far, whatever happened before. After each occurrence of L the automaton continues. A door opening is an example of the work of the continuous recognizer. The door opens, whenever the combination of digits (e.g., 123) is entered, and it does not matter which digits were before. Figure 2.43 shows the continuous recognizer of the sequence 123. Here, s_3 is an accepting state.

The continuous recognizer of L should not miss any occurrence of L. If L occurs, at the next step the recognizer tries to detect another occurrence of L. If occurrences of L overlap, when one occurrence

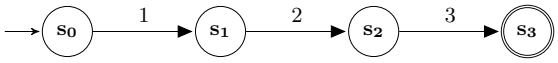


Figure 2.42 A recognizer of a sequence 123. A state denoted with an arrow (resp. double circles) is an initial state (resp. are accepting states). Transitions which are not defined are forbidden.

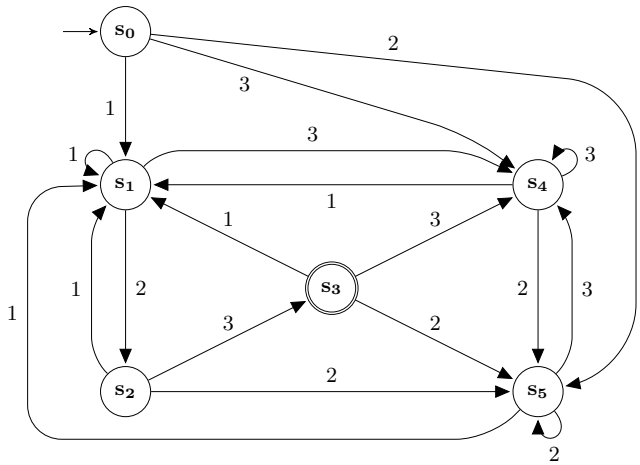


Figure 2.43 A continuous recognizer of a sequence 123.

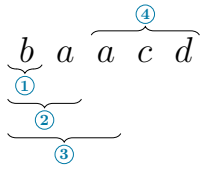


Figure 2.44 Overlapping occurrences of a regular language $ba^* + acd$ in a sequence $baacd$.

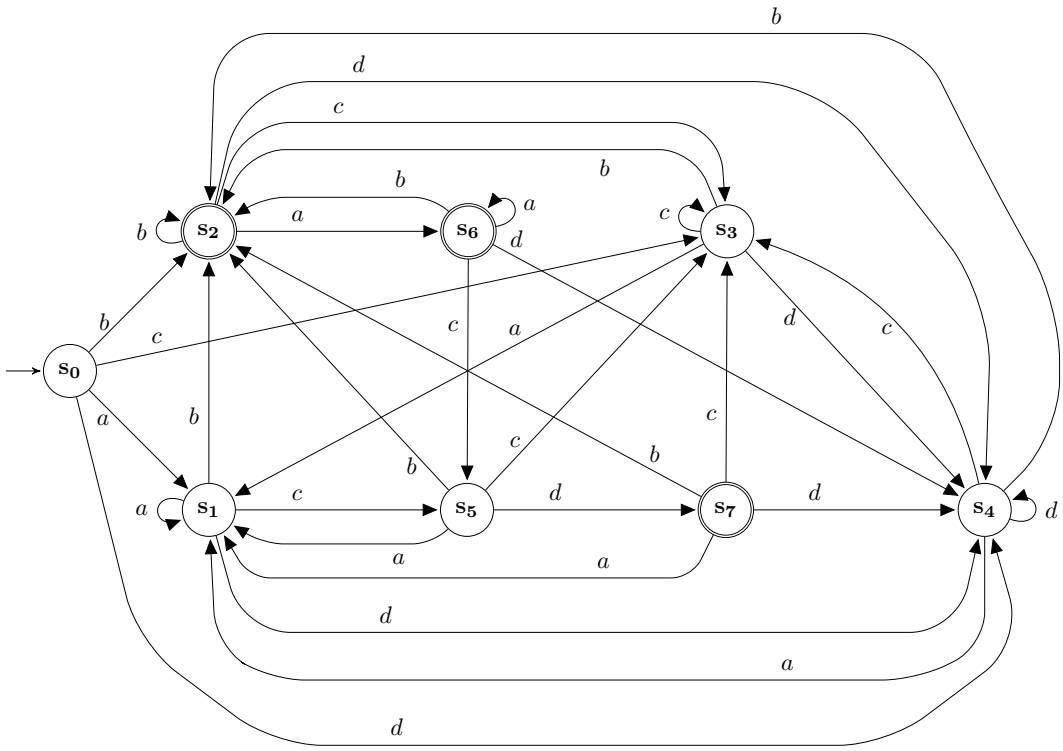


Figure 2.45 A continuous recognizer of a regular language $ba^* + acd$.

of L is detected, the continuous recognizer returns to a state, which corresponds to another occurring L. For instance, consider a language L defined by the regular expression $ba^* + acd$ (here, "a^*" is a Kleene star, and "+" is the union operator). Consider a sequence *baacd* (Fig. 2.44). The sequence has four overlapping occurrences of L: ①, ② and ③ correspond to ba^*, ④ is the sequence *acd*. Figure 2.45 shows the continuous recognizer of $ba^* + acd$. When ① occurs, the recognizer is in the accepting state s_2. When either ② or ③ occurs, the recognizer is in the accepting state s_6. After the occurrence of ③, when c takes place, the recognizer enters s_5. Then, when d happens, the recognizer moves to the accepting state s_7, and thus the occurrence of ④ is detected.

Notice, when the continuous recognizer is in an accepting state, it means that a sequence of its language has been detected *whatever happened before*. Thus, the continuous recognizer in Figure 2.45 is in the accepting state s_2 whenever a sequence of the form dbb^* occurs. This happens because one b is detected (i.e., a sequence of the language define by $ba^* + acd$), not because db.

In Chapter 4, we define continuous recognizers for loose-orderings. In Chapters 8 and 9, we refer to continuous recognizers, when we describe the implementability of our stubbing framework.

2.5 Components and Contracts

As complexity of software and hardware systems increases, and the development and verification time decreases it becomes unavoidable to *reuse* a lot of previous work when designing new systems. Reusing parts of a previous system requires that these parts be properly defined as *components* and equipped with some form of a *specification*. In the world of hardware IPs are typical components.

A *contract* is one possible form of a specification. A contract characterizes under which context a component is *assumed* to operate, and what a component *guarantees*, if it works in a proper context. Contract theories strictly follow the principle of *separation of concerns*: the specification of assumptions is separated from the specification of guarantees. For instance, Figure 2.46 illustrates a contract for a class computing a square root of a number. It is *assumed* that the actual number is set *before* a square root is computed, and that the number is non-negative. If the assumptions hold, the computation of a square root is *guaranteed* by the component.

Design-by-Contract The *"design-by-contract" principle* was first introduced in the Eiffel programming language by Bertrand Meyer [Mey92; Mey97]. The principle is based on the idea that the design of software modules should imperatively include the specification phase. The principle has been successfully applied to object-oriented programming along with the wide acceptance of pre-/post-condition style of specification (e.g., Eiffel [Swi93], iContracts [Kra98], JASS [Bar+01]). In the hardware world contracts are used for hardware optimization. Thus, by exploiting *(sequential) don't care* sets, it is possible to optimize area and performance of the chip [Dev91].

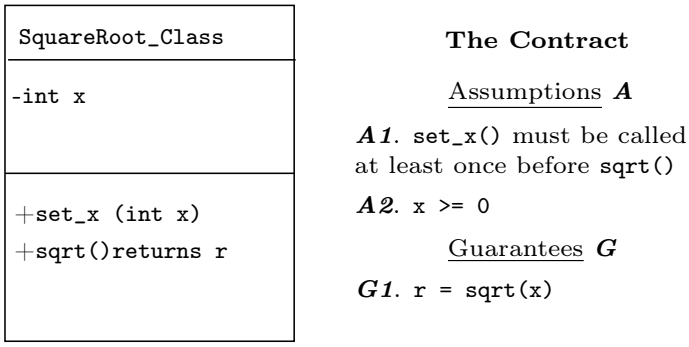


Figure 2.46 Contracts in object-oriented languages.

Part II

Running Example

Chapter 3

The Running Example

Contents

3.1 The External View and Functionality of the System **39**
3.2 SystemC/TLM Virtual Prototype **41**
3.2.1 TL Models of the Components 41
3.2.2 SystemC/TLM-2.0 Implementation of the Components 45
3.2.3 Communication Mechanism 47
3.3 The Embedded Software and the CPU **50**
3.3.1 The Control Algorithm 50
3.3.2 Interrupt Handling 50
3.3.3 Execution Modes 52
3.4 Formal Specification of Components **53**
3.4.1 The Embedded Software and the CPU 54
3.4.2 The Image Processing Unit (IPU) 57
3.4.3 The Liquid Crystal Display Controller (LCDC) 58
3.4.4 The Image Sensor (SEN) 59
3.4.5 Other Components 59
3.5 Synchronization Bugs **60**

The given chapter describes our running example which we call "Smart Intercom". It is a mixed component-based hardware and software system used for controlling accesses to a building based on the results of a face recognition analysis. A virtual prototype of the system has been implemented using standardized SystemC/TLM-2.0 [Sys]. It serves as a reference specification and illustration for the following parts of this thesis, as well as the basis for the experiments.

The material of this chapter is structured in the following way: In Section 3.1, we describe the high-level functionality of the device and its user interface. In Section 3.2, we present the TLM virtual prototype of the system, list details of the SystemC/TLM implementation, and give a rough idea about the behavior of the components constituting the designed SoC. In Section 3.3, the control performed by the software is described. Finally, in Section 3.5, we list synchronization bugs of the intercom system, and explain how they are detected at simulation time.

Section 3.4 is technical; it specifies the behavior of the system's components. Its understanding requires familiarization with Chapter 4. We suggest the reader to skip this section when reading the chapter for the first time. In the following parts of the document we refer to Section 3.4 when needed.

3.1 The External View and Functionality of the System

The user interface of the Smart Intercom device includes four elements: an image sensor (e.g., a digital camera), a liquid crystal display (LCDC) and two buttons (Fig 3.1). The Button-On-Off switches the system on and off. The Button-Start starts face recognition.

The functionality of the system consists of three phases (Fig. 3.2):

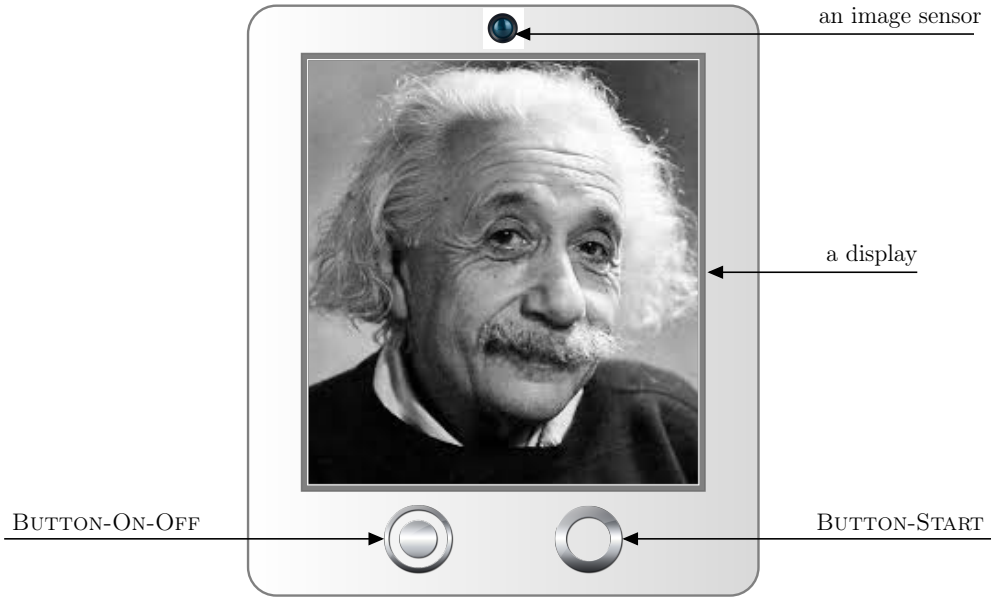


Figure 3.1 A sketch of the user interface of the Smart Intercom device.

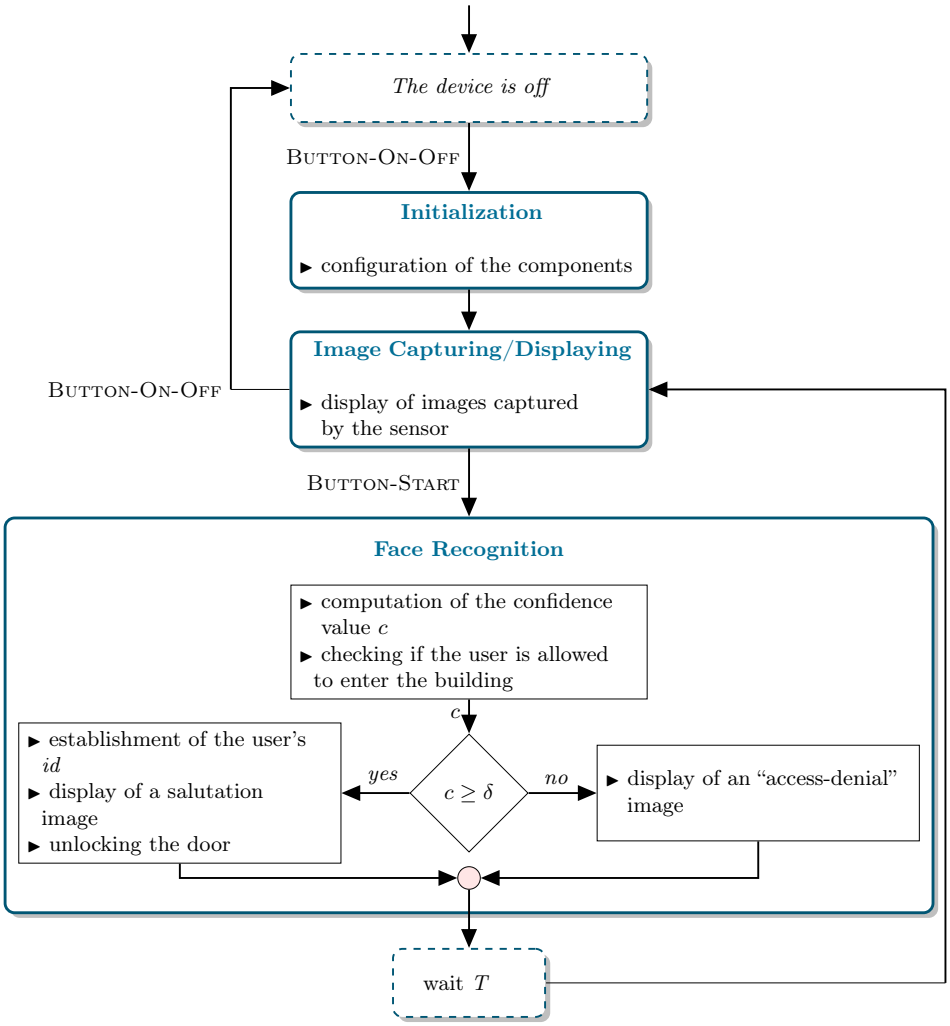


Figure 3.2 High-level functionality of the Smart Intercom.

1. The *initialization phase* starts when the user switches the device on by pressing BUTTON-ON-OFF; all subcomponents (e.g., the image sensor) are initialized. At any time the user can switch the device off by pressing again BUTTON-ON-OFF during the capturing/displaying phase.

2. During the *image capturing/displaying phase*, the images captured by the sensor are shown on the display.

3. The *face recognition* phase is started when the user presses BUTTON-START. During this phase the system computes a confidence value c of the face recognition, which is meant to be compared to a threshold δ. If $c < \delta$, then there is no significant match; it means that the user will see an "access-denial" notification on the screen. If $c \geq \delta$, then there is a significant match; in this case the system also computes an *id* of the person, which is used in the salutation message shown before the door opens. After some time T the system returns to the capturing/displaying phase.

It is assumed that all the users allowed to get access to the building are registered, i.e., the reference images of their faces are stored in the system's image database called *image gallery*.

3.2 SystemC/TLM Virtual Prototype

The designed intercom SoC is simple but representative; its representation is enough to illustrate our work. The hardware architecture of the SoC has common hardware components which can be used in real, industrial case-studies; the system is characterized by relatively complex communication protocols between hardware components. The SoC has the embedded software. The software ensures that the components, being executed together, implement the functionality of the system described in Section 3.1. We make the emphasis on the *synchronization* of the components; each component synchronizes with the rest of the system by exposing its *border behavior*, i.e., getting/producing transactions/interrupt requests (see more details below).

The TLM virtual prototype of the SoC's hardware platform is shown in Figure 3.3. It comprises one *processor node*, the set of *peripheral components*, one *hardware acceleration* block, the system's *memory*, and a memory-mapped *bus*. Specifically, Figure 3.3 shows the following components:

1. A central processing unit (CPU), on top of which the embedded software is executed (in the sequel we use "embedded software" and "CPU" interchangeably);
2. A timer to measure the elapsed time *T1* between the end of the face recognition phase and the beginning of the next capturing/displaying phase (TMR1);
3. A door lock actuator (LOCK);
4. A system's memory (MEM);
5. A timer controlled by the GPIO to measure a sampling period of keystrokes (TMR2);
6. A component to handle buttons (GPIO);
7. An image sensor (SEN);
8. A hardware accelerator called *image processing unit* (IPU), which performs a face recognition analysis;
9. A liquid crystal display controller (LCDC);
10. An interrupt controller (INTC);
11. A memory-mapped bus (Bus).

The components interact by means of transactions and interrupts; interrupts are modeled by means of SystemC signal channels. Through initiator (resp. target) TLM ports components send (resp. receive) transactions. The components send (resp. receive) interrupt requests by writing to SystemC signal channels (resp. being notified about the change of values of the channels). The architecture of the system is as it is shown in Figure 3.3.

In the following sections we describe in more details the TL models of the listed components, show an example of the SystemC/TLM implementation of one of the components, and explain a communication principle.

3.2.1 TL Models of the Components

TL models of the components can include the implementation of algorithms. Thus, the IPU implements the face recognition algorithm, the embedded software (the CPU) implements the control algorithm. The components can have data and control registers (Fig. 3.4). The registers are written (reps. read) by means

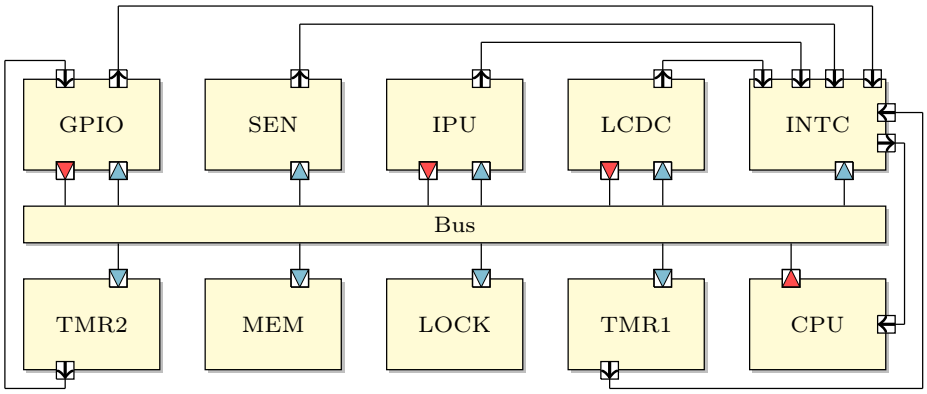


Figure 3.3 The TLM platform of the Smart Intercom device.

of write (resp. read) transactions. Writing the control registers triggers computation processes of the components (e.g., face recognition). To model duration of the computations, we use SystemC simulation time. This is specified with the *fuzzy* version of the `wait(TIME)` function introduced by [Cor08] (see more detail in Sec. 3.2.2.3).

Based on the ability to send and receive transactions, the TL components are classified as initiators, targets, or mixed initiator-targets. They are described below. Here, we focus only on the observed behavior of the components, i.e., sequences of received and sent transactions and interrupts. Formal specifications of the components is defined in Section 3.4.

3.2.1.1 The Initiator Component: The CPU

The CPU is a pure initiator component (Fig. 3.4(a)). It may receive interrupts from the interrupt controller INTC through its SystemC port *irq-cpu*. The embedded software running on the CPU may initiate transactions for other components of the system; those transactions are sent through the TLM initiator port of the CPU. The embedded software performs the control of all other components of the system. Its algorithm is discussed in Section 3.3.

3.2.1.2 Target Components

The timers TMR1 and TMR2, the memory MEM, the lock actuator LOCK, the interrupt controller INTC, and the image sensor SEN are targets, i.e., they can only receive and execute transactions.

The Timers The timers TMR1 and TMR2 are functionally identical (Fig. 3.4(c)). Both components can be (i) configured with a time interval by writing the read-write data register `scale`, (ii) started by writing the control register `start`, (iii) stopped by writing the control register `stop`. When the TMR1 (resp. the TMR2) is active, it sends interrupt requests through the port *irq-tmr1* (resp. *irq-tmr2*) whenever the defined interval of simulation time elapses.

The Memory The MEM stores the image gallery of the system, an image for face recognition analysis, the set of salutation images for the registered users, one "access-denial" image, and the double buffer of the LCDC. It is accessed by the CPU, the IPU, and the LCDC.

The Lock Actuator The LOCK component has one write-only control register `unlock` (Fig. 3.4(d)). When it is written, the actuator opens the entrance door.

The Interrupt Controller The INTC (Fig. 3.4(e)) has the set of SystemC ports through which the INTC receives interrupt requests from the processors. It gets a request from:

- the LCDC when the display is refreshed,
- the SEN when the image is captured,
- the GPIO when the button Button-Start is pressed,
- the IPU when started face recognition has finished,
- the TMR1 when time *T1* has elapsed.

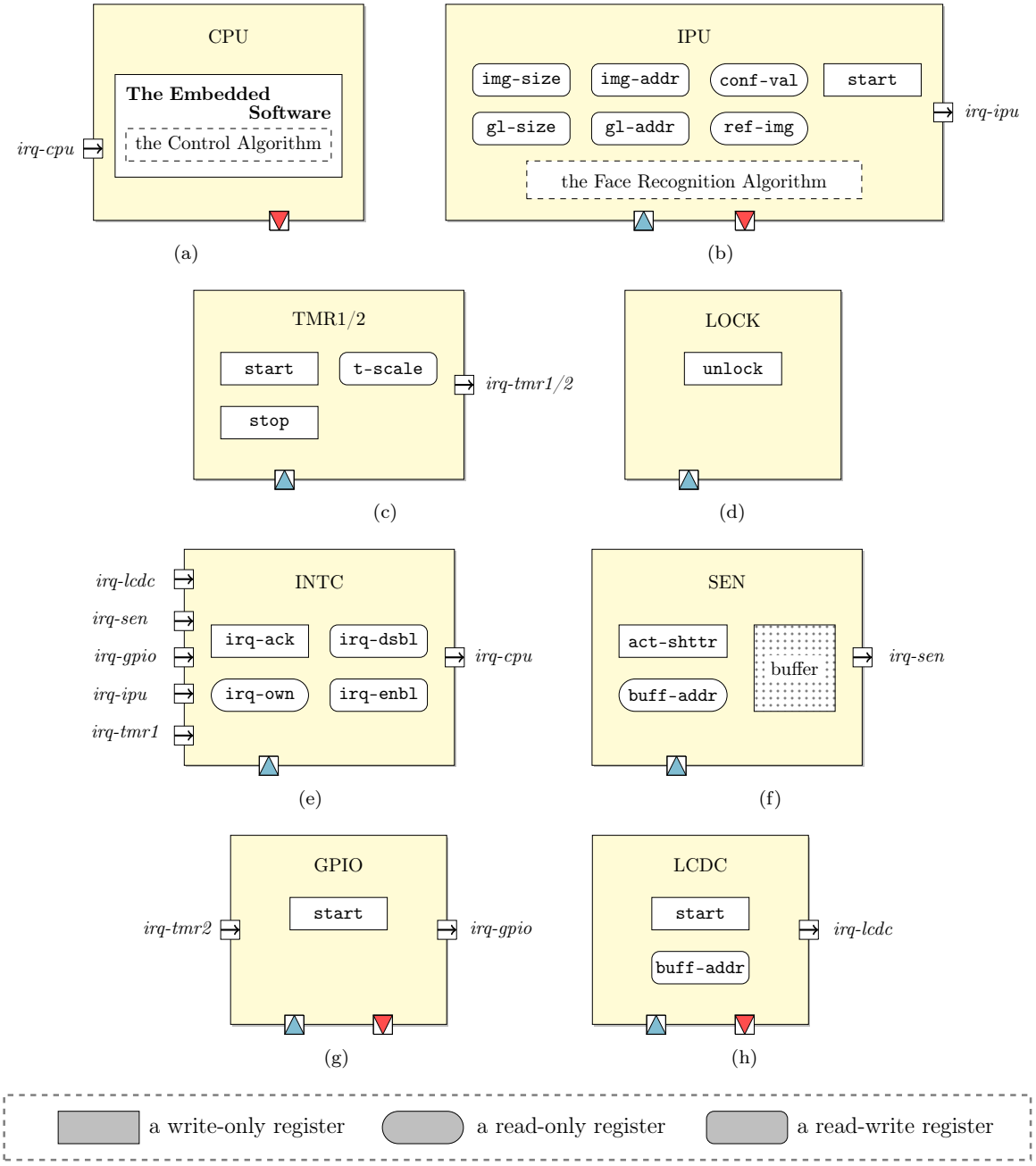


Figure 3.4 The TL models of the components constituting the intercom system.

The ports (interrupt inputs) are respectively *irq-lcdc*, *irq-sen*, *irq-gpio*, *irq-ipu*, *irq-tmr1* as represented in Figure 3.4(e). By generating interrupts through the interrupt output *irq-cpu*, the interrupt controller notifies the CPU about occurrences of one of the listed events. The CPU coordinates the work of the whole system by writing/reading a set of registers in the INTC:

- by writing the write-only register `irq-ack` the CPU acknowledges interrupt requests,
- by reading the read-only status register `irq-own`, the CPU establishes the owner of interrupts,
- the read-write registers `irq-dsbl/enbl` are used to mask (resp. unmask) interrupts.

The model of the INTC was inspired by the Xilinx OPB Interrupt Controller (v1.00c) [Xil]. Interrupt requests from different components are prioritized. The priority is assigned according to the order: the TMR1, the IPU, the GPIO, the SEN and the LCDC. Interrupt requests from the TMR1 (resp. the LCDC) have the highest (resp. lowest) priority.

The image sensor The SEN has an accessible buffer, where images captured by the component are stored (Fig. 3.4(f)). The address of the buffer is stored in the data read-only register `buff-addr`. To make the sensor to capture an image, one needs to activate its shutter by writing the control register `act-shttr`. The SEN sends interrupts through the SystemC port *irq-sen* when it captures an image.

To model the capturing of images by the sensor, we use the set of files-images. When the shutter of the SEN is activates, the component randomly picks up one of those files, and loads the file's content into its buffer.

3.2.1.3 Initiator-Target Components

There are tree mixed initiator-target components: the image processing unit IPU, the GPIO and the LCDC. They act as initiators when they configure and use other components:

- the IPU reads images from the memory MEM;
- the LCDC reads images from the buffer (stored in the memory MEM);
- the GPIO uses the timer TMR2.

The listed components are targets, when they are controlled by the CPU (the embedded software).

The Image Processing Unit Figure 3.4(b) shows the TL model of the IPU component. The read/write data registers `img-size`, `gl-size`, `img-addr`, `img-size` are used for the component's configuration; they store respectively values of the image size, the image gallery size, the address of the user's picture to be analyzed, and the size of the image. A confidence value computed during the face recognition by the IPU is stored in the read-only register `conf-val`. The IPU stores the address of the reference image corresponding to the best match to the read-only register `ref-img`. Writing the control register `start` launches the face recognition. During the recognition process, the IPU accesses the external image gallery and the image under analysis by sending transitions through its initiator socket. When recognition terminates, the IPU sends an interrupt through the port *irq-ipu*.

The face recognition procedure of the IPU is modeled with byte-by-byte comparison of images. The IPU reads the analyzed image from the memory, then reads the reference images from the image gallery. If one of the reference images matches the analyzed image, face recognition is successful. In this case, the IPU randomly chooses a confidence value in the interval $[70, 100]$. If the match of images was not established, the IPU stores zero value to the confidence value by writing the register `conf-val`. Notice that one could use the specialized face recognition libraries such as OpenCV [Opea], in order to implement the recognition analysis performed by the IPU. Nevertheless, even our simplistic implementation of the recognition is sufficient to observe synchronization bugs (see Sec. 3.5).

The General Purpose Input/Output The GPIO component can be launched by writing the control register `start` (Fig. 3.4(g)). The component configures and starts the TMR2. The component receives interrupts from the timer via the port *irq-tmr2*. It samples the state of Button-Start, when the timer's interrupt is received. If the button is pressed, the component sends an interrupt via its port *irq-gpio*. Button-Start is implemented by means of the SDL library [Sdl].

```
/*FILE: IPU.h*/

//use namespace tlm, sc_core
SC_MODULE(IPU){
  sc_out<bool>          irq_ipu; /*port*/
  initiator_socket<IPU> is;      /*sockets*/
  target_socket<IPU>    ts;

  SC_CTOR(IPU){
    SC_THREAD(face_recognition_thread);
    sensitive << start_event;
    irq_ipu.initialize(false);

    /*initialize registers*/
    img_addr = 0; gl_addr  = 0; gl_size  = 0; img_size = 0;
    conf_val = 0; }

  /*functions to access registers*/
  tlm_response_status read(uint32_t addr,  uint32_t &data);
  tlm_response_status write(uint32_t addr, uint32_t data);

private:
  sc_event start_event;

  /*behavior of the component*/
  void face_recognition_thread();

  /*registers*/
  uint32_t img_size, img_addr, gl_size, gl_addr, conf_val;

  /*face recognition procedures*/
  void upload_image(uint32_t addr, uint32_t* img_buffer);
  int  compare_facial_features();

  /*internal buffers*/
  uint32_t* img_buffer = NULL;
  uint32_t* gl_img_buffer = NULL; };
```

Figure 3.5 The declaration of the IPU.

The Liquid Crystal Display Controller The LCDC has an external double-buffer. The component has two registers: the read-write register `buff-addr` is used to set and get the address of the buffer, and the write-only control register `start` launches the component (Fig. 3.4(h)). When it is started, the LCDC reads the buffer and refreshes the display. The component generates interrupts through its interrupt output port *irq-lcdc* after each refresh cycle. The display is implemented by means of the SDL library.

3.2.2 SystemC/TLM-2.0 Implementation of the Components

We use a standard approach to implement TLM components in SystemC as it was defined in Section 2.1.5.3. For instance, Figure 3.5 illustrates the declaration of the component on the example of the IPU. Each component is defined as the SystemC module `sc_module`. If the component can send (resp. receive) interrupts, it possess output (resp. input) SystemC ports (e.g., line 5). If the component is an initiator (resp. a target), it has an initiator (resp. target) TLM port (e.g., lines 6, 7). Ports enable communication between components. The component can have data registers (e.g., line 29), and methods to access them (lines 19, 20). The behavior of the component is defined by SystemC processes (e.g., line 10). The processes are triggered when corresponding control registers are written (e.g., line 11). The component can have functions implementing specific algorithms (e.g., lines 32, 33).

3.2.2.1 Registers Access

To provide an access to read (resp. write) registers, the components implement a `read(const uint32_t addr, uint_32 data)` (resp. `write(const uint32_t addr, uint_32 data)`) method (e.g., see Fig. 3.6). The first argument defines the address (offset) of the register to be read (reps. written). Offsets of the registers are statically defined. The second argument is used to get the register's value (resp. to specify the value to be set to the register). The `read(...)` (resp. `write(...)`) function is called each time

```
/*FILE: IPU.cpp*/

//use namespace tlm, sc_core
//uint32_t conf_val, img_addr, gl_addr, img_size, gl_size

tlm_response_status IPU::read(uint32_t addr, uint32_t &data){
  /*identify the register*/
  switch(addr){
  case conf_val:  data = conf_val; break;
  case img_addr:  data = img_addr; break;
  case gl_addr:   data = gl_addr;  break;
  case img_size:  data = img_size; break;
  case gl_size:   data = gl_size;  break;
  default:
    SC_REPORT_ERROR(name(), "register is not implemented");
    return TLM_ADDRESS_ERROR_RESPONSE; }
return TLM_OK_RESPONSE; }
```

(a) Reading registers of the IPU

```
1  /*FILE: IPU.cpp*/
2
3  //use namespace tlm, sc_core
4  //uint32_t img_addr, gl_addr, gl_size, img_size
5  //sc_event start_event
6
7  tlm_response_status IPU::write(uint32_t addr, uint32_t data){
8    /*identify the register*/
9    switch(addr){
10   case img_addr: img_addr = data;      break;
11   case gl_addr:  gl_addr = data;       break;
12   case img_size: img_size = data;      break;
13   case gl_size:  gl_size = data;       break;
14   case start:    start_event.notify(); break;
15   default:
16     SC_REPORT_ERROR(name(), "register is not implemented");
17     return TLM_ADDRESS_ERROR_RESPONSE;  }
18   return TLM_OK_RESPONSE; }
```

(b) Writing registers of the IPU

Figure 3.6 Access to the IPU's registers.

the component receives a read (resp. write) transaction (see Sec. 3.2.3 below). Invocation of the method `write(const uint32_t addr, uint_32 data)`, such that `addr` is an address of a control register, can notify an event which triggers a computation process of the component (e.g., line 14 in Fig. 3.6(b)).

3.2.2.2 Behavior of the Component

The behavior of the components is modeled by means of the SystemC `SC_THREAD`s[1]. For instance, Figure 3.7 shows the `SC_THREAD` of the IPU. `SC_THREAD`s are sensitive to events: when write-only control registers are written by means of the `write(...)` function, events triggering the `SC_THREAD`s are notified. Thus, the `face_recognition_thread` of the IPU shown in Figure 3.7 is triggered when the event `start_event` occurs (line 4). The simulation processes of the components can:

- perform internal calculations of the components, call the components' functions (e.g., lines 7 – 33),
- send transactions through the initiator socket (e.g., line 13),
- let simulation time pass (e.g., lines 34, 38),
- send interrupts through ports (e.g., lines 37, 39).

3.2.2.3 Timing

To integrate the intrinsically timed components (which are the timers TMR1, TMR2, and the LCDC) with untimed components, we follow the guidelines given in [Cor08] and use *fuzzy timing* with randomization, also known as *loose timing*. We specify non-deterministic delays using the construct `pv_wait(a, b,`

[1] The SystemC threads are defined in Section 2.1.4.3.

```
/*FILE: IPU.cpp*/
void IPU::face_recognition_thread(){
  while(true){
  wait(start_event);
    conf_val = 0;

    /*initialize buffers*/
    img_buffer     = new uint32_t[img_size/sizeof(uint32_t)];
    gl_img_buffer  = new uint32_t[img_size/sizeof(uint32_t)];

    /*upload the analyzed image*/
    for(int i=0; i<img_size/sizeof(uint32_t); i++)
      is.read(img_addr+(i*sizeof(uint32_t)), img_buffer[i]);

    uint32_t temp_conf_val = 0;
    uint32_t gl_img_addr;
    int i = 0;
    while ( i < gl_size ){
      gl_img_addr = gl_addr + i * img_size;
      upload_image(gl_img_addr, gl_img_buffer);
      temp_conf_val = compare_facial_features();
      if(temp_conf_val > conf_val)
        conf_val = temp_conf_val;
      i += 1; }

    srand(time(0));
    /* if the correspondence was found then the value is
       taken from the range [70 .. 100];*/
    if(conf_val == 1)
      conf_val = rand()%30 + 70;
    /* otherwise from [0 .. 70)*/
    else
      conf_val = rand()%70;
    pv_wait(100, 500, SC_NS);
    delete [] img_buffer;
    delete [] gl_img_buffer;
    irq_ipu.write(true);
    pw_wait(1, 5, SC_NS);
    irq_ipu.write(false);
  }}
```

Figure 3.7 The behavior of the IPU.

`TIME_SCALE)` (Fig. 3.8). The simulation engine draws a random value in the interval $[a, b]$, thus exploring more behaviors. It allows us:

- to define explicit yielding points which let the components to advance,
- to avoid synchronization on timing, which may have spurious effect on the activities relying on TL models.

The `pv_wait(...)` is also used to model non-deterministic duration of a component's activity. For instance, it is specified that the face recognition may take from 100 up to 500 nanoseconds (see line 34 in Fig. 3.7).

```
//use namespace std, sc_core

#define pv_wait(a, b, TIME_SCALE) wait(a+(rand()%b), TIME_SCALE)
```

Figure 3.8 The definition of the `pv_wait()` function.

3.2.3 Communication Mechanism

The communication between components is *blocking*, i.e., each transaction is processed within one functional call (more details in Sec. 2.1.5). The initiator process is blocked until the communication finishes,

i.e., until the implementation of the function in the target returns. It is implemented by means of the blocking TLM transport interface[2].

3.2.3.1 TLM Ports

To enable communication between components by means of transactions, we define two classes `initiator_socket<...>` and `target_socket<...>` implementing respectively initiator and target sockets. They are derived from the standard TLM sockets `tlm_initiator_socket<...>` and `tlm_target_socket<...>` respectively (see Fig. 3.9). Both classes are parameterized with the type `M` of a component possessing a socket; in our case the type is always `sc_module`. Instances of the classes have pointers to their parent components.

The `initiator_socket` provides two methods `write(const uint32_taddr, uint32_t& data)` and `read(const uint32_taddr, uint32_t& data)` (Fig. 3.9(a)). They are used to send transactions through the initiator socket. When the component invokes one of these methods, it specifies:

- the destination address of a transaction (the first `addr` argument),
- the data to be sent or received by means of the transaction (the second `data` argument).

When invoked, the methods `write(...)` and `read(...)` create a transaction object using the specified address and the data, and send the created transaction to the connected target socket.

The `target_socket` implements the blocking transport interface represented by the function `b_transport(tlm_generic_payload tr, sc_time& t)` (Fig. 3.9(b)). When this function is called, the address and the data are recovered from the received transaction object `tr`. Then, depending on the type of the transaction (read or write), either `write(...)` or `read(...)` method of the socket's parent component is called. (Notice that the second argument `sc_time& t` is never used, since in our example we always assume that transactions are fulfilled immediately without delays).

3.2.3.2 The Interconnect: The Bus

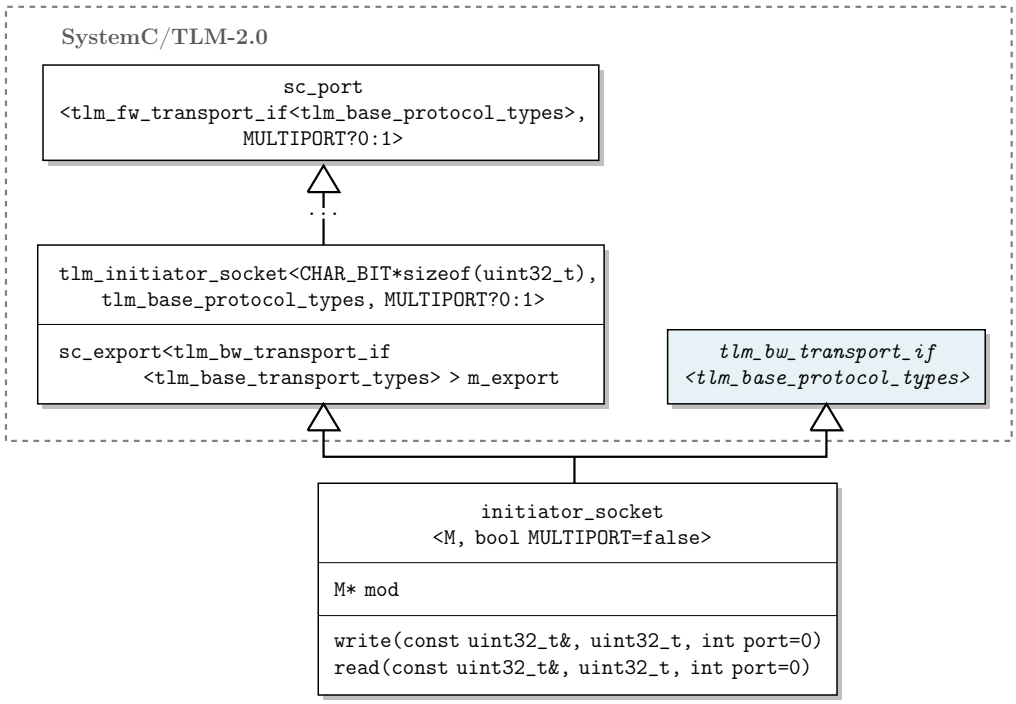
The components of the virtual prototype communicate through the memory-mapped Bus. The Bus maintains the correspondence between addresses and the components. For instance, to read the content of the memory, the IPU sends a read transaction *tr*, specifying an address. When the transaction *tr* reaches the Bus, the component retrieves the address and checks to which component it belongs. After the identity of the target is established (in this case the MEM component), the Bus passes on the transaction *tr* to the destination.

The interconnect Bus has the standard `tlm_target_socket<...>` and the `tlm_initiator_socket<...>`. The Bus is a communication channel, it implements the `b_transport(_generic_payload tr, sc_time& t)` method of the TLM blocking transport interface. All initiator components being connected to the target socket of the Bus can call the function `b_transport(tlm_generic_payload tr, sc_time& t)`. In the function, the Bus retrieves the address from the transaction `tr`, checks the mapping of addresses of the components, and passes on the transaction `tr` to the appropriate destination through its initiator socket.

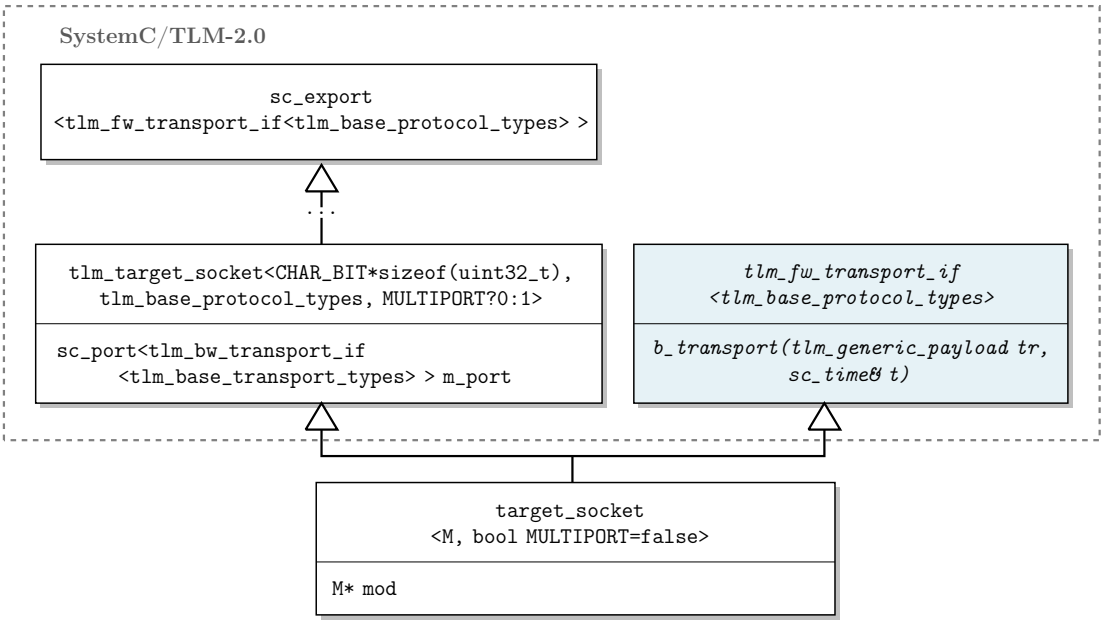
3.2.3.3 Communication Principle

Figure 3.10 summarizes the sequence of function calls performed to send a read transaction. The initiator invokes the `read(...)` method of its socket providing the address and the data to be sent. This is ①. The initiator socket creates a transaction and sends it calling the `b_transport(...)` method of the connect target socket of the Bus. This is ②. Since the Bus is the communication channel and its target socket is an export, the method `b_transport(...)` implemented by the Bus is called. The Bus establishes the destination component and calls the `b_transport(...)` function of the component's target socket connected to the initiator socket of the Bus. This is ③. The `b_transport(...)` of the target socket recovers parameters of the received transaction and calls the `read(...)` of the parent component. This is ④. The sequence ①-④ is the forward path, the sequence ⑤-⑧ is the return path. When the initiator component gets the data, i.e., the call of the `read(...)` function of its initiator socket returns, the component may proceed in its execution.

[2]The TLM communication principles via TLM ports is explained in Section 2.1.5.2.



(a) An initiator socket



(b) A target socket

Figure 3.9 The definition of TLM sockets.

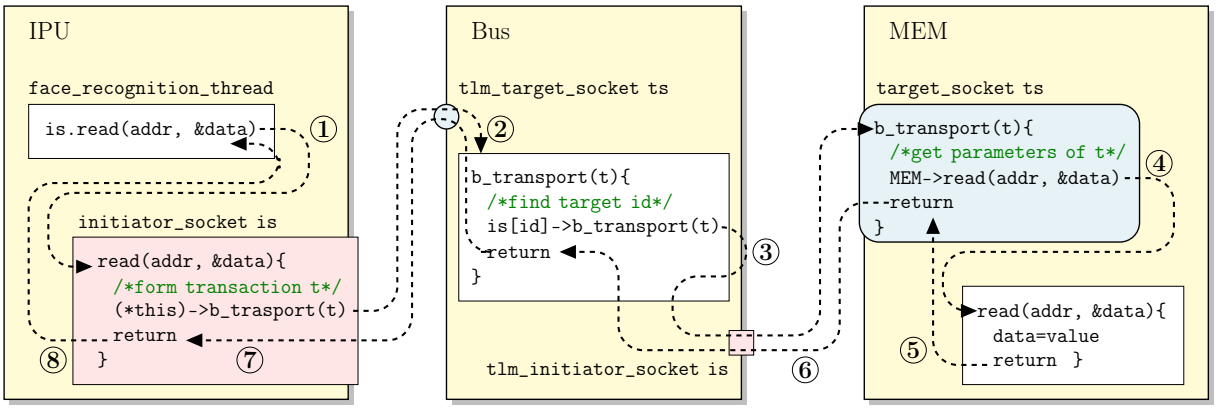


Figure 3.10 The communication principle.

3.3 The Embedded Software and the CPU

The embedded software running on the CPU coordinates execution of the processors: the LCDC, the SEN, the GPIO, the IPU and the TMR1 (Fig. 3.3). The CPU represented by its port *irq-cpu* and the initiator socket implements the hardware. The embedded software is executed on top of the CPU by means of the *native wrapper*:

- the `read(...)` (resp. `write(...)`) operations performed by the embedded software are translated into the calls of the `read(...)` (resp. `write(...)`) method of the initiator socket of the CPU;
- the interrupts gotten by the CPU invoke the interrupt handling methods `interrupt_pos_edge_handler()` and `interrupt_neg_edge_handler()` of the embedded software (see below).

In the sequel, we always consider the CPU *and* the embedded software, and use their names interchangeably. The TL model of the CPU together with the embedded software comprise ≈280 lines of C++ code.

3.3.1 The Control Algorithm

The control implemented by the CPU consists of four modes (states): *initialization*, *image capturing/displaying*, *face recognition*, *salutation*, *restart*. They match the execution phases of the system (see Sec. 3.1). The transitions between modes are defined by the types of interrupts the CPU gets. Figure 3.11 illustrates the control as a state machine. Here, states are the execution modes of the CPU. Labels IRQ_LCDC, IRQ_SEN, IRQ_GPIO, IRQ_IPU and IRQ_TMR1 represent occurrences of interrupts belonging to the LCDC, the SEN, the GPIO, the IPU and the TMR1 respectively. For each state the outgoing transitions represent all possible interrupts that may occur in that state. For instance, in the initialization mode the CPU can get the interrupt from the SEN, but not from the IPU. The CPU can mask some interrupts. For instance, the IRQ_GPIO cannot occur when the CPU is in the face recognition mode, because the CPU has masked it before entering that mode (see below). The occurrences of interrupts can be indirectly triggered by the behavior of the CPU. For instance, the IRQ_IPU (resp. IRQ_SEN) occurs when the IPU finishes face recognition (resp. the SEN captures an image) which is started by the CPU in its face recognition mode (resp. initialization, image capturing/displaying or restart mode). In the following sections, we first describe the interrupt handling principle of the embedded software, then we define the behavior of the CPU in each of its execution modes.

3.3.2 Interrupt Handling

To handle incoming interrupts, the embedded software has two functions `interrupt_pos_edge_handler()` and `interrupt_neg_edge_handler()` shown in Figures 3.12(a) and 3.12(b) respectively. The former (resp. the latter) is called each time the CPU gets a positive (resp. negative) edge of an interrupt signal trough its port *irq-cpu*.

When a positive edge is detected, the embedded software establishes the owner of the interrupt by reading the `irq-own` register of the interrupt controller (lines 6, 7 in Fig. 3.12(a)). Then the CPU acknowledges the interrupt by writing the `irq-ack` register of the INTC (lines 11, 14, 19, 22, 28). If the

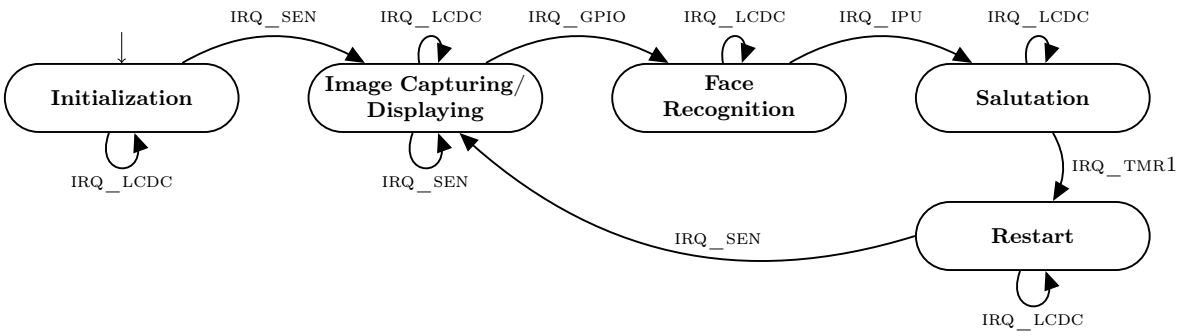


Figure 3.11 The control algorithm implemented by the CPU.

```
1  /*FILE: software.cpp*/
2
3  //uint32_t ack
4  void interrupt_pos_edge_handler(){
5    /*establish the owner*/
6    uint32_t irq_owner;
7    read(INTC_BASEADDR+IRQ_OWN, irq_owner);
8    ack = irq_owner;
9    switch(irq_owner){
10   case 4: //LCDC
11     write(INTC_BASEADDR+IRQ_ACK, 4);
12     break;
13   case 3: //SEN
14     write(INTC_BASEADDR+IRQ_ACK, 3);
15     break;
16   case 2: //GPIO
17     write(INTC_BASEADDR+IRQ_DSBL, 2); //mask the GPIO's interrupts
18     write(INTC_BASEADDR+IRQ_DSBL, 3); //mask the SEN's interrupts
19     write(INTC_BASEADDR+IRQ_ACK, 2);
20     break;
21   case 1: //IPU
22     write(INTC_BASEADDR+IRQ_ACK, 1);
23     break;
24   case 0: //TMR1
25     write(TMR1_BASEADDR+TMR_STOP, 0);  //stop the TMR1
26     write(INTC_BASEADDR+INTC_ENBL, 2); //enable the GPIO's interrupts
27     write(INTC_BASEADDR+INTC_ENBL, 3); //enable the SEN's interrupts
28     write(INTC_BASEADDR+IRQ_ACK, 0);
29     break;
30   default: cout << "ERROR: unknown source of the interrupt\n";
31     exit(1);
32   }}
```

(a) Handling the positive edge of interrupts

```
1  /*FILE: software.cpp*/
2
3  //uint32_t ack
4  void interrupt_neg_edge_handler(){
5    switch(ack){
6    case 4: /*IRQ_LCDC*/                      break;
7    case 3: /*IRQ_SEN */ image_capturing_displaying(); break;
8    case 2: /*IRQ_GPIO*/ face_recognition(); break;
9    case 1: /*IRQ_IPU */ salutation();        break;
10   case 0: /*IRQ_TMR1*/ restart();           break;
11   default: cout << "ERROR: unknown source of the interrupt\n";
12     exit(1);
13   }}
```

(b) Handling the negative edge of interrupts

Figure 3.12 The interrupt handling methods of the embedded software.

interrupt was received from the GPIO (i.e., BUTTON-START has been pressed), the CPU masks interrupts of both the GPIO and the SEN by writing the register `irq-dsbl` of the INTC (lines 17, 18). Masking

of interrupts guarantees that the started face recognition cannot be restarted unless it terminates, i.e., the presses of BUTTON-START are ignored. When the CPU gets an interrupt from the TMR1 (i.e., face recognition terminated and time T has elapsed), the CPU stops the timer, and unmasks the interrupts from the GPIO and the SEN (lines 26, 27).

When the method `interrupt_neg_edge_handler()` is called, the CPU is notified about the fact that the previously sent acknowledgement has had effect. Depending on the owner of the acknowledged interrupt, the CPU enters one of its execution modes (lines 7 – 10 in Fig. 3.12(b)).

3.3.3 Execution Modes

3.3.3.1 Initialization

Figure 3.13 provides the operations performed by the CPU in the initialization mode. When the system is started and the CPU is launched, the embedded software does the following:

- it configures the timer TMR1 and the LCDC setting respectively the time interval and the address of the buffer (lines 7, 8);
- the CPU reads the buffer address of the SEN (line 9);
- the CPU activates the shutter of the image sensor, starts the LCDL, and then the GPIO (lines 10 – 12).

The initialization should take time in the interval $[50, 200]$ nanoseconds (line 13 in Fig. 3.13).

```
1  /*FILE: software.cpp*/
2
3  //uint32_t sen_buf_addr, old_img_addr, data
4  void initialization(){
5    uint32_t t_scale;
6    t_scale = 1 + (rand() % 100);
7    write(TMR1_BASEADDR+TMR_T_SCALE,     t_scale);
8    write(LCDC_BASEADDR+LCDC_BUFF_ADDR, old_img_addr);
9    read (SEN_BASEADDR +SEN_BUF_ADDR,    sen_buf_addr);
10   write(SEN_BASEADDR +SEN_ACT_SHTTR,   data);
11   write(LCDC_BASEADDR+LCDC_START,      data);
12   write(GPIO_BASEADDR+GPIO_START,      data);
13   pv_wait(50, 200, SC_NS); }
```

Figure 3.13 The initialization mode.

3.3.3.2 Image Capturing/Displaying

Figure 3.14 shows the image capturing/displaying mode of the CPU. The component does the following:

- it switches to the inactive buffer of the LCDC (see lines 6 – 8);
- the component reads the content of the sensor's buffer (lines 9, 10);
- the CPU writes to the LCDC's buffer the image captured by the SEN (lines 11, 12);
- it activates the inactive buffer by setting the new buffer address of the LCDC (line 13);
- it activates the shutter of the SEN to make the component capture another image (line 14).

It takes the CPU at most 400 nanoseconds to perform the transfer of images (line 15).

3.3.3.3 Face Recognition

When the IRQ_GPIO occurs, the embedded software enters the face recognition mode (Fig. 3.15) and does the following:

- transfers the last captured image of the image sensor to the memory MEM (lines 6 – 9);
- configures the IPU providing the address of the image to be analyzed, the address of the image gallery, the size of one image in bytes, and the size of the image gallery (lines 10 – 13);
- finally, the CPU starts face recognition (line 14).

The CPU completes all mentioned activities within at most 400 nanoseconds (line 15).

```
/*FILE: software.cpp*/

//uint32_t sen_buf_addr, old_img_addr, new_img_addr, data
//uint32_t* img_buffer
void image_capturing_displaying(){
  uint32_t temp_addr = old_img_addr;
  old_img_addr = new_img_addr;
  new_img_addr = temp_addr;
  for(int i = 0; i < SEN_BUFFER_SIZE/sizeof(uint32_t); i++ )
    read(SEN_BASEADDR+sen_buf_addr+(i*sizeof(uint32_t)), img_buffer[i]);
  for(int i = 0; i < SEN_BUFFER_SIZE/sizeof(uint32_t); i++ )
    write(old_img_addr + (i*sizeof(uint32_t)), img_buffer[i]);
  write(LCDC_BASEADDR+ LCDC_BUFF_ADDR, old_img_addr);
  write(SEN_BASEADDR + SEN_ACT_SHTTR,  data);
  pv_wait(100, 400, SC_NS); }
```

Figure 3.14 The image capturing/displaying mode.

```
/*FILE: software.cpp*/

//uint32_t sen_buf_addr, img_addr, gl_addr, data
//uint32_t* img_buffer
void face_recognition(){
  for(int i = 0; i < SEN_BUFFER_SIZE/sizeof(uint32_t); i++ )
    read(SEN_BASEADDR+sen_buf_addr+(i*sizeof(uint32_t)), img_buffer[i]);
  for(int i = 0; i < SEN_BUFFER_SIZE/sizeof(uint32_t); i++ )
    write(img_addr + (i*sizeof(uint32_t)), img_buffer[i]);
  write(IPU_BASEADDR+IPU_IMG_ADDR,  img_addr);
  write(IPU_BASEADDR+IPU_GL_ADDR,   gl_addr);
  write(IPU_BASEADDR+IPU_IMG_SIZE,  IMG_SIZE);
  write(IPU_BASEADDR+IPU_GL_SIZE,   GALLERY_SIZE);
  write(IPU_BASEADDR+IPU_START, data);
  pv_wait(100, 400, SC_NS); }
```

Figure 3.15 The face recognition mode.

3.3.3.4 Salutation

When face recognition is finished and the IRQ_IPU occurs, the CPU enters the salutation mode. Here, it gets the confidence value c computed by the IPU. If c is smaller than the threshold δ, i.e., face recognition did not detect significant match, the CPU writes to the inactive buffer of the LCDC the "access-denial" image from the MEM. If the confidence value c is greater than the threshold $c \geq \delta$, face recognition terminates with a significant match. In this case the CPU establishes the *id* of the user, and writes a salutation image for the user to the inactive buffer of the LCDC. Then, the CPU switches the buffer address of the LCDC, (potentially) unlocks the door, and starts the timer TMR1. The salutation may take up to 400 nanoseconds.

3.3.3.5 Restart

When time T elapses and the IRQ_TMR1 occurs, the CPU immediately activates the shutter of the image sensor SEN. Later on, when the SEN captures an image and the IRQ_SEN occurs, the CPU again enters the image capturing/receiving mode.

3.4 Formal Specification of Components

This section provides the list of properties which specify the components of the running example. The emphasis is made on the properties which can be expressed by means of the loose-ordering language. The syntax and semantics of the language are defined in Chapters 4 and 5. It is assumed that the reader is familiar with the respective material. We do not consider properties which are beyond the expressivity of loose-orderings. They can be defined by means of other specification formalisms. For instance, one may define behavior of a fully deterministic component (like the INTC) as an imperative program. The properties defined here are scattered over the present document: they serve as examples and as the intercom's reference specification for the first and second parts of this thesis.

Recall, the loose-ordering properties of a component C are formulated in terms of the component's *inputs* and *outputs*. Input is any activity of the system's components that may affect C (e.g., getting an interrupt). Output is any activity of C which is visible for other components (e.g., acknowledgement of the received interrupt). We propose two types of loose-ordering properties: an *antecedent requirements* and a *timed implication constraints*. The former defines what the component *assumes* (expects) about its environment; the latter states what the component *guarantees* if it is properly used (i.e., its assumptions are fulfilled).

In the following sections we provide specification of the CPU, the IPU, the LCDC and the SEN. Other components are either trivial (e.g., the memory MEM and the lock actuator LOCK), or their specification cannot be expressed in terms of the loose-ordering language (e.g., the timers TMR1 and TMR2, the GPIO, the INTC). We comment the behavior of the latter ones in Section 3.4.5. There we also discuss the specification of the fully deterministic Bus. For each considered component we provide the set of its inputs and outputs and explain their meaning. Then, we formulate loose-ordering properties in terms of those inputs and outputs.

3.4.1 The Embedded Software and the CPU

Table 3.1 summarizes the set of inputs and outputs of the CPU. The left column of the table lists the interface names, the right column explains the purpose of each input/output. The input *start* represents the launching of the CPU; its occurrence coincides with the start of the whole system (start of a SystemC simulation). We define the set of the component's outputs applying the following naming conventions:

1. *set-Y-X* (resp. *get-Y-X*) output corresponds to writing (resp. reading) a *data register* Y of a component X (e.g., the outputs *set-img-add-IPU* and *get-conf-val-IPU*);
2. *Y-X* output corresponds to activation of a control register Y of a component X (e.g., the outputs *start-IPU* and *unlock-LOCK*);
3. *read-buff-X* (resp. *write-buff-X*) output represents the reading (resp. writing) of the buffer belonging to a component X (e.g., the outputs *read-buff-SEN* and *write-buff-LCDC*);
4. *read-MEM* (resp. *write-MEM*) output corresponds to reading (resp. writing) the system's memory *MEM*.

Some outputs (e.g., *set-buff-addr-LCDC*) have several numbered instances. Different instances correspond to different timed implication constrains.

3.4.1.1 Antecedent Requirements

The antecedent requirements A1-CPU and A2-CPU state that the CPU (its embedded software) can get interrupts (positive and negative edges) only if it has been started (X stands for *lcdc*, *sen*, *gpio*, *ipu* and *tmr1*). The assumption A3-CPU specifies that the positive and negative edges of interrupts belonging to a component X should alternate starting from a positive edge.

$$\big(\textit{start} \ll \textit{set-irq-pos-X} \mid \textit{Non-Repeated}\big) \tag{A1-CPU}$$

$$\big(\textit{start} \ll \textit{set-irq-neg-X} \mid \textit{Non-Repeated}\big) \tag{A2-CPU}$$

$$\big(\textit{set-irq-pos-X} \ll \textit{set-irq-neg-X} \mid \textit{Non-Repeated}\big) \tag{A3-CPU}$$

The antecedent requirement A4-CPU states that the CPU can get interrupts from other components (the LCDC, the SEN, the GPIO, the IPU and the TMR1) only if it has activated at least one of them.

$$\Big(\big(\{\textit{start-LCDC}, \textit{act-shttr-SEN}, \textit{start-GPIO}, \textit{start-IPU}, \textit{start-TMR1}\}, \vee, \textit{Non-Shuffled}\big) \ll \textit{set-irq-pos} \mid \textit{Non-Repeated}\Big) \tag{A4-CPU}$$

3.4.1.2 Timed Implication Constraints

The timed implication constraints T1-CPU–T5-CPU specify the interrupt handling behavior of the embedded software. The property T1-CPU (resp. T2-CPU, T3-CPU) specifies that whenever the CPU

Interface Names	Interpretation
Inputs:	**Actions of the Environment:**
start	start the CPU
set-irq-pos/neg-lcdc/sen/gpio/ipu/tmr1	set the positive (resp. negative) edge of an interrupt belonging to the LCDC (resp. the SEN, the GPIO, the IPU, the TMR1) by means of the CPU's port *irq-cpu*
Outputs:	**Actions of the CPU (the embedded software):**
set-irq-ack-lcdc/sen/gpio/ipu/tmr1-INTC	acknowledge an interrupt belonging to the LCDC (resp. SEN, GPIO, IPU, TMR1) by writing the register `irq-ack` of the INTC
set-irq-enbl-gpio/sen-INTC	enable interrupts from the GPIO (resp. the SEN) by writing the register `irq-enbl` of the INTC
set-irq-dsbl-gpio/sen-INTC	disable interrupts from the GPIO (resp. the SEN) by writing the register `irq-dsbl` of the INTC
read/write-MEM	read from (resp. write to) the memory MEM
read-buff-SEN-1/2	read the buffer of the image sensor SEN (two instances of the output for the timed implication constraints T7-CPU and T8-CPU respectively)
get-buff-addr-SEN	get the buffer address of the SEN by reading its register `buff-addr`
act-shttr-SEN-1/2	activate a shutter of the SEN by writing its control register `act-shttr` (tree instances of the output for the timed implication constraints T6-CPU, T7-CPU and T10-CPU respectively)
set-buff-addr-LCDC-1/2/3	set the address of the LCDC's buffer by writing its register `buff-addr` (tree instances of the output for the timed implication constraints T6-CPU, T7-CPU and T9-CPU respectively)
write-buff-LCDC-1/2	write to the buffer of the LCDC (two instances of the output for the guarantees T7-CPU and T9-CPU respectively)
start-LCDL/GPIO	start the LCDC (resp. GPIO) by writing the control register `start`
set-img-addr-IPU	configure the IPU with the address of the analyzed image by writing its register `img-addr`
set-img-size-IPU	configure the IPU with the size of the image by writing the register `img-size`
set-gl-addr-IPU	configure the IPU with the address of the image gallery by setting the value of the register `gl-addr`
set-gl-size-IPU	configure the IPU with the size of the image gallery by writing the component's register `gl-size`
get-conf-val-IPU	get the confidence value computed by the face recognition of the IPU by reading the register `conf-val`
get-ref-img-IPU	get the address of the reference image corresponding to the best match of the face recognition by reading the register `ref-img` of the IPU
start-IPU	start the face recognition analysis performed by the IPU by writing its control register `start`
unlock-LOCK	unlock the door by writing the control register `unlock` of the LOCK
set-t-scale-TMR1	set the time scale of the timer TMR1 by writing the register `t-scale`
start/stop-TMR1	start (resp. stop) the timer TMR1 by writing the control register `start` (resp. `stop`)

Table 3.1 The input/output interface of the CPU (and the embedded software).

gets an interrupt belonging to the LCDC (resp. the SEN, the IPU) it should *immediately* (i.e., within 0 nanoseconds) acknowledge it.

$$\big(\textit{set-irq-pos-lcdc} \Longrightarrow \textit{set-irq-ack-lcdc-INTC} \mid 0ns\big) \tag{T1-CPU}$$

$$\big(\textit{set-irq-pos-sen} \Longrightarrow \textit{set-irq-ack-sen-INTC} \mid 0ns\big) \tag{T2-CPU}$$

$$\big(\textit{set-irq-pos-ipu} \Longrightarrow \textit{set-irq-ack-ipu-INTC} \mid 0ns\big) \tag{T3-CPU}$$

If the CPU gets an interrupt belonging to the GPIO, the embedded software should first mask interrupts of the GPIO and the image sensor SEN in any order, and then send acknowledgement for the received interrupt:

$$\begin{aligned}\Big(\textit{set-irq-pos-gpio} \Longrightarrow &\big(\{\textit{set-irq-dsbl-gpio-INTC}, \textit{set-irq-dsbl-sen-INTC}\}, \wedge, \textit{Non-Shuffled}\big)\\ &< \textit{set-irq-ack-gpio-INTC} \mid 0ns\Big)\end{aligned} \tag{T4-CPU}$$

The time implication constraint T5-CPU defines that if the CPU gets an interrupt belonging to the timer TMR1, the embedded software (i) stops the timer, (ii) enables interrupts from both the GPIO and the SEN in any order, and (iii) acknowledges the interrupt of the TMR1:

$$\begin{aligned}&\Big(\textit{get-irq-pos-tmr1} \Longrightarrow \textit{stop-TMR1}\\ &< \big(\{\textit{set-irq-enbl-gpio-INTC}, \textit{set-irq-enbl-sen-INTC}\}, \wedge, \textit{Non-Shuffled}\big)\\ &< \textit{set-irq-ack-tmr1-INTC} \mid 0ns\Big)\end{aligned} \tag{T5-CPU}$$

The timed implication constraint T6-CPU specifies the *initialization mode* of the CPU. It defines that when the CPU is started, the embedded software configures the TMR1 and the LCDC with respectively the time scale value and the address of the buffer, and gets the address of the SEN's buffer; all these actions are performed in *any order*. Then the CPU activates the shutter of the SEN, and starts the LCDC and the GPIO; everything is done in any order. All mentioned actions should not take more than 200 nanoseconds:

$$\begin{aligned}&\Big(\textit{start} \Longrightarrow \big(\{\textit{set-t-scale-TMR1}, \textit{set-buff-addr-LCDC-1}, \textit{get-buff-addr-SEN}\}, \wedge, \textit{Non-Shuffled}\big)\\ &< \big(\{\textit{act-shttr-SEN-1}, \textit{start-LCDC}, \textit{start-GPIO}\}, \wedge, \textit{Non-Shuffled}\big) \mid 200ns\Big)\end{aligned} \tag{T6-CPU}$$

The guarantee T7-CPU specifies the *image capturing/displaying mode* of the CPU. It states the following: If acknowledgement for the received interrupt of the SEN had effect (i.e., a negative edge of the interrupt has been detected), the software moves the last captured image by the SEN from the SEN's buffer to the buffer of the LCDC. Then the CPU reconfigures the address of the LCDC's buffer and activates the SEN's shutter. All actions should be performed within at most 400 nanoseconds.

$$\begin{aligned}&\big(\textit{set-irq-ack-sen-INTC} < \textit{set-irq-neg-sen} \Longrightarrow \textit{read-buff-SEN-1}^{[100,19000]}\\ &< \textit{write-buff-LCDC-1}^{[100,19000]} < \textit{set-buff-addr-LCDC-2} < \textit{act-shttr-SEN-2} \mid 400ns\big)\end{aligned} \tag{T7-CPU}$$

The guarantee T8-CPU defines the set of actions performed by the CPU when the interrupt of the GPIO was successfully acknowledged; it corresponds to the *face recognition mode*. The actions are: (i) transfer of the last image captured by the SEN before the press of the Button-Start was detected by the GPIO, (ii) configuration of the IPU, (iii) start of the face recognition:

$$\begin{aligned}&\Big(\textit{set-irq-ack-gpio-INTC} < \textit{set-irq-neg-gpio} \Longrightarrow \textit{read-buff-SEN-2}^{[100,19000]} < \textit{write-MEM}^{[100,19000]}\\ &< \big(\{\textit{set-img-addr-IPU}, \textit{set-img-size-IPU}, \textit{set-gl-addr-IPU}, \textit{set-gl-size-IPU}\}, \wedge, \textit{Non-Shuffled}\big)\\ &< \textit{start-IPU} \mid 400ns\Big)\end{aligned} \tag{T8-CPU}$$

The timed implication constraint T9-CPU defines the *salutation mode* of the CPU. When acknowledgement for an interrupt belonging to the IPU was sent and a negative edge of the interrupt has been

Interface Names	Interpretation
Inputs:	**Actions on the Environment:**
set/get-img-addr	set (resp. get) the image address by writing (resp. reading) the register `img-addr`
set/get-img-size	set (resp. get) the image size by writing (resp. reading) the register `img-size`
set/get-gl-addr	set (resp. get) the gallery address by writing (resp. reading) the register `gl-addr`
set/get-gl-size	set (resp. get) the gallery size by writing (resp. reading) the register `gl-size`
get-conf-val	get the confidence value by reading the register `conf-val`
get-ref-img	get the reference image address by reading the register `ref-img`
start	start the face recognition by writing the control register `start`
Outputs:	**Actions of the IPU:**
read-img	read the image under analysis by accessing external memory
read-gl-img	read images from the gallery by accessing external memory
set-irq-pos/neg	set the positive (resp. negative) edge of an interrupt through the port *irq-ipu*

Table 3.2 The input/output interface of the IPU.

detected, the embedded software (i) gets the confidence value and the address of the reference image computed by the IPU in any order, (ii) writes the image to be displayed (either the salutation or "access-denied" image) to the LCDC's buffer, (iii) reconfigures the LCDC with the new buffer address, and (iv) starts the timer TMR1:

$$\begin{aligned}&\Big(\textit{set-irq-ack-ipu-INTC} < \textit{set-irq-neg-ipu}\\&\Longrightarrow \big(\{\textit{get-conf-val-IPU}, \textit{get-ref-img-IPU}\}, \wedge, \textit{Non-Shuffled}\big) < \textit{read-MEM}^{[100,19000]}\\&< \textit{write-buff-LCDC-2}^{[100,19000]} < \textit{set-buff-addr-LCDC-3} < \textit{start-TMR1} \mid 400ns\Big)\end{aligned} \tag{T9-CPU}$$

The property T10-CPU corresponds to the *restart mode* of the CPU. It states that once an interrupt from the TMR1 was acknowledged and its negative edge has been detected, the CPU *immediately* activates the shutter of the SEN.

$$\big(\textit{set-irq-ack-tmr1-INTC} < \textit{set-irq-neg-tmr1} \Longrightarrow \textit{act-shttr-SEN-3} \mid 0ns\big) \tag{T10-CPU}$$

3.4.2 The Image Processing Unit (IPU)

The interface of the Image Processing Unit (IPU) is presented in Table 3.2. As before, the left column is the list of the component's inputs and outputs, the right column is the interpretation of each interface name.

3.4.2.1 Antecedent Requirements

The antecedent requirement A1-IPU states that the value of a register Y should be written at least once before it is read (Y stands for *img-add*, *img-size*, *gl-addr*, *gl-size*).

$$\big(\textit{set-X} \ll \textit{get-X} \mid \textit{Non-Repeated}\big) \tag{A1-IPU}$$

The face recognition of the IPU can be started only if the component is fully configured, i.e., the image address, the image size, the gallery address, the gallery size are defined at least once (A2-IPU). Moreover, the image address should be set before each face recognition (A3-IPU).

$$\begin{aligned}&\Big(\big(\{\textit{set-img-addr}, \textit{set-img-size}, \textit{set-gl-addr}, \textit{set-gl-size}\}, \wedge, \textit{Non-Shuffled}\big)\\&\quad \ll \textit{start} \mid \textit{Non-Repeated}\Big)\end{aligned} \tag{A2-IPU}$$

Interface Names	Interpretation
Inputs:	**Actions of the Environment:**
set/get-buff-addr	set (resp. get) the address of the LCDC's buffer by writing (resp. reading) its register `buff-addr`
start	start the component by writing the register `start`
Outputs:	**Actions of the LCDC:**
read-buff-1/2/3	read the content of the buffer from external memory (tree instances of the output for the timed implication constraints T1-LCDC, T2-LCDC and T3-LCDC respectively)
set-irq-pos/neg-1/2/3	set the positive (resp. negative) edge of an interrupt through the port *irq-lcdc* (tree instances of the output for the timed implication constraints T1-LCDC, T2-LCDC and T3-LCDC respectively)

Table 3.3 The input/output interface of the LCDC.

$$\big(\textit{set-img-addr} \ll \textit{start} \mid \textit{Repeated}\big) \tag{A3-IPU}$$

The confidence value (resp. the address of the reference image) provided by the IPU can be accessed only if it has been computed at least once, i.e., at least one started face recognition has terminated with an interrupt:

$$\big(\textit{set-irq-pos} \ll \textit{get-conf-val} \mid \textit{Non-Repeated}\big) \tag{A4-IPU}$$

$$\big(\textit{set-irq-pos} \ll \textit{get-ref-img} \mid \textit{Non-Repeated}\big) \tag{A5-IPU}$$

3.4.2.2 Timed Implication Constraint

The timed implication constraint T1-IPU states that when the face recognition performed by the IPU is started, the component (i) reads the image under analysis from external memory and the images from the gallery, (ii) sends an interrupt (the positive edge being followed by the negative edge). The face recognition should not take more than 500 nanoseconds.

$$\begin{aligned}\Big(&\textit{set-img-addr} < \textit{start} \Longrightarrow \big(\{\textit{read-img}^{[100,19000]}, \textit{read-gl-img}^{[10K,2000000]}\}, \wedge, \textit{Shuffled}\big)\\ &< \textit{set-irq-pos} < \textit{set-irq-neg} \mid 500ns\Big)\end{aligned} \tag{T1-IPU}$$

3.4.3 The Liquid Crystal Display Controller (LCDC)

The input/output interface of the LCDC is defined in Table 3.3.

3.4.3.1 Antecedent Requirements

The assumption A1-LCDC defines that the address of the LCDC's buffer can be read only if it has been set at least once. The assumption A2-LCDC states that the LCDC can be started only if the buffer address is defined.

$$\big(\textit{set-buff-addr} \ll \textit{get-buff-addr} \mid \textit{Non-Repeated}\big) \tag{A1-LCDC}$$

$$\big(\textit{set-buff-addr} \ll \textit{start} \mid \textit{Non-Repeated}\big) \tag{A2-LCDC}$$

3.4.3.2 Timed Implication Constraints

The intended behavior of the started LCDC is the following: (i) the component reads its buffer, and then (ii) sends an interrupt. The activities (i) and (ii) should take at most 300 nanoseconds; this value represents the screen refreshing time. After sending an interrupt the LCDC loops to the reading of its buffer. The first iteration of the LCDC's loop behavior is formulated by the guarantee T1-LCDC. The timed implication constraints T2-LCDC and T3-LCDC encode the repeated behavior of the LCDC: the

Interface Names	Interpretation
Inputs:	**Actions of the Environment:**
act-shttr	activate the shutter of the SEN by writing its control register `act-shttr`
get-buff-addr	get the address of the buffer by reading the register `buff-addr`
read-buff	read the content of the SEN's buffer
Outputs:	**Actions of the SEN:**
set-irq-pos/neg	set the positive (resp. negative) edge of an interrupt through the port *irq-sen*

Table 3.4 The input/output interface of the SEN

guarantee T2-LCDC specifies the behavior of the even iterations of the loop. The guarantee T3-LCDC specifies the behavior of the odd iterations except the first one. Conceptually outputs *X-1/2/3* (*X* stands for *read-buff*, *set-irq-pos*, *set-irq-neg*) represent one action *X* performed by the LCDC.

$$\big(\textit{set-buff-addr} < \textit{start} \Longrightarrow \textit{read-buff-1}^{[100,19000]} < \textit{set-irq-pos-1} < \textit{set-irq-neg-1} \mid 300ns\big) \quad \text{(T1-LCDC)}$$

$$\Big(\big(\{\textit{set-irq-neg-1}, \textit{set-irq-neg-3}\}, \vee, \textit{Non-Shuffled}\big) \Longrightarrow \textit{read-buff-2}^{[100,19000]} < \textit{set-irq-pos-2} < \textit{set-irq-neg-2} \mid 300ns\Big) \quad \text{(T2-LCDC)}$$

$$\big(\textit{set-irq-neg-2} \Longrightarrow \textit{read-buff-3}^{[100,19000]} < \textit{set-irq-pos-3} < \textit{set-irq-neg-3} \mid 300ns\big) \quad \text{(T3-LCDC)}$$

3.4.4 The Image Sensor (SEN)

Table 3.4 defines the input/output interfaces of the image sensor SEN. The antecedent requirement A1-SEN states that the SEN's buffer can be accessed only if its address has been requested at least once. The timed implication constraint T1-SEN defines the behavior of the SEN. It states that if the shutter of the SEN is activated, the component sends an interrupt (after capturing an image) within 200 nanoseconds.

$$\big(\textit{get-buff-addr} \ll \textit{read-buff} \mid \textit{Non-Repeated}\big) \quad \text{(A1-SEN)}$$

$$\big(\textit{act-shttr} \Longrightarrow \textit{set-irq-pos} < \textit{set-irq-neg} \mid 200ns\big) \quad \text{(T1-SEN)}$$

3.4.5 Other Components

The Timers TMR1 and TMR2 The behavior of the timers cannot be specified in terms of the loose-ordering language. When the timer TMR1 (resp. the TMR2) is started, it produces an interrupt through its port *irq-tmr1* (resp. *irq-tmr2*) whenever the predefined time interval elapses (i.e., at each clock tick). Recall, we refer to a time interval between two consecutive interrupts as a time scale of the timer. Production of interrupts takes place unless the timer is *stopped*. To specify only repetitive production of interrupts, one could apply the approach used for the specification of the LCDC. Particularly, one could define timed implication constraints for (i) the first production of an interrupt after start of the timer, (ii) odd and even productions of interrupts. Nevertheless, it is not sufficient, because the timed implication constraint ($\mathcal{P} \Longrightarrow \mathcal{Q} \mid t$) allows checking only the fact that the production of outputs $\mathcal{Q}$ (in our case interrupts) *does not take more time than t*, where *t* is *statically defined*. For the timers it causes a problem for two reasons: (i) between two sent interrupts it should elapse the exact specified time interval, i.e., if it takes less time than *t* to produce the next interrupt, it is an error; (ii) the time scale is defined *dynamically* as a value of the register `t-scale`.

Another problem emerges when one makes an attempt to define stopping of the timer. One would need to state that an interrupt is produced *only if* the timer is not stopped, and this type of properties is not expressible in the loose-ordering language.

The GPIO The component produces interrupts through its port *irq-gpio*, if the BUTTON-START is pressed. Since the GPIO encapsulates the button, the production of interrupts is *uncontrollable*, i.e., it cannot be activated by means of inputs. Therefore, the GPIO's behavior cannot be defined in terms of inputs/outputs only; particularly it cannot be expressed in the loose-ordering language.

The Interrupt Controller When a component is completely deterministic, as the interrupt controller INTC, it does not seem to be reasonable to use loose-orderings for its specification (though in some cases it can be possible, see description of the Bus below). We provide the specification of the INTC in the form of the imperative C++ program. The intended behavior of the INTC cannot be expressed only in terms of the loose-ordering language due to the same reason as the stopping behavior of the timers discussed above: masking interrupt requests may *cancel* (or *postpone*) generation of respective interrupts by the INTC.

The Bus The behavior of the Bus can be defined as the conjunction of the timed implication constraints of the form:

$$\big(input \Longrightarrow output \mid 0ns\big) \qquad \text{(T1-BUS)}$$

where *input* (resp. *output*) stands for all different transactions which the Bus can receive from (resp. should send to) the connected components (the CPU, the IPU, etc.). An alternative to the guarantee T1-BUS could be the `if-then` construction provided by most of the imperative languages:

`if` *input* `then` *output*

3.5 Synchronization Bugs

The TLM virtual prototype presented in Section 3.2 consists of the set of components which implement the functionality of the intercom as it is defined in Section 3.1. Individually all components are implemented as it is defined in Section 3.2.1, and controlled by the embedded software running on top of the CPU, which algorithm is discussed in Section 3.3. To implement the functionality, some components implement algorithms (e.g., the IPU, the SEN, the CPU) and all of them *synchronize* with each other by means of transactions and interrupts.

Even if each of the components are "correct" in isolation (e.g., implementation of computation algorithms is ensured to be correct), it can be difficult to guarantee that the ensemble of the components will implement the specified (expected) behavior [AH01]. In other words, when the components are put together bugs may appear due their wrong synchronization. We call a *synchronization bug* a malfunction of the system caused by the communication of two components. Although the description of the TL model of the intercom provided in Section 3.2 and the embedded software introduced in Section 3.3 are free from synchronization bugs, this result was achieved through the painful process of testing, debugging and exploration of sources of the system's malfunction. All bugs occurred due to the wrong synchronization of the components since it is assumed that the computation algorithms of the IPU and the SEN are correct (in our case they are trivial as it was described in Sections 3.2.1.2 and 3.2.1.3). The list of the synchronization bugs is provided below. For sake of convenience, we associate them with respective reference names. The listed synchronization bugs will be investigated in Chapter 7 as a part of our experimental settings. There are bugs which are *visible* to the user: when they occur the system continues to operate, but the results the user gets do not correspond to his/her expectations. The expectations of the user in this case are defined by the functional specification of the intercom defined in Section 3.1. These bugs are BS1 – BS12. There are also bugs which cause crash of the system, they are BS13 – BS15.

BS1: The BUTTON-START does not respond whatever the execution phase of the system is.

BS2: The display of the intercom is always black.

BS3: The display of the intercom shows nonsense.

BS4: The system gets stuck at the face recognition phase: the BUTTON-START does not respond, the same (currently analyzed) image is shown on the display.

BS5: When the intercom is in the face recognition phase (i.e., the face recognition was started with the press of the BUTTON-START and it has not terminated), the display starts to show the images

being captured by the sensor SEN (i.e, the intercom "jumps" to the image capturing/displaying phase).

BS6: For *any user* (either registered in the system or unknown) face recognition always results in the closed door and the "access-denied" image shown on the display.

BS7: The *registered user* gets the "access-denied" notification on the display and (s)he is forbidden to enter the building.

BS8: The *registered user* sees on the screen a salutation image addressed to another user.

BS9: The *unregistered user* sees on the screen a salutation image addressed to someone else, and gets access into the building.

BS10: The intercom starts face recognition of either the "access-denied" or the salutation image.

BS11: When the face recognition terminates and either the "access-denied" image or salutation one is shown on the display, the system gets stuck: the Button-Start does not respond, no change of the image on the display occurs.

BS12: The user is not informed about the results of the face recognition, i.e., neither the "access-denied" image nor the salutation one is shown on the display.

BS13: The Bus fails to establish the target component for a transaction initiated by the CPU.

BS14: The Bus cannot establish the target component for a transaction initiated by the IPU.

BS15: The Bus cannot establish the target component for a transaction initiated by the LCDC.

When any of these bugs occurs, how would one establish the source of the occurring bug? For instance, if the Button-Start does not respond (BS1), is it a fault of the GPIO, or the embedded software which "forgot" to start the component? Or is it the TMR2 which was started by the GPIO without the time scale being specified? Or the bug was caused by some other reason? In Chapter 7 it will be shown that in fact each of the listed possibilities could cause the described defect of the system. Moreover, any of the bugs listed above can have several potential sources, and their localization requires additional time and effort. To ensure functional correctness of the system and to facilitate localization of the bugs, one needs to use the system's *specification*[3]. Moreover the specification should be *executable* such that the system's faults with regard to the specification could be detected during simulation of the SystemC/TLM virtual prototype. The first part of this document is devoted to the specification and testing of TL models.

Summary

In this chapter, we have presented our running example named "Smart Intercome". It is a system-on-chip which controls the access to a building based on the face recognition analysis. We have given an overview of the system's functionality, have considered the TL models of the system's components, and have explained the control algorithm implemented by the embedded software. The properties of these components are used through the whole document serving examples for the introduced notions, as well as a basis for our experiments. We have listed the set of synchronization bugs which can occur in the intercom system.

[3]The concept of specification is introduced in the background chapter (see Sec. 2.3.1).

Part III

Efficient Monitoring of Loose-Ordering Properties for SystemC/TLM

Chapter 4

Introducing the Notion of Loose-Ordering

Contents

4.1 Introduction **65**
4.2 Motivation for Loose-Orderings **67**
4.2.1 Over-Constraining 67
4.2.2 Robustness of the Embedded Software 68
4.3 Proposal **69**
4.3.1 Loose-Orderings 69
4.3.2 Loose-Ordering Patterns 69
4.4 Definitions **70**
4.4.1 Input/Output Interface 71
4.4.2 Loose-Orderings 71
4.4.3 An Antecedent Requirement 72
4.4.4 A Timed Implication Constraint 74
4.5 Encoding Loose-Ordering Properties **75**
4.5.1 The Encoded Subset 75
4.5.2 Encoding Using SERE Operators 76
4.5.3 Encoding Using LTL Operators 78

In this chapter we introduce specification primitives called *loose-orderings* which facilitate definition of order and number non-determinism. To the best of our knowledge such primitives are original; they are not supported by existing specification languages. Moreover, encoding of our primitive constructs into those languages using their provided operators (like temporal and SERE[1] operators) can be hard, and can lead to combinatorial explosion either of the size of the obtained temporal formula, or of the size of its vocabulary. We define the set of specification patterns on top of loose-orderings to facilitate definition of the most common types of properties of the hardware design. The patterns define *assumptions* and *guarantees* of specified components.

4.1 Introduction

SystemC-based Transaction-Level Modeling (TLM) [Sys] has been very successful in providing high-level executable component-based models for systems-on-chip (SoCs). The rationale has been to raise the level of abstraction by removing details of lower models like RTL models, especially on timing aspects. The notion of *loose-timing* is very interesting in that perspective. Exact delays in SystemC models (e.g., `wait(100, SC_NS);`) have been identified as a source of over-constraints and spurious synchronizations in models. The *loose-timing* principle allows to write `wait (90, 110, SC_NS);`, to specify a non-deterministic delay. The simulation engine draws a random value in the interval, thus exploring more behaviors. The *coverage* problem involved by this non-deterministic specification has been addressed in, e.g., [Hel+06], using dynamic partial order reduction techniques [FG05].

[1] Recall, according to [18505] SERE is an acronym which is translated to *"Sequential Extended Regular Expression"*.

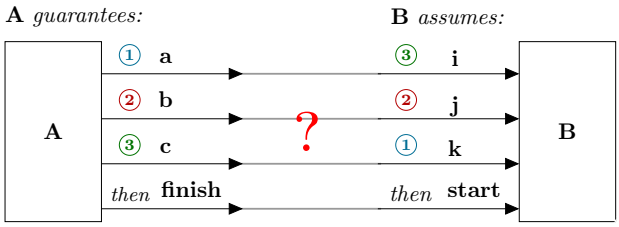


Figure 4.1 Example of over-constraints related to the order of interactions.

Introducing Loose-Ordering In this thesis, we identify another source of over-constraints. Typically, when a component needs several input data (e.g., the address of an image, the size of it, etc.) before one of the functions it provides (e.g., some transformation of the image) can be started, the *order* in which the input data elements are provided is usually irrelevant. The same is true for data and control outputs. Any specification in which the order is imposed is over-constrained. For example, consider a component **A** (resp. **B**) with data outputs **a, b, c** (resp. data inputs **i, j, k**) and a control output **finish** (resp. a control input **start**). Suppose a specification of **A** (resp. **B**) stating that **a, b, c, finish** (resp. **k, j, i, start**) are produced (resp. consumed) in this exact *total* order. The connection of **a** to **i**, **b** to **j** and **c** to **k** is incompatible with such a specification (Fig. 4.1). In fact, **a, b, c** can be produced in any order, before **finish**; and **i, j, k** can be received in any order, before **start**. The specification of **A** and **B** should not over-constrain the order of interactions.

This type of property can already be expressed in languages like the Property Specification Language (PSL) [CVK04; 18505][2]. But even simple loose-ordering properties require complex formulas, hence dedicated constructs are helpful. By reviewing industrial models we identified main *loose-ordering* properties and proposed the set of *patterns* to capture them. This is not meant to be a complete specification language that could replace PSL or *e* [IJ04], but rather a proposal of new dedicated constructs that could be integrated in these existing languages.

Efficient Monitoring for Loose-ordering Properties The complete integration of loose-ordering principles in the ABV[3] framework has two facets: monitoring these new properties efficiently, and generating random values according to these properties. In the first part of this document we address the *monitoring* aspects for loose-ordering properties. When any component (of type **A** or **B** of the above example) is specified with a loose-ordering property, the ABV framework has to include the monitoring of this type of constraints. The first idea is to translate automatically our new properties into PSL, for which there exist monitor generation techniques [Pie07; PF08]. However, we will show that this produces complex formulas. Then, even the efficient techniques for exploiting such logics (e.g., the automatic modular generation of monitors described in [MAB07a; PF08]) cannot do better than producing complex monitors from the obtained complex formulas.

We propose two different translations of loose-ordering properties into PSL, and a direct translation into very efficient monitors that benefit from the particular form of our properties (Fig. 4.2). We do not prove formally the uniqueness of the translation into PSL. Our encoding is intuitive and it is not difficult to check its correctness. The direct synthesis of very efficient monitors is compositional, thus, it is easier to prove their correctness.

A complete integration of loose-ordering principles in the ABV framework also requires an extension of the random generation techniques: when a component like the component **A** of the above example is used in simulation, several orders for producing outputs should be generated. This is similar to drawing a random value for the loose-timing variant of the `wait` instruction. We address this point in the second part of the thesis (see Chapter 8).

The contributions are: (i) a new notion called *loose-ordering*, allowing to remove the sources of over-constraints due to the order of interactions between components in a TL model; (ii) a set of *patterns* to capture these properties; (iii) a translation into PSL, for comparison purposes; (iv) a direct translation into efficient SystemC monitors.

[2] PSL stands for PSL 1.1.
[3] ABV stands for *Assertion-Based Verification*.

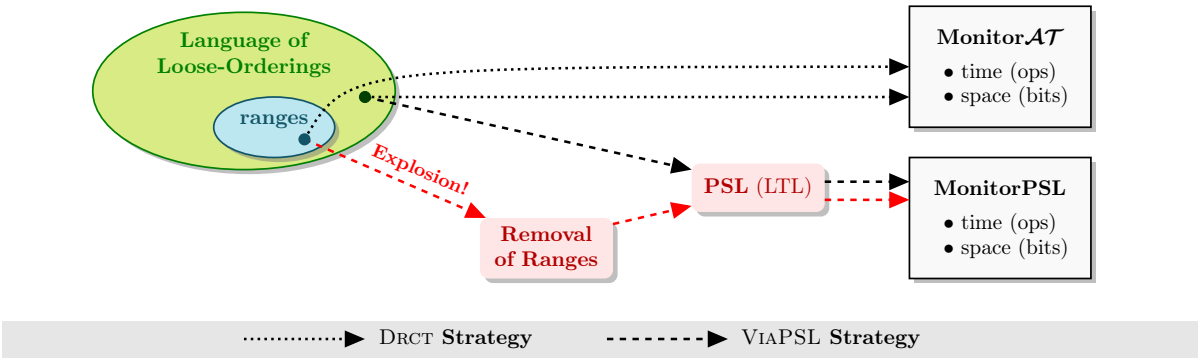


Figure 4.2 Monitoring loose-orderings: attempts at comparison.

Organization of the Material Section 4.2 details motivation for loose-orderings and illustrates our patterns. In Section 4.4 we define loose-orderings and the patterns formally. Section 4.5 provides translation of loose-orderings into PSL. The direct translation of loose-orderings into efficient SystemC/TLM monitors and the evaluation results are presented in Chapters 5, 6 and Chapter 7 respectively.

4.2 Motivation for Loose-Orderings

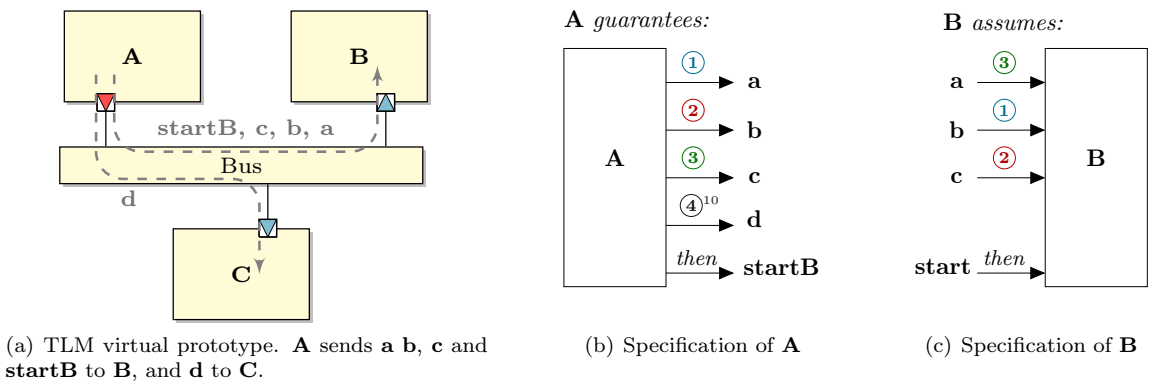
Synchronization bugs of TL models usually occur because some computation of a component is started before all relevant registers of the component get values (i.e., before the component is configured), or because results of the computation (which are stored in respective registers of the component) are accessed before they are computed (i.e., before the computation terminates). Such types of properties can be *implicitly assumed* when a TL model is developed. It makes it easy to "forget" about some of such implicit assumptions while implementing a model. Moreover, when respective synchronization bugs occur it can be very hard to find their sources, specially if the same malfunction has several causes (e.g., see Chapter 7). To avoid these problems, one needs to specify components *explicitly*.

To check synchronization between TLM components, one needs to specify their *border behavior* (protocol). Such specification defines what a component *expects* (assumes) to get from other components, and what it *provides* (guarantees) if its expectations are met. Specification consists of *temporal properties* defining *sequences* (orders) of expected input and output events. For instance, Figure 4.3 shows the specification of TLM components **A** and **B**: **A** guarantees to provide **a**, **b**, **c** in this order, then **d** ten times, and then **startB**; **B** assumes to get **b**, **c**, **a** in this order and then **start** (**a**, **c**, etc., may correspond to either TLM transactions or SystemC interrupt requests).

4.2.1 Over-Constraining

Order properties as those shown for **A** and **B** in Figure 4.3 are typical for most of hardware specification languages relying on regular expressions and LTL. This is because the definition of exact orders in those languages is intuitive. Such properties may cause *order over-constraining* of the design. Order over-constraining makes components *incompatible* and leads to spurious detection of bugs. For instance, suppose the component **A** sends **a**, **b**, **c** and **startB** to **B** (see Fig. 4.4(a)). If **A** behaves according to its specification (see Fig. 4.3(b)) and first provides **a**, it violates the specification of **B** which states that **B** should get **b** first (see Fig. 4.3(c)). Thus, the specification of **B** complains despite the fact that the component may operate normally *whatever the order* in which it gets its inputs (which can be the case for TLM components). It means that there is a need for specification primitives which would allow *order non-determinism*. Figure 4.4 shows the non-deterministic specification of the components **A** and **B**. Here, **A** (resp. **B**) guarantees to provide (resp. assumes to get) **a**, **b** and **c** *in any order*, before **startB** (resp. **start**). At simulation time **A** will produce **a**, **b** and **c** for **B** in one specific order.

Another source of over-constraining emerges due to the need to specify that some operation of a component is repeated. The problem raises because the number of repetitions may not be statically known. For instance, specification of **A** states that **d** should be repeated 10 times in a row (Fig. 4.3(c)). If **A** produces **d** 11 times, it violates its guarantees. The problem can be solved, if one defines that **d** is repeated *several times* from a certain *range* (see Fig. 4.4(b)). Therefore, there is a need for specification

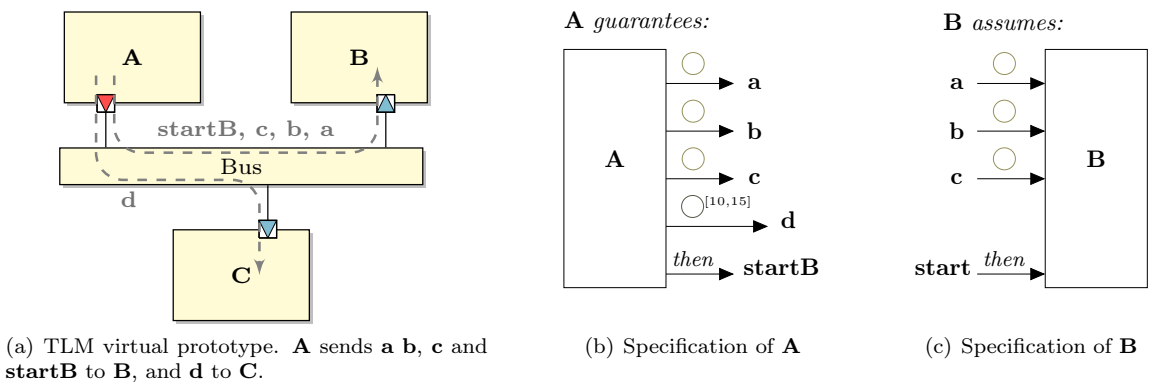


(a) TLM virtual prototype. **A** sends **a** **b**, **c** and **startB** to **B**, and **d** to **C**.

(b) Specification of **A**

(c) Specification of **B**

Figure 4.3 Example of over-constrained specifications for components **A** and **B**.



(a) TLM virtual prototype. **A** sends **a** **b**, **c** and **startB** to **B**, and **d** to **C**.

(b) Specification of **A**

(c) Specification of **B**

Figure 4.4 Non-deterministic specifications for components **A** and **B**.

primitive which would allow *number non-determinism*. At simulation time the component will repeat **d** one specific number of times in the defined range.

4.2.2 Robustness of the Embedded Software

Even when we define non-deterministic specifications for **A** and **B** as described above, the *deterministic* SystemC scheduler (see Sec. 2.1.4.8) may not *cover* the possible choices well. For instance, during all run simulations **A** may always provide **a**, **b** and **c** in this exact order, and other orders of the outputs might not be examined. This is similar to the coverage problem involved by *loose-timing* specifications which has been addressed in [Hel+06].

The coverage of *loosely-orderings* effects robustness of the embedded software. For instance, let assume that the component **B** (resp. **D**) guarantees to provide an interrupt **irqB** (resp. **irqD**) when it gets **startB** (resp. **startD**). The order in which the component **A** starts **B** and **D** (see Fig. 4.5) may imply the order in which **A** gets interrupts from the components. Thus, to ensure that **A** is robust, i.e., the component can adequately react on the interrupts **irqB** and **irqD**, one needs to consider all possible activation orders

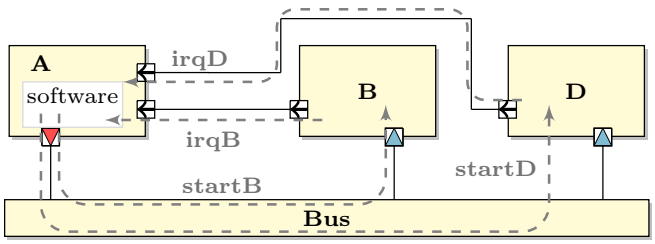


Figure 4.5 Robustness of the embedded software: any order of **startB** and **startD** should be examined to observe all orders of **irqB** and **irqD**.

of **B** and **D**. Although we do not address the coverage problem involved by *loose-orderings* in this work, in Chapter 10 we will provide guidelines on writing TL models of components in such a way that each component may produce its outputs in non-deterministic order at simulation.

4.3 Proposal

4.3.1 Loose-Orderings

To address the over-constraining issues discussed above, we introduce the notion of *loose-orderings*. They allow (i) the definition of *ranges* of the form $\mathbf{d}^{[10,15]}$ read as *repeat* **d** *from* 10 *to* 15 *times in a row*, (ii) definition of *fragments* made of ranges that can occur in any order, (iii) the definition of *loose-orderings* made of ordered sequences of fragments (notice, we call it loose-ordering because the order *inside* fragments is free). For instance, one may define the loose-ordering for the component **A** in the following way

$$\underbrace{\underbrace{(\{\mathbf{a},\mathbf{b},\mathbf{c}\},\wedge,\textit{Non-Shuffled})}_{\text{a fragment}} < \overbrace{\mathbf{d}^{[10,15]}}^{\text{a range}}}_{\text{a loose-ordering}}$$

It states that **a**, **b** and **c** all ($\wedge$) occur, in any order, and then **d** occurs several times in the range $[10, 15]$. The meaning of *Non-Shuffled*, the syntax and the semantics of loose-orderings will be defined in Section 4.4.2 below.

Loose-orderings are an intermediate form between the explicit enumeration of all total orders of names (e.g., **abcd...d**, **bacd...d**, etc.) and any partial order (e.g., **a**, **b**, **c** and **d** in any order). The concept is intuitive, and it seems to be adequate for industrial SystemC/TLM models (see Sec. 3.4).

4.3.2 Loose-Ordering Patterns

To facilitate specification of components, we propose a set of patterns defined on top of loose-orderings. The patterns are defined based on the observation that most of synchronization bugs of TL models (e.g., see the list of bugs in Sec. 3.5) occur due to the violation of certain types of properties. The first group of those properties ensures *proper configuration* of components before their start. For instance, recall our intercom running example (see Chapter. 3). One of its (potential) malfunctions could be described as "*an unregistered user gets access to the building*". This bug is caused by the wrong synchronization of the CPU and the IPU: the CPU does not provide the address of the analyzed image to the IPU before starting face recognition. The synchronization bugs BS1, BS7, BS11, BS14, BS15 of our running examples defined Section 3.5 are caused by violating this type of properties.

The second group of properties concerns *relevance of computed data*; they state that results of computation can be accessed only after they have been computed. For instance, recall again the intercom system. Because the CPU gets the reference image before it is actually computed by the IPU, *the user may see on the display of the intercom a salutation image addressed to another user* (this is the bug BS8 from Sec. 3.5). Other examples of the violation of this type of properties are the synchronization bugs BS9 and BS13 defined in Section 3.5.

The third (and the last) type of properties defines *obligations* of components. Such properties state that when a certain *condition* holds, a component should fulfill its *promises*. This type of properties is very common in hardware design. To enable their definition, hardware specification languages like PSL and SVA provide the set of operators of the "implication" kind $\alpha \Mapsto \beta$ (e.g., the PSL property **{a; b}** $\Mapsto$ **{c; d}** uses the weak suffix implication operators "$\Mapsto$" to state that after **a** and **b** occurring in this order, **c** and then **d** should occur, see Sec. 2.3.2.5). The antecedent α (resp. the consequent β) of those operators corresponds to a condition (resp. a promise of a component). For instance, when the CPU detects the press of BUTTON-START (the condition), it should disable the button before starting face recognition (the promise). Otherwise, "*a user may start face recognition of either "access-denied" or salutation image*" (this is the bug BS10, see Sec. 3.5). The bugs BS1, BS2, BS3, BS4, BS5, BS11 and BS12 may appear in the TLM virtual prototype of the intercom due to the violation of this kind of properties.

Generalizing properties of the configuration and the data relevance, we define an *antecedent requirement* pattern of the form

$$(\mathcal{P} \ll i \mid \nabla) \qquad \text{(ANTECEDENT REQUIREMENT)}$$

Grammar Rule	**Constraints**
a range $\mathcal{R} = n^{[u,v]}$	$\alpha(\mathcal{R}) = \{n\} \subseteq I \cup O$ $u, v \in \mathbb{N}$
a fragment $\mathcal{F} = \big(\{\mathcal{R}_1, \ldots, \mathcal{R}_n\}, \sharp, \sqcup\!\sqcup\big)$	$i \neq j \Longrightarrow \alpha(\mathcal{R}_i) \cap \alpha(\mathcal{R}_j) = \varnothing$, $\sharp \in \{\wedge, \vee\}$, $\sqcup\!\sqcup \in \{\textit{Non-Shuffled}, \textit{Shuffled}\}$
a loose-ordering $\mathcal{L} = \mathcal{F}_1 < \cdots < \mathcal{F}_q$	$i \neq j \Longrightarrow \alpha(\mathcal{F}_i) \cap \alpha(\mathcal{F}_j) = \varnothing$
$\mathcal{P} = \mathcal{L}$	$\alpha(\mathcal{P}) \subseteq I \cup O$
$\mathcal{Q} = \mathcal{L}$	$\alpha(\mathcal{Q}) \subseteq O$
an antecedent requirement $\mathcal{A} = \big(\mathcal{P} \ll i \mid \nabla\big)$	$\alpha(\mathcal{P}) \cap \{i\} = \varnothing$ $i \in I, \quad \nabla \in \{\textit{Non-Repeated}, \textit{Repeated}\}$
a timed implication constraint $\mathcal{T} = \big(\mathcal{P} \Longrightarrow \mathcal{Q} \mid t\big)$	$t \in \mathbb{N}$

Table 4.1 Abstract grammar and constraints for well-formed formulas.

where $\mathcal{P}$ is a loose-ordering. Intuitively it means the following: before i can occur the loose-ordering $\mathcal{P}$ should have been observed. The parameter $\nabla \in \{\textit{Non-Repeated}, \textit{Repeated}\}$ defines if $\mathcal{P}$ is repeated before the next occurrence of i. $\mathcal{P}$ may define a fragment stating that the configuration data (like **a**, **b** and **c** of the component **B** in Fig. 4.4(c)) is provided in any order; i could stand for a control input (like **start** of **B**). The syntax and the semantics are explained in Section 4.4.3 below.

To define obligations of components, we adapt the suffix implication operator $\alpha \Mapsto \beta$ (see definition in Sec. 2.3.2.5) to our setting as follows: the antecedent α (resp. the consequent β) is a loose-ordering made of inputs and/or outputs (resp. only outputs) of a component. We introduce a *timed implication constraint* with the following syntax

$$\big(\mathcal{P} \Longrightarrow \mathcal{Q} \mid t\big) \qquad \text{(TIMED IMPLICATION CONSTRAINT)}$$

where $\mathcal{P}$ and $\mathcal{Q}$ are loose-orderings and t defines a *bound*. The property has the following meaning: when $\mathcal{P}$ is observed, $\mathcal{Q}$ should be produced within t. Semantically a bound defines the maximum number of steps which can take place before $\mathcal{Q}$ is produced after $\mathcal{P}$ (see Chapter 9). When the design is simulated by means of any scheduler with time, a bound can be mapped to simulation time of that scheduler. The pattern is defined in Section 4.4.4.

4.4 Definitions

We generalize the case-study properties with two patterns for: *antecedent requirements* and *timed implication constraints*. Both patterns are written on the vocabulary of the *input/output interface* (I, O) of the component. The syntax is given by the abstract grammar shown in Table 4.1. The right column of the table is the set of additional syntactic constraints defining well-formed formulas of this grammar. They are expressed using α, which denotes the vocabulary of the respective formula, i.e, the set of interface names (inputs or outputs) that appear in the formula. The constraints mainly state that we should not reuse the same interface names in two ranges, or fragments, of the same property. They are needed for the direct translation of loose-orderings into *efficient* SystemC monitors; the constraints are the trade-off between the usefulness of loose-orderings and the efficiency of the monitors (see Chapter 5). All well-formed formulas are interpreted on sequences where only the names of the root pattern appear; only one name at a time can occur due to asynchrony of considered models. The notion of time used in the timed implication constraint is mapped directly to simulation time of the SystemC simulation kernel.

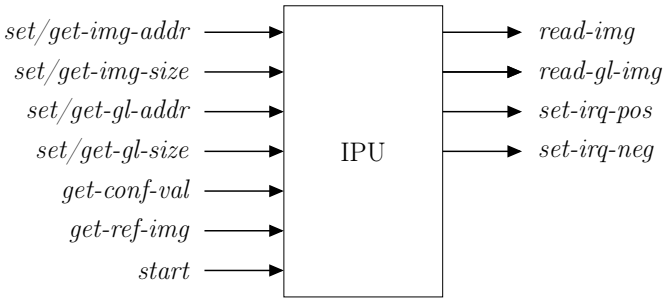


Figure 4.6 The graphical representation of the IPU's input/output interface. Arrow from the left (resp. to the right) are inputs (resp. outputs). *set-X* (resp. *get-X*) stands for "read" (resp. "write") of the IPU's register *X*; *read-img* (resp. *read-gl-img*) stands for "read image" (resp. "read image gallery") operation; *set-irq-pos* (resp. *set-irq-neg*) stands for "set positive (resp. negative) edge of interrupt".

4.4.1 Input/Output Interface

The input/output interface of a component C is a pair of sets (I, O), where I is the set of inputs and O is the set of outputs. *Input* is any action of other components that affects C. *Output* is any activity performed by the component C and visible for other components.

For instance, consider the image processing unit (IPU) which is the part of our running example (see Sec. 3.2.1.3). Recall, the IPU component has the set of read/write data registers `img-addr`, `img-size`, `gl-size` and `gl-addr`; they are used to configure the IPU before the start of face recognition. The recognition is launched by writing the control register `start`. To perform face recognition analysis, the component reads images from the external memory. When face recognition terminates, the IPU sends an interrupt through its interrupt port *irq-ipu*. The results of face recognition are stored into the read-only registers `conf-val` and `ref-img`.

Figure 4.6 shows the input/output interface of the IPU component. We define it as follows. Each data register can be either read or written by other components. Thus, we introduce one input to represent read (resp. write) operation performed on any of the data registers of the IPU. For instance, we define an input *set-img-addr* (resp. *get-img-addr*) to present the "read" (resp. "write") operation for the register `img-addr`[4]. An input *start* represents writing the control register `start`.

The outputs of the IPU represent all actions performed by the IPU while the face recognition is running. They are: *read-img* (resp. *read-gl-img*) which corresponds to reading the analyzed image (resp. images of the image gallery) from the external memory; *set-irq-pos* (resp. *set-irq-neg*) which represents sending the positive (resp. negative) edge of the interrupt signal through the interrupt output port *irq-ipu* of the IPU.

Definition of input/output interfaces for other components of the intercom system is defined in Section 3.4.

4.4.2 Loose-Orderings

As Table 4.1 states, loose-orderings are made of fragments, and fragments are made of ranges. The latter are defined as follows.

Definition 7: Range — A range $\mathcal{R} = n^{[u,v]}$ denotes any sequence of k occurrences of n, $n \in I \cup O$ and $k \in [u, v]$.

For instance, one may define that the IPU component repeats the read image operation (the output *read-img*) several number of times in $[100, 19000]$ using a range of the form:

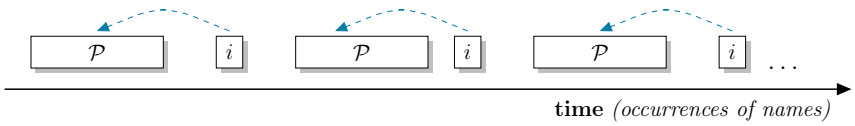
$$\textit{read-img}^{[100,19000]}.$$

Fragments made of ranges can have conjunctive or disjunctive, shuffled or non-shuffled semantics.

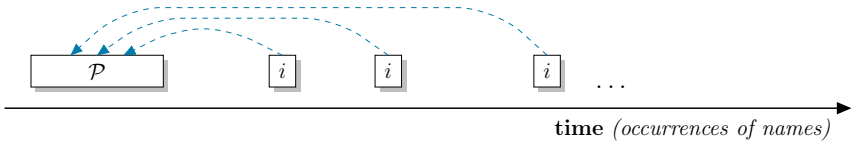
Definition 8: Shuffle of Sequences — A shuffle of two sequences s_1 and s_2 is a sequence s that one can get by interleaving positions of s_1 and s_2 in any way [Hop+00].

Definition 9: Fragment — A fragment $\mathcal{F} = \big(\{\mathcal{R}_1, \ldots, \mathcal{R}_m\}, \sharp, \sqcup\!\sqcup\big)$, $\sharp \in \{\wedge, \vee\}$, $\sqcup\!\sqcup \in \{\textit{Shuffled}, \textit{Non-Shuffled}\}$ is made of sequences $s_1, \ldots s_m$ matching the corresponding ranges. If $\sharp = \wedge$,

[4]Notice, despite the fact that the confidence value (resp. the reference image address) is the *result* (output) of the face recognition, other components get it by reading the register `conf-val` (resp. `ref-img`), i.e., by providing the respective input *get-conf-val* (resp. *get-ref-img*).



(a) A repeated context $\nabla = \textit{Repeated}$



(b) A non-repeated context $\nabla = \textit{Non-Repeated}$

Figure 4.7 Semantics of the antecedent requirement $\mathcal{A} = (\mathcal{P} \ll i \mid \nabla)$.

then all these s_is should appear; if $\sharp = \vee$, at least one of these s_is should appear, and possible several of them. If $\sqcup\!\sqcup = \textit{Shuffled}$, then the appearing s_is are *shuffled*; if $\sqcup\!\sqcup = \textit{Non-Shuffled}$, then the appearing s_is are *concatenated* in any order. The values of $\sharp$ and $\sqcup\!\sqcup$ define semantics of $\mathcal{F}$.

For instance, the configuration of the IPU with the image address (the input *set-img-addr*), the image size (the input *set-img-size*), the gallery size (*set-gl-size*) and the gallery address (*set-gl-addr*) can be defined with the fragment:

$$\big(\{\textit{set-img-addr}, \textit{set-img-size}, \textit{set-gl-size}, \textit{set-gl-addr}\}, \wedge, \textit{Non-Shuffled}\big).$$

It states that *each* of the mentioned inputs should occur. Notice, shuffling here is meaningless since ranges are trivial (they define only one occurrence of respective names).

The fragment

$$\big(\{\textit{read-img}^{[100,19000]}, \textit{read-gl-img}^{[10000,2000000]}\}, \wedge, \textit{Shuffled}\big)$$

states that the IPU may perform read image (*read-img*) and read image gallery (*read-gl-img*) operations several number of times defined by the respective ranges, and the order of those read operations does not matter.

For sake of brevity, if fragments are not shuffled (i.e., $\sqcup\!\sqcup = \textit{Non-Shuffled}$), below we simply denote them as $\mathcal{F} = \big(\{\mathcal{R}_1, \ldots, \mathcal{R}_m\}, \sharp\big)$.

Definition 10: Loose-Ordering — A loose-ordering $\mathcal{L} = \mathcal{F}_1 < \cdots < \mathcal{F}_\ell$ is made of sequences $s_1, \ldots s_\ell$ matching the corresponding fragments. All the s_is should appear, concatenated in this exact order. *Notice we call it loose-ordering because the order inside fragments is free.*

Example 4.4.1. Loose-Ordering – Consider the loose-ordering

$$\ell = \big(\{n_1, n_2\}, \wedge\big) < n_3^{[2,8]} < \big(\{n_4, n_5\}, \vee\big).$$

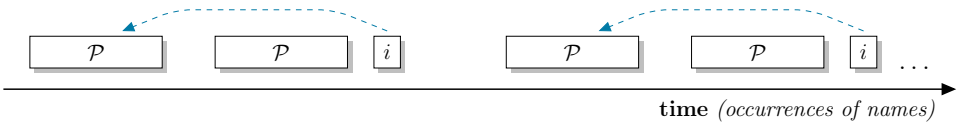
It defines sequences such that: n_1 **and** n_2 come first in any order (i.e., n_1 followed by n_2 or the reverse); then we have several n_3 in a row (the number of occurrences of n_3 is in $[2, 8]$); then we have either n_4 **or** n_5, **or** both in any order. Thus, a sequence $n_2 n_1 n_3 n_3 n_5$ is compliant with the loose-ordering ℓ; the underlined letter in a sequence $n_2 \underline{n_3} n_4$ violates ℓ since it occurs "too soon", before the first fragment $\big(\{n_1, n_2\}, \wedge\big)$ has happened. □

4.4.3 An Antecedent Requirement

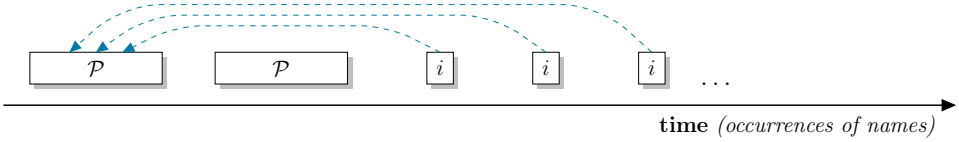
The antecedent requirement defines how a component should be activated. In other words, it states an *assumption* of the component about its environment.

Definition 11: Antecedent requirement — An antecedent requirement $\mathcal{A} = \big(\mathcal{P} \ll i \mid \nabla\big)$, $\nabla \in \{\textit{Non-Repeated}, \textit{Repeated}\}$, means that i can occur only if $\mathcal{P}$ has been observed before. When $\nabla = \textit{Repeated}$ the condition has to be repeated: each occurrence of i should be preceded by its "own" occurrence of $\mathcal{P}$, i.e. an occurrence that happened since the last i (see Fig. 4.7(a)). When $\nabla = \textit{Non-Repeated}$ one occurrence of $\mathcal{P}$ is enough to validate all the further occurrences of i (Fig. 4.7(b)).

Notice, the loose-ordering $\mathcal{P}$ can occur (potentially) several times before the respective occurrence of i. In this case, if $\mathcal{A}$ has a non-repeated context ($\nabla = \textit{Non-Repeated}$), the very first occurrence of $\mathcal{P}$ is



(a) A repeated context $\nabla = \mathit{Repeated}$



(b) A non-repeated context $\nabla = \mathit{Non\text{-}Repeated}$

Figure 4.8 Elaborated semantics of $\mathcal{A} = (\mathcal{P} \ll i \mid \nabla)$.

taken into account (see Fig. 4.8(b)); if $\mathcal{A}$ has a repeated context ($\nabla = \mathit{Repeated}$), the very first occurrence of $\mathcal{P}$ since the last i is taken into account (see Fig. 4.8(a)).

As it is shown in Table 4.1, the name i is an input ($i \in I$) and a vocabulary of the loose-ordering $\mathcal{P}$ is such that $\alpha(\mathcal{P}) \cap i = \varnothing$. Notice, names of $\alpha(\mathcal{P})$ may also occur before, after or between occurrences of $\mathcal{P}$ and i. The loose-ordering $\mathcal{P}$ may depend on *outputs* of a specified component (e.g., see Examples 4.4.4 and 4.4.6 below). We give below examples of the property with various contexts. The examples are inspired by our running example. The full list of defined antecedent properties for the components of the intercom system can be found in Section 3.4 of Chapter 3. In Chapter 7 it is shown how the definition of those properties helps detect and localize the synchronization bugs listed in Section 3.5.

Example 4.4.2. Simple Non-repeated Requirement – Consider the IPU component with the input/output interface shown in Figure 4.6. Any of the component's read-write data registers (e.g., `img-add`) can be read only after its value has been set (written) at least once (and there is no need for a repetition). Recall, the input *set-img-addr* (resp. the output *get-img-addr*) of the IPU's input/output interface corresponds to "read" (resp. "write") of the register `img-addr`. Thus, the stated above property can be formulated as an antecedent requirement:

$$\big(\mathit{set\text{-}img\text{-}addr} \ll \mathit{get\text{-}img\text{-}addr} \;\mid\; \mathit{Non\text{-}Repeated}\big)$$

□

Example 4.4.3. Simple Repeated Requirement – To ensure that the user gets correct the result of face recognition, one needs to provide the new image address to the IPU before *each start* of the recognition analysis:

$$\big(\mathit{set\text{-}img\text{-}addr} \ll \mathit{start} \;\mid\; \mathit{Repeated}\big)$$

□

Example 4.4.4. Simple Repeated Requirement – The confidence value c, which is the result of face recognition, stored in the "read-only" register `conf-val`, can be read (the input *get-conf-val*) only after the respective face recognition terminates. Face recognition terminates when the IPU sends an interrupt (i.e., produces the output *set-irq-pos*). Thus, the property is formulated as an antecedent requirement with a repeated context:

$$\big(\mathit{set\text{-}irq\text{-}pos} \ll \mathit{get\text{-}conf\text{-}val} \;\mid\; \mathit{Repeated}\big)$$

This property ensures that the environment using the IPU gets the *relevant* value of c. Notice, in this case the right part of the property depends on the output of the component. Also notice that such formulation of the property means that the result can be read only once. □

Example 4.4.5. Non-Repeated Requirement with a Conjunction – Before face recognition is started, the IPU needs to be provided *at least once* with the address of the image to be analyzed, the size of the image, the size of the gallery, the address of the gallery, i.e., the values of the respective data registers should be defined. The property is an antecedent requirement:

$$\Big(\big(\{\mathit{set\text{-}img\text{-}add}, \mathit{set\text{-}img\text{-}size}, \mathit{set\text{-}gl\text{-}size}, \mathit{set\text{-}gl\text{-}addr}\}, \wedge\big) \ll \mathit{start} \;\mid\; \mathit{Non\text{-}Repeated}\Big).$$

□

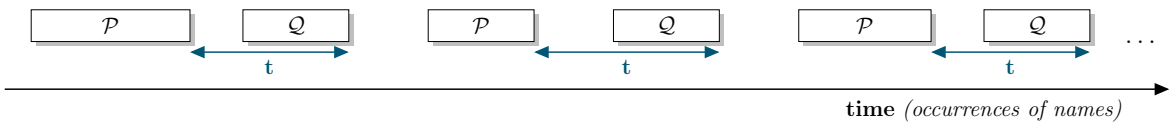


Figure 4.9 Semantics of a timed implication constraint $\mathcal{T} = (\mathcal{P} \Longrightarrow \mathcal{Q} \mid t)$.

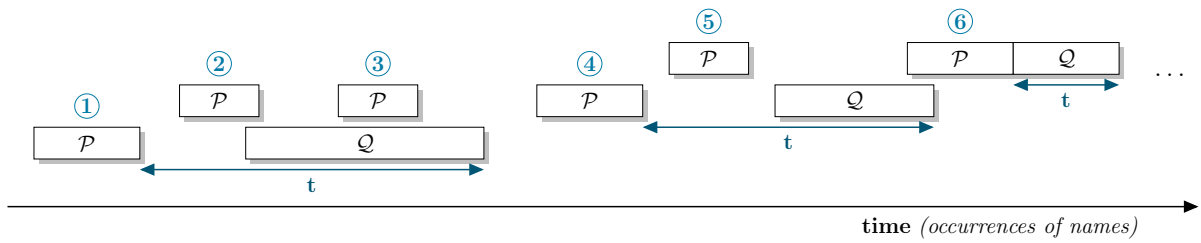


Figure 4.10 Ignoring occurrences of $\mathcal{P}$: the second, the third and the fifth occurrences of $\mathcal{P}$ are ignored because they overlap with $\mathcal{Q}$.

Example 4.4.6. Non-repeated Requirement with a Disjunction – The CPU can get interrupts from the LCDC, the image sensor SEN, the GPIO, the IPU and the timer TMR1. To start getting interrupts, the CPU has to activate *at least one* of those components before. The constraint is of the form:

$$\Big(\big(\{\textit{start-LCDC}, \textit{act-shttr-SEN}, \textit{start-GPIO}, \textit{start-IPU}, \textit{start-TMR1}\}, \vee\big) \\ \ll \textit{set-irq-pos} \mid \textit{Non-Repeated}\Big), \tag{4.1}$$

where the *start-X*s and *act-shttr-SEN* are the outputs of the CPU, and *set-irq-pos* is one of its inputs[5]. □

4.4.4 A Timed Implication Constraint

Properties of the "timed implication constraint" type define obligations (guarantees) of a component, which must be fulfilled when certain conditions hold. The pattern is adaptation of the suffix implication operator available in hardware specification languages (see Sec. 2.3.2.5) to loose-orderings.

Definition 12: Timed implication constraint — A timed implication constraint $\mathcal{T} = (\mathcal{P} \Longrightarrow \mathcal{Q} \mid t)$ means that, whenever $\mathcal{P}$ has been observed, $\mathcal{Q}$ should occur, and should have finished before t time units have elapsed since the end of $\mathcal{P}$ (see Fig. 4.9). This pattern is implicitly of the "repeated" kind: when $\mathcal{P}$ has been observed, $\mathcal{Q}$ should occur, and if a new occurrence of $\mathcal{P}$ is observed, a new occurrence of $\mathcal{Q}$ should occur.

The loose-ordering $\mathcal{P}$ can be seen as a *condition* which triggers the production of a component's outputs with respect to the loose-ordering $\mathcal{Q}$. Since both $\mathcal{P}$ and $\mathcal{Q}$ define sequences of names, we need to define semantics of the pattern when occurrences of $\mathcal{P}$ and $\mathcal{Q}$ *overlap*. Overlapping may occur due to shared names of $\mathcal{P}$ and $\mathcal{Q}$ (see Table 4.1), and/or because of names of $\mathcal{P}$ which may happen while $\mathcal{Q}$ is taking place. We decide that occurrences of $\mathcal{P}$ which take place either while occurrence of $\mathcal{Q}$ (e.g., see 2 and 3 in Fig. 4.10), or in between another occurrence of $\mathcal{P}$ and the respective occurrence of $\mathcal{Q}$ (e.g., see 5) are ignored. That is, $\mathcal{Q}$ cannot be restarted unless it terminates.

We give below examples of timed implication constraints defined for the IPU and the CPU (the embedded software). The full list of the properties is provided in Section 3.4.

Example 4.4.7. – If face recognition starts properly with the defined image address, the IPU reads the analyzed image and images from the external gallery, and then sends an interrupt (the positive edge of the interrupt followed by the negative edge). All outputs must be produced within 500 nanoseconds; it models the duration taken by face recognition. The timed implication constraint specifying the behavior

[5]Here, *set-irq-pos* is the input of the CPU, because it represents interrupts the CPU *gets* from other components. One could notice that the IPU has the output of the same name (see Example 4.4.4). For the IPU the output *set-irq-pos* represents interrupts *sent* by the component itself. For the full list of inputs/outputs of the components of the intercom system see Section 3.4.

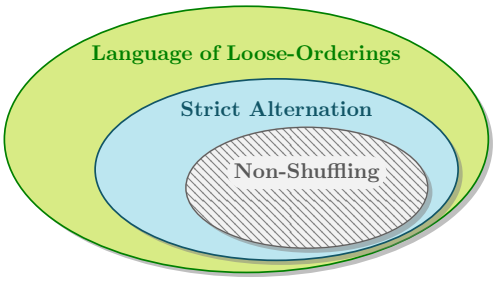


Figure 4.11 The subset of the loose-ordering language encoded into SERE and LTL (the lined area).

of the IPU is the following:

$$\Big(\textit{set-img-addr} < \textit{start} \Longrightarrow \big(\{\textit{read-img}^{[100,19000]}, \textit{read-gl-img}^{[10000,2000000]}\}, \wedge, \textit{Shuffled}\big) < \textit{set-irq-pos} < \textit{set-irq-neg} \mid 500ns\Big)$$

□

Example 4.4.8. – When the system is launched and the CPU is started (the input *start*), the embedded software configures the timer TMR1 and the LCDC with the time scale (the output *set-t-scale-TMR1*) and the address of the buffer (the output *set-buff-addr-LCDC*) respectively, and reads the address of the SEN's buffer (the output *get-buff-addr-SEN*). Then the software starts the image sensor, the LCDC and the GPIO in any order. The described initialization should not take more that 200 nanoseconds:

$$\Big(\textit{start} \Longrightarrow \big(\{\textit{set-t-scale-TMR1}, \textit{set-buff-addr-LCDC}, \textit{get-buff-addr-SEN}\}, \wedge\big) < \big(\{\textit{act-shttr-SEN}, \textit{start-LCDC}, \textit{start-GPIO}\}, \wedge\big) \mid 200ns\Big)$$

□

4.5 Encoding Loose-Ordering Properties

In this section, we show one possible encoding of loose-ordering properties using SERE and LTL operators. This encoding is used in Chapter 7 in order to compare efficiency of our SystemC monitors for loose-orderings (see Chapter 6) with monitors available for SERE and LTL fragments of PSL [PF08; FP10; BA14].

4.5.1 The Encoded Subset

We provide SERE and LTL encoding for the subset of the loose-ordering language which has the following characteristics:

1. For $\mathcal{A} = \big(\mathcal{P} \ll i \mid \nabla\big)$ (resp. $\mathcal{T} = \big(\mathcal{P} \Longrightarrow \mathcal{Q} \mid t\big)$) *only one occurrence of* $\mathcal{P}$ *can take place before the respective occurrence of* i (resp. $\mathcal{Q}$). Moreover, names of $\alpha(\mathcal{P})$ (resp. $\alpha(\mathcal{Q})$) do not occur before or after occurrences of $\mathcal{P}$ (resp. $\mathcal{Q}$). The syntax of $\mathcal{T}$ is such that the vocabularies of $\mathcal{P}$ and $\mathcal{Q}$ are disjoint, i.e., $\alpha(\mathcal{P}) \cap \alpha(\mathcal{Q}) = \varnothing$. The semantics of $\mathcal{T}$ is such that $\mathcal{P}$ and $\mathcal{Q}$ do not overlap. We refer to this subset as *Strict Alternation* (see Fig. 4.11).

2. Neither $\mathcal{A}$ nor $\mathcal{T}$ has fragments with shuffled semantics $\sqcup\!\sqcup = \textit{Shuffled}$. In Figure 4.11 this subset is referred to as *Non-Shuffling*.

The defined subset of the loose-ordering language has been chosen because its encoding into SERE and LTL is relatively easy and leads to the minimum explosion of the size of obtained formulas (Sec. 4.5.2) or their vocabularies (Sec. 4.5.3). Moreover, the subset has (very) intuitive semantics, thus, it is easy to check correctness of the encodings. Although we carefully examine only loose-ordering with non-shuffled fragments, here we also provide the idea of the encoding of fragments with shuffled semantics into SERE and LTL. Notice, we do not prove formally the correctness of the proposed encoding. Our goal is to show that encoding of order non-determinism into specification languages based on regular expressions and linear temporal logic is not very intuitive, and due to that can be hard.

4.5.2 Encoding Using SERE Operators

Consider the antecedent requirement $\mathcal{A} = \big(\mathcal{P} \ll i \mid \nabla\big)$ and the timed implication constraint $\mathcal{T} = \big(\mathcal{P} \Longrightarrow \mathcal{Q} \mid t\big)$. The simplest translation of $\mathcal{A}$ and $\mathcal{T}$ consists in enumerating all total orders compatible with the loose-orderings $\mathcal{P}$ and $\mathcal{Q}$ as it is defined in Section 4.4.2.

4.5.2.1 Encoding Loose-Orderings

Range A range $\mathcal{R} = n^{[u,v]}$ is encoded by the explicit enumeration of all possible sequences of n:

$$\{\ n[*u] \mid \ \cdots \mid \ n[*v]\ \} \tag{4.2}$$

Here, "$[*i]$" is the *bounded ;-iteration* operator, and "$|$" is the *union* operator (see definitions in Sec. 2.3.2.4). Notice, $n[*i]$ is a syntactic sugar for

$$\underbrace{n;\ n;\ \ldots;\ n}_{i \text{ times}}$$

("$;$" stands for *concatenation*). A range $\mathcal{R} = n^{[u,v]}$ defines $(v - u + 1)$ sequences of n.

Fragment The encoding of a fragment $\mathcal{F} = \big(\{\mathcal{R}_1, \ldots, \mathcal{R}_m\}, \sharp, \textit{Non-Shuffled}\big)$ depends on $\sharp$. If $\sharp = \wedge$ one needs to enumerate all possible permutations of the length m of all sequences defined by the ranges $\mathcal{R}_j$s. For instance, for a fragment with two ranges $\mathcal{F} = \big(\{\mathcal{R}_1, \mathcal{R}_2\}, \wedge, \textit{Non-Shuffled}\big)$ the encoding is of the form

$$\begin{aligned}\big\{&\{\text{sequences of } \mathcal{R}_1\};\ \{\text{sequences of } \mathcal{R}_2\} \mid \\ &\{\text{sequences of } \mathcal{R}_2\};\ \{\text{sequences of } \mathcal{R}_1\}\big\}\end{aligned}$$

Here, "$\{\text{sequences of } \mathcal{R}_i\};\ \{\text{sequences of } \mathcal{R}_j\}$" is a Cartesian product of sets, i.e., all sequences of $\mathcal{R}_i$ are concatenated with all sequences of $\mathcal{R}_j$.

If $\sharp = \vee$ one needs to enumerate all possible permutations of any length in $[1, m]$ of all sequences defined by the ranges $\mathcal{R}_j$s. For example, a fragment $\mathcal{F} = \big(\{\mathcal{R}_1, \mathcal{R}_2\}, \vee, \textit{Non-Shuffled}\big)$ can be represented as

$$\begin{aligned}\big\{&\{\text{sequences of } \mathcal{R}_1\} \mid \\ &\{\text{sequences of } \mathcal{R}_2\} \mid \\ &\{\text{sequences of } \mathcal{R}_1\};\ \{\text{sequences of } \mathcal{R}_2\} \mid \\ &\{\text{sequences of } \mathcal{R}_2\};\ \{\text{sequences of } \mathcal{R}_1\}\big\}\end{aligned}$$

Let $\mathcal{F} = \big(\{\mathcal{R}_1, \ldots, \mathcal{R}_m\}, \sharp, \textit{Non-Shuffled}\big)$, and for any $j \in [1, m]$, $\mathcal{R}_j = n^{[u_j, v_j]}$. The total number of sequences defined by $\mathcal{F}$ when $\sharp = \wedge$ is defined by Formula 4.3:

$$m! \prod_{j=1}^{m} (v_j - u_j + 1) \tag{4.3}$$

Here, $m!$ defines the number of possible permutations of the $\mathcal{R}_j$s, and the product defines the number of sequences obtained by concatenating all sequences of the $\mathcal{R}_j$s in one specific order, i.e., a Cartesian product of sets. The total number of sequences compatible with a fragment $\mathcal{F}$ of the disjunctive semantics ($\sharp = \vee$) is defined by Formula 4.4:

$$\sum_{k=1}^{m} \left[\frac{m!}{(m-p)!} \prod_{j=1}^{p} (v_j^p - u_j^p + 1) \right] \tag{4.4}$$

Here, the fraction defines the number of permutations of p *among* m *ranges* of the fragment $\mathcal{F}$, and v_j^p, u_j^p are parameters of those p ranges, for all $j \in [1, p]$. (The formula $\frac{m!}{(m-p)!}$ defines *p-permutations of m elements* [Fel68].)

Loose-Ordering The encoding of a loose-ordering $\mathcal{L} = \mathcal{F}_1 < \cdots < \mathcal{F}_\ell$ is the set of all sequences made by concatenating all sequences of all the fragments $\mathcal{F}_k$s in the order defined by $\mathcal{L}$:

$$\big\{\{\text{sequences of } \mathcal{F}_1\};\ \{\text{sequences of } \mathcal{F}_2\};\ \ldots;\ \{\text{sequences of } \mathcal{F}_\ell\}\big\}$$

As before, "$\{$sequences of $\mathcal{F}_k\}$; $\{$sequences of $\mathcal{F}_{k+1}\}$" defines a Cartesian product of the respective sets. Let for all $k \in [1, \ell]$, $\mathcal{F}_k = \big(\{\mathcal{R}^k_1, \ldots, \mathcal{R}^k_{m^k}\}, \sharp, \textit{Non-Shuffled}\big)$, and for any $j \in [1, m^k]$, $\mathcal{R}^k_j = n^{[u^k_j, v^k_j]}$. When all the $\mathcal{F}_k$s have the conjunctive semantics ($\sharp = \wedge$), the total number of sequences defined by the loose-ordering $\mathcal{L}$ is computed with Formula 4.5:

$$\prod_{k=1}^{\ell} \left[m^k! \prod_{j=1}^{m^k} (v^k_j - u^k_j + 1) \right] \tag{4.5}$$

Here, the component in parentheses is the number of sequences of a fragment $\mathcal{F}_k$ (for any $k \in [1, \ell]$) with $\sharp = \wedge$ (see Formula 4.3). This component should be replaced by Formula 4.4, to compute the number of sequences of the loose-ordering $\mathcal{L}$ with all the $\mathcal{F}_k$s having $\sharp = \vee$.

Example 4.5.1. – Consider a loose-ordering $\ell = \big(\{n_1^{[1,2]}, n_2^{[3,4]}\}, \wedge, \textit{Non-Shuffled}\big) < n_3$. It is made of two fragments. The encoding of the first fragment $\big(\{n_1^{[1,2]}, n_2^{[3,4]}\}, \wedge, \textit{Non-Shuffled}\big)$ is the SERE Formula 4.6:

$$\begin{aligned} \{\ & n_1[*1]; n_2[*3]\ \mid n_1[*1]; n_2[*4]\ \mid n_1[*2]; n_2[*3]\ \mid n_1[*2]; n_2[*4]\ \mid \\ & n_2[*3]; n_1[*1]\ \mid\ n_2[*3]; n_1[*2]\ \mid\ n_2[*4]; n_1[*1]\ \mid\ n_2[*4]; n_1[*2]\ \} \end{aligned} \tag{4.6}$$

The encoding of the second fragment n_3 is trivial. The encoding of the loose-ordering of ℓ is obtained by concatenating all sequences defined by the SERE Formula 4.6 with n_3 (e.g., $\{n_1[*1]; n_2[*3]; n_3\}$). □

4.5.2.2 Encoding Loose-Ordering Patterns

Consider the antecedent requirement $\mathcal{A} = \big(\mathcal{P} \ll i \mid \nabla\big)$. We encode the loose-ordering $\mathcal{P}$ as it is described in Section 4.5.2.1 above. When $\mathcal{A}$ has a non-repeated context ($\nabla = \textit{Non-Repeated}$), the encoding of the pattern is obtained by concatenating all sequences of $\mathcal{P}$ with i:

$$\big\{\{\text{sequences of } \mathcal{P}\};\ \{i\}\big\}$$

To encode the antecedent requirement with a repeated context ($\nabla = \textit{Repeated}$), one may use a *Kleene star*:

$$\big\{\{\text{sequences of } \mathcal{P}\};\ \{i\}\big\}[*]$$

It reflects our assumption that only one $\mathcal{P}$ occurs before respective occurrence of i (i.e., $\mathcal{P}$ and i alternate starting from $\mathcal{P}$). The set of sequences of the timed implication constraint $\mathcal{T} = \big(\mathcal{P} \Longrightarrow \mathcal{Q} \mid t\big)$ is defined in a similar way. One needs to concatenate all sequences of $\mathcal{P}$ with all sequences of $\mathcal{Q}$ in this order, and apply a Kleene star:

$$\big\{\{\text{sequences of } \mathcal{P}\};\ \{\text{sequences of } \mathcal{Q}\}\big\}[*]$$

Here, the Kleene star encodes alternation of $\mathcal{P}$ and $\mathcal{Q}$ starting from $\mathcal{P}$. The encoding is possible due to the assumptions we made in Section 4.5. Notice that we do not encode t.

4.5.2.3 The Number of Operators in SERE Encodings

Since the proposed encoding explicitly enumerates all total orders, the use of regular expressions causes a *combinatorial explosion* of the *length* (number of operators) of the SERE formula. The number of SERE operators needed to encode the antecedent requirement $\mathcal{A} = \big(\mathcal{P} \ll i \mid \nabla\big)$ (resp. the timed implication constraint $\mathcal{T} = \big(\mathcal{P} \Longrightarrow \mathcal{Q} \mid t\big)$) is proportional to the number of sequences compatible with $\mathcal{A}$ (resp. $\mathcal{T}$). Thus, when the loose-ordering $\mathcal{P}$ of $\mathcal{A}$ is such that $\mathcal{P} = \mathcal{F}_1 < \cdots < \mathcal{F}_\ell$, for all $k \in [1, \ell]$ $\mathcal{F}_k = \big(\{\mathcal{R}^k_1, \ldots, \mathcal{R}^k_{m^k}\}, \wedge, \textit{Non-Shuffled}\big)$, and for all $j \in [1, m^k]$, $\mathcal{R}^k_j = n^{[u_j, v_j]}$ the number of operators is

approximately equal to Formula 4.7.

$$\prod_{k=1}^{\ell}\left[m^k!\prod_{j=1}^{m^k}(v_j^k-u_j^k+1)\right] \tag{4.7}$$

EXAMPLE 4.5.2. – To get an intuition about the number of operators defined by Formula 4.7, consider a loose orderings

$$\ell = n_1^{[100,19000]} < n_2^{[100,19000]} < \big(\{n_3,n_4,n_5,n_6\},\wedge,\textit{Non-Shuffled}\big) < n_7.$$

It is inspired by the specification of the CPU (see the property T8-CPU in Sec. 3.4.1.2). The length of the encoding of ℓ into SERE is approximately equal to:

$$\approx (19000-99)\times(19000-99)\times 4!\times 1 = 18901^2\times 24 = \mathbf{8.573.947.224} \text{ SERE operators.}$$

□

4.5.2.4 Remark: When Ranges are Shuffled

In Section 4.5.2.3 we provided the approximate length of the SERE encoding for a loose-ordering $\mathcal{L} = \mathcal{F}_1 < \cdots < \mathcal{F}_\ell$ under assumption that ranges of the fragments $\mathcal{F}_k$s are not shuffled. The encoding of a fragment $\mathcal{F} = \big(\{\mathcal{R}_1,\ldots,\mathcal{R}_m\},\sharp,\textit{Shuffled}\big)$ with shuffled ranges and the conjunctive semantics $\sharp = \wedge$ (resp. the disjunctive semantics $\sharp = \vee$) is the set of all sequences obtained by (i) concatenating all sequences of all (resp. any $p \in [1,m]$) ranges $\mathcal{R}_j$s; (ii) enumerating all (different) permutations of the letters for all those sequences. Obviously, shuffling of ranges leads to more explosive SERE encoding than the encoding defined in Section 4.5.2.3. Although shuffle of regular languages is defined [Hop+00; EPH06], to the best of our knowledge the respective constructs are not supported by any of the existing specification languages.

EXAMPLE 4.5.3. – Consider a shuffled fragment $\big(\{n_1,n_2^3\},\wedge,\textit{Shuffled}\big)$. It defines that exactly one occurrence of n_1 and exactly three occurrences of n_2 should occur; the order of occurrences is not fixed. To encode the fragment into SERE, one needs to enumerate all possible permutations of the occurrences of n_1 and n_2:

$$\big\{n_1;n_2;n_2;n_2 \;\mid\; n_2;n_1;n_2;n_2 \;\mid\; n_2;n_2;n_1;n_2 \;\mid\; n_2;n_2;n_2;n_1\big\}.$$

□

4.5.3 Encoding Using LTL Operators

The encodings of the antecedent requirement $\mathcal{A} = \big(\mathcal{P} \ll i \mid \nabla\big)$ and the timed implication constraint $\mathcal{T} = \big(\mathcal{P} \Longrightarrow \mathcal{Q} \mid t\big)$ into LTL are *Boolean combinations of LTL formulas*. We use LTL operators *always* "$\Box$", *until* "$\mathcal{U}$", *next* "$\bigcirc$" and Boolean operators "$\wedge$", "$\vee$", "$\neg$", "$\rightarrow$" (see definitions in Sec. 2.3.2). The purpose of this section is to show that it is feasible to express loose-orderings in temporal logic, although this task is not trivial. Since conjunction and nesting of temporal formulas is not very intuitive, we tested our LTL encodings by (i) translating them into equivalent Büchi automata, (ii) translating obtained Büchi automata into monitors of the respective LTL formulas. Both steps were performed using the SPOT tool (see below).

4.5.3.1 Background Note: SPOT Tool

SPOT is a C++11 library for LTL, ω-automata manipulation and model checking [DLP04; DL+16]. ω-automata recognize *infinite* sequences. They have states and transitions. A run of the automaton is accepting if it satisfies the automaton's *acceptance condition*. The simplest kind of ω-automata is Büchi automata. Their acceptance condition states that a run is accepting if and only if it visits some *accepting state* infinitely often. For instance, Figure 4.12 shows a Büchi automaton as it is represented in SPOT. The automaton accepts sequences such that at least two consecutive occurrences of $b\wedge\neg a$ happen infinitely often. Its labels are Boolean formulas.

SPOT translates temporal formulas into equivalent Büchi automata. Recall, a temporal formula is constructed by means of temporal operators (*always* "$\Box$", *eventually* "$\Diamond$", etc.), Boolean operators (*conjunction* "$\wedge$", *disjunction* "$\vee$", etc.), SERE operators (*concatenation* ";", *union* "|", etc.) and SERE-LTL operators (*suffix implications* "$\mapsto$" and "$\Mapsto$"). Their definition is provided in Section 2.3.2 of the

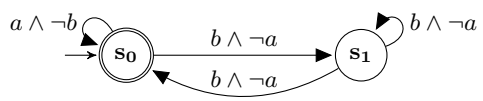


Figure 4.12 Example of Büchi automaton defined in SPOT. A state marked with an arrow is initial, a double-circled state is accepting. Labels are Boolean formulas.

background chapter. The translation performed by SPOT relies on the results of [McN06; LP85; Var96; Cou99]. When a Büchi automaton for a formula ϕ is obtained, the tool may produce a *monitor* of ϕ, i.e., a finite state machine which accepts all finite prefixes compatible with ϕ. To define temporal formulas, SPOT provides its own syntax [DL16]. The SPOT's operators `G`, `F`, `U`, `W` and `X` correspond to the PSL's operators `always`, `eventually!`, `until!`, `until` and `next!` respectively. The syntax of other operators is as it is defined for PSL[6].

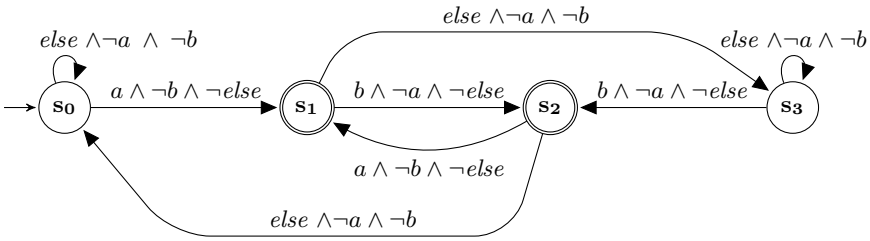
Example 4.5.4. – Consider the LTL formula

$$\phi = \Box\big(\neg(a \wedge b)\big) \wedge \Box\big(a \rightarrow \bigcirc(\neg a \,\mathcal{U}\, b)\big) \wedge \Box\big(b \rightarrow \bigcirc(\neg b \,\mathcal{U}\, a)\big) \wedge \big(\neg b \,\mathcal{U}\, a\big)$$

It defines that (i) a and b never occur simultaneously, (ii) a and b alternate starting from a. Figure 4.13 provides ① definition of ϕ in the SPOT syntax, ② a Büchi automaton equivalent to ϕ and ③ a corresponding SPOT monitor. The monitor ③ recognizes finite prefixes of sequences which satisfy ϕ. It has semantics of a recognizer for a regular language, i.e., its missing transitions are interpreted as errors. For instance, occurrence of a when the monitor is in s_1 is an error. □

① `G(!(a & b)) & G(a -> X(!a U b)) & G(b -> X(!b U a)) & (!b U a)`

②



③

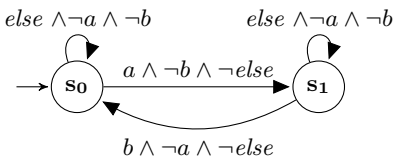


Figure 4.13 Example of a Büchi automaton and a monitor produced by SPOT for LTL formula. *else* stands for any name which is not in $\{a, b\}$.

4.5.3.2 Encoding Loose-Ordering Patterns

Consider the antecedent requirement $\mathcal{A} = \big(\mathcal{P} \ll i \mid \nabla\big)$ and the timed implication constraint $\mathcal{T} = \big(\mathcal{P} \Longrightarrow \mathcal{Q} \mid t\big)$. Let $\mathcal{P}$ and $\mathcal{Q}$ be $\mathcal{F}_1 < \cdots < \mathcal{F}_p$. Recall, we encode the subset of the loose-ordering language which (i) satisfies the condition of the strict alternation of $\mathcal{P}$ and i for $\mathcal{A}$, and $\mathcal{P}$ and $\mathcal{Q}$ for $\mathcal{T}$ as defined in Section 4.5.1, (ii) does not contain fragments with shuffled semantics. The considered subset of the loose-ordering language allows us to consider both $\mathcal{A} = \big(\mathcal{P} \ll i \mid \nabla\big)$ and $\mathcal{T} = \big(\mathcal{P} \Longrightarrow \mathcal{Q} \mid t\big)$ as a loose-ordering $\mathcal{L} = \mathcal{F}_1 < \cdots < \mathcal{F}_\ell < \mathcal{F}'$ such that the last marked fragment is $\mathcal{F}' = i$ for $\mathcal{A}$, and $\mathcal{F}'$ coincides with the last fragment of $\mathcal{Q}$ for $\mathcal{T}$. Repeating $\mathcal{L}$ we reproduce the repeated nature of $\mathcal{A}$ and $\mathcal{T}$ respectively. The encoding of the antecedent requirement $\mathcal{A}$ with a non-repeated context is the encoding of one occurrence of $\mathcal{L}$.

[6]The list of PSL operators is provided in Table 2.1 of Section 2.3.2.

The idea of expressing loose-orderings in LTL is the following. To encode $\mathcal{A}$ and $\mathcal{T}$ of the repeated kind, we consider the last fragment $\mathcal{F}'$ of $\mathcal{L} = \mathcal{F}_1 < \cdots < \mathcal{F}_\ell < \mathcal{F}'$ as a *resetting point*: (i) $\mathcal{F}'$ should alternate with occurrences of all other fragments $\mathcal{F}_k$s which take place as it is specified by $\mathcal{L}$, (ii) none of the $\mathcal{F}_k$s can occur twice unless occurrence of $\mathcal{F}'$. To encode $\mathcal{A}$ with a non-repeated context we consider $\mathcal{F}'$ as a *delimiter*: only before $\mathcal{F}'$ all the $\mathcal{F}_k$s should occur in the specified order, after $\mathcal{F}'$ occurrences of the names of the $\mathcal{F}_k$s are unrestricted. To encode order of the $\mathcal{F}_k$s, we define LTL formulas which constrain occurrences of *each name* of the fragments, and take conjunction of those formulas.

Dealing with Ranges LTL lacks counting facilities [Wol81]. Provided that ranges are not shuffled, one way to encode ranges is to use a big disjunction of nested "next" operators encoding all sequences defined by ranges. For instance, the encoding of the range $n^{[2,4]}$ defining sequences of consecutive occurrences of the name n of the length $2, 3$ or 4 into temporal logic by means of nested next operators "$\bigcirc$" can be done as follows:

$$\underbrace{\big(n \wedge \bigcirc n\big)}_{\text{2 occurrences}} \vee \underbrace{\big(n \wedge \bigcirc(n \wedge \bigcirc n)\big)}_{\text{3 occurrences}} \vee \underbrace{\big(n \wedge \bigcirc\big(n \wedge \bigcirc(n \wedge \bigcirc n)\big)\big)}_{\text{4 occurrences}}$$

This encoding is as explosive as the encoding into regular expressions since all possible sequences are expressed explicitly (see Sec. 4.5.2). Moreover, it is hard to ensure that such LTL formulas have their intended semantics, specially when conjunction of several of them is taken.

We propose an alternative approach to deal with ranges. The approach relies on the use of a *lexical analyzer* which treats different sequences of consecutive occurrences of a range's name as new elements. For instance, the range $n^{[1,2]}$ defines sequences n and nn, i.e., $n^{[1,2]} = \{n, nn\}$. Let n^1 stands for the sequence n, and let n^2 stands for the sequence nn. When the lexical analyzer detects n (resp. nn) it returns the name n^1 (resp. n^2). Thus, $n^{[1,2]} = \{n^1, n^2\}$. In the sequel we use *renaming of sequences* such that a sequence of p occurrences of n corresponds to a new name n^p. Thus, the new vocabulary of $n^{[u,v]}$ is $\hat{\alpha} = \{n^u, \ldots, n^v\}$ (instead of $\alpha = \{n\}$), and $n^{[u,v]} = \{n^u, \ldots, n^v\}$.

The unrolling of ranges causes *explosion of the size of vocabulary*. For instance, a new vocabulary of a range $n^{[100,19000]}$ after introduction of new names is of the size $19000 - 100 + 1 = \mathbf{18901}$. In the general case, when a range is $\mathcal{R} = n^{[u,v]}$, the size of a new vocabulary $\hat{\alpha}(\mathcal{R})$ is defined by the formula $|\hat{\alpha}(\mathcal{R})| = v - u + 1$.

Occurrences of names of $\hat{\alpha}(\mathcal{R})$ are mutually exclusive: if a sequence of 5 occurrences of n takes place (i.e, a name $n^5 \in \hat{\alpha}(\mathcal{R})$ occurs), neither a sequence of 4 occurrences ($n^4 \in \hat{\alpha}$), nor of 6 occurrences ($n^6 \in \hat{\alpha}$) can take place.

Encoding Repeated Loose-Ordering We consider a loose-ordering $\mathcal{L} = \mathcal{F}_1 < \cdots < \mathcal{F}_\ell < \mathcal{F}'$ such that $\forall k \in [1, \ell], \mathcal{F}_k = \big(\{\mathcal{R}^k_1, \ldots, \mathcal{R}^k_{m^k}\}, \sharp, \textit{Non-Shuffled}\big)$ and $\mathcal{F}' = \big(\{\mathcal{R}'_1, \ldots, \mathcal{R}'_{m'}\}, \sharp, \textit{Non-Shuffled}\big)$. All ranges of the fragments of $\mathcal{L}$ are unrolled as described above. The vocabulary of $\mathcal{L}$ (resp. $\mathcal{F}$, $\mathcal{R}$) is $\hat{\alpha}(\mathcal{L})$ (resp. $\hat{\alpha}(\mathcal{F})$, $\hat{\alpha}(\mathcal{R})$). The encoding of the *repetition* of $\mathcal{L}$ is the conjunction of LTL formulas constraining occurrences of the individual names of $\hat{\alpha}(\mathcal{L})$. The formulas are listed and explained below.

Exclusiveness: due to asynchrony of TL models, none of the names of $\hat{\alpha}(\mathcal{L})$ should occur simultaneously (Formula 4.8). For example, one may express mutual exclusiveness of names a and b by the LTL formula:

$$\Box\big(\neg(a \wedge b)\big),$$

read as "*never a and b at the same time*".

$$\bigwedge_{\forall n^x, n^y \in \hat{\alpha}(\mathcal{L}):\ n^x \neq n^y} \Big(\Box\big(\neg(n^x \wedge n^y)\big)\Big) \tag{4.8}$$

Notice, Exclusiveness is implicitly assumed for all the formulas introduced in the sequel. The following two formulas are constraints on the occurrences of names of ranges of different fragments of $\mathcal{L}$.

Range: only one name of a range $\mathcal{R}^k_j$ (for $k \in [1, \ell]$, for $j \in [1, m^k]$) can occur *before* each occurrence of the fragment $\mathcal{F}'$ (its names). The order relation between the occurrences of names makes sense since we assume that names do not occur simultaneously. If the semantics of $\mathcal{F}'$ is conjunctive ($\sharp = \wedge$), the constraint is encoded by Formula 4.9. If the semantics of $\mathcal{F}'$ is disjunctive ($\sharp = \vee$), the constraint is encoded by Formula 4.10. Formula 4.9 (resp. Formula 4.10) for each *ordered* pair of names n^x, n^y

$(n^x \neq n^y)$ of a range $\mathcal{R}_j^k$ states that always if n^x occurs, then n^y does not occur unless at least one name n^z per range of the fragment $\mathcal{F}'$ (resp. at least one name of the fragment $\mathcal{F}'$) occurs.

$$\bigwedge_{\substack{k\in[1,\ell],\ j\in[1,m^k],\ i\in[1,m'] \\ \forall n^x,n^y\in\hat{\alpha}(\mathcal{R}_j^k):\ n^x\neq n^y}} \Big(\Box\Big(n^x \rightarrow (\bigvee_{\forall n^z\in\hat{\alpha}(\mathcal{R}'_i)} (\neg n^y\ \mathcal{U}\ n^z))\Big)\Big) \tag{4.9}$$

$$\bigwedge_{\substack{k\in[1,\ell],\ j\in[1,m^k] \\ \forall n^x,n^y\in\hat{\alpha}(\mathcal{R}_j^k):\ n^x\neq n^y}} \Big(\Box\Big(n^x \rightarrow (\bigvee_{\substack{i\in[1,m'] \\ \forall n^z\in\hat{\alpha}(\mathcal{R}'_i)}} (\neg n^y\ \mathcal{U}\ n^z))\Big)\Big) \tag{4.10}$$

Range$\mathcal{F}'$: at most one name per range of the reset fragment $\mathcal{F}'$ should occur before all the fragments $\mathcal{F}_k$s (for $k \in [1, \ell]$). The constraint is encoded by Formula 4.11. Here, the first (resp. second) big conjunct is over all $\mathcal{F}_k$s which have the conjunctive (resp. disjunctive) semantics $\sharp = \wedge$ (resp. $\sharp = \vee$). It states that at most one name per range of $\mathcal{F}'$ should occur before the names of *all* ranges (resp. *at least one* range) of $\mathcal{F}_k$s.

$$\begin{aligned} &\bigwedge_{\substack{i\in[1,m'] \\ \forall n^x,n^y\in\hat{\alpha}(\mathcal{R}'_i):\ n^x\neq n^y \\ k\in[1,\ell] \text{ such that for } \mathcal{F}_k\ \sharp=\wedge \\ j\in[1,m^k]}} \Big(\Box\Big(n^x \rightarrow (\bigvee_{\forall n^z\in\hat{\alpha}(\mathcal{R}_j^k)} (\neg n^y\ \mathcal{U}\ n^z))\Big)\Big) \\ \wedge\ &\bigwedge_{\substack{i\in[1,m'] \\ \forall n^x,n^y\in\hat{\alpha}(\mathcal{R}'_i):\ n^x\neq n^y \\ k\in[1,\ell] \text{ such that for } \mathcal{F}_k\ \sharp=\vee}} \Big(\Box\Big(n^x \rightarrow (\bigvee_{\substack{j\in[1,m^k] \\ \forall n^z\in\hat{\alpha}(\mathcal{R}_j^k)}} (\neg n^y\ \mathcal{U}\ n^z))\Big)\Big) \end{aligned} \tag{4.11}$$

Notice, the constraints Range and Range$\mathcal{F}'$ are symmetric.

MaxOne: names of the fragments $\mathcal{F}_k$s can occur *at most once* before each occurrence of the fragment $\mathcal{F}'$ (its names). If $\mathcal{F}'$ has the conjunctive (resp. disjunctive) semantics, then the names of $\mathcal{F}_k$s should occur at most once before occurrences of the names of *all* ranges (resp. *at least one* range) of $\mathcal{F}'$. The constraint is encoded by Formula 4.12 (resp. Formula 4.13).

$$\bigwedge_{\substack{k\in[1,\ell],\ \forall n^x\in\hat{\alpha}(\mathcal{F}_k) \\ i\in[1,m']}} \Big(\Box\Big(n^x \rightarrow \bigcirc(\bigvee_{\forall n^z\in\hat{\alpha}(\mathcal{R}'_i)} (\neg n^x\ \mathcal{U}\ n^z))\Big)\Big) \tag{4.12}$$

$$\bigwedge_{k\in[1,\ell],\ \forall n^x\in\hat{\alpha}(\mathcal{F}_k)} \Big(\Box\Big(n^x \rightarrow \bigcirc(\bigvee_{i\in[1,m'],\ \forall n^z\in\hat{\alpha}(\mathcal{R}'_i)} (\neg n^x\ \mathcal{U}\ n^z))\Big)\Big) \tag{4.13}$$

The following two formulas Order and Order$\mathcal{F}'$ encode the order of fragments of the loose-ordering $\mathcal{L} = \mathcal{F}_1 < \cdots < \mathcal{F}_\ell < \mathcal{F}'$.

Order: if the fragment $\mathcal{F}_k$ has started (i.e., its names occur), the preceding fragment $\mathcal{F}_{k-1}$ has lost its turn and cannot occur unless the resetting fragment $\mathcal{F}'$ occurs. If the semantics of $\mathcal{F}'$ is conjunctive (resp. disjunctive), then the names of $\mathcal{F}_{k-1}$ should not occur unless the names of *all* ranges (resp. *at least one* range) of $\mathcal{F}'$ occur. The constraint is encoded by Formula 4.14 (resp. Formula 4.15).

$$\bigwedge_{\substack{k\in[2,\ell],\ i\in[1,m'] \\ \forall n^x\in\hat{\alpha}(\mathcal{F}_k) \\ \forall n^y\in\hat{\alpha}(\mathcal{F}_{k-1})}} \Big(\Box\Big(n^x \rightarrow (\bigvee_{\forall n^z\in\hat{\alpha}(\mathcal{R}'_i)} (\neg n^y\ \mathcal{U}\ n^z))\Big)\Big) \tag{4.14}$$

$$\bigwedge_{\substack{k\in[2,\ell] \\ \forall n^x\in\hat{\alpha}(\mathcal{F}_k) \\ \forall n^y\in\hat{\alpha}(\mathcal{F}_{k-1})}} \Big(\Box\Big(n^x \rightarrow (\bigvee_{\substack{i\in[1,m'] \\ \forall n^z\in\hat{\alpha}(\mathcal{R}'_i)}} (\neg n^y\ \mathcal{U}\ n^z))\Big)\Big) \tag{4.15}$$

Order$\mathcal{F}'$: if the fragment $\mathcal{F}'$ has started (i.e., its names occur), the preceding fragment $\mathcal{F}_\ell$ has lost its turn and cannot occur unless the fragment $\mathcal{F}_1$ occurs. This constraint is symmetric to Order. Its LTL encoding is defined by Formula 4.16 (resp. Formula 4.17) when the semantics of $\mathcal{F}_1$ is conjunctive (resp. disjunctive). Notice, the proposed encoding can be applied to a loose-ordering which has at least three fragments.

$$\bigwedge_{\substack{\forall n^x \in \hat{\alpha}(\mathcal{F}') \\ \forall n^y \in \hat{\alpha}(\mathcal{F}_\ell) \\ j \in [1, m^1]}} \Big(\Box \Big(n^x \rightarrow \big(\bigvee_{\forall n^z \in \hat{\alpha}(\mathcal{R}^1_j)} (\neg n^y \, \mathcal{U} \, n^z) \big) \Big) \Big) \tag{4.16}$$

$$\bigwedge_{\substack{\forall n^x \in \hat{\alpha}(\mathcal{F}') \\ \forall n^y \in \hat{\alpha}(\mathcal{F}_\ell)}} \Big(\Box \Big(n^x \rightarrow \big(\bigvee_{\substack{j \in [1, m^1] \\ \forall n^z \in \hat{\alpha}(\mathcal{R}^1_j)}} (\neg n^y \, \mathcal{U} \, n^z) \big) \Big) \Big) \tag{4.17}$$

The constraints First$\mathcal{F}'$ and After$\mathcal{F}'$ define the alternation of the ordered fragments $\mathcal{F}_1 < \cdots < \mathcal{F}_\ell$ and the resetting fragment $\mathcal{F}'$.

First$\mathcal{F}'$: the fragment $\mathcal{F}'$ can occur for the first time only after all the fragments $\mathcal{F}_k$s were observed. The constraint is encoded by Formula 4.18. Here, the first big conjunct states that the names of $\mathcal{F}'$ should not occur before the names of *all* ranges of the fragments $\mathcal{F}_k$s with the conjunctive semantics ($\sharp = \wedge$) occur. The second big conjunct states that the names of $\mathcal{F}'$ should not occur before the names of *at least one* range per $\mathcal{F}_k$ with the disjunctive semantics ($\sharp = \vee$) occur.

$$\begin{aligned} & \bigwedge_{\substack{\forall n^x \in \hat{\alpha}(\mathcal{F}') \\ k \in [1,\ell] \text{ such that for } \mathcal{F}_k \ \sharp = \wedge \\ j \in [1, m^k]}} \big(\bigvee_{\forall n^y \in \hat{\alpha}(\mathcal{R}^k_j)} (\neg n^x \, \mathcal{U} \, n^y) \big) \\ \wedge & \bigwedge_{\substack{\forall n^x \in \hat{\alpha}(\mathcal{F}') \\ k \in [1,\ell] \text{ such that for } \mathcal{F}_k \ \sharp = \vee}} \big(\bigvee_{\substack{j \in [1, m^k] \\ \forall n^y \in \hat{\alpha}(\mathcal{R}^k_j)}} (\neg n^x \, \mathcal{U} \, n^y) \big) \end{aligned} \tag{4.18}$$

After$\mathcal{F}'$: the fragments $\mathcal{F}_k$s should be observed before each occurrence of the fragment $\mathcal{F}'$. The constraint is encoded by Formula 4.19. Here, the first big conjunct states that the names of $\mathcal{F}'$, when they occur, should not be repeated unless the names of *all* ranges of $\mathcal{F}_k$s with $\sharp = \wedge$ occur. The second big conjunct states that the names of $\mathcal{F}'$, when they occur, should not be repeated unless the names of *at least one* range per fragment $\mathcal{F}_k$ with the disjunctive semantics ($\sharp = \vee$) occur.

$$\begin{aligned} & \bigwedge_{\substack{\forall n^x \in \hat{\alpha}(\mathcal{F}') \\ k \in [1,\ell] \text{ such that for } \mathcal{F}_k \ \sharp = \wedge \\ j \in [1, m^k]}} \Big(\Box \Big(n^x \rightarrow \bigcirc \big(\bigvee_{\forall n^y \in \hat{\alpha}(\mathcal{R}^k_j)} (\neg n^x \, \mathcal{U} \, n^y) \big) \Big) \Big) \\ \wedge & \bigwedge_{\substack{\forall n^x \in \hat{\alpha}(\mathcal{F}') \\ k \in [1,\ell] \text{ such that for } \mathcal{F}_k \ \sharp = \vee}} \Big(\Box \Big(n^x \rightarrow \bigcirc \big(\bigvee_{\substack{j \in [1, m^k] \\ \forall n^y \in \hat{\alpha}(\mathcal{R}^k_j)}} (\neg n^x \, \mathcal{U} \, n^y) \big) \Big) \Big) \end{aligned} \tag{4.19}$$

The LTL encoding of $\mathcal{A}$ with a repeated context $\nabla = \textit{Repeated}$ is Conjunction 4.20. The LTL encoding of $\mathcal{T}$ is Conjunction 4.21. Here, the conjuncts represent constraints introduced thus far. Notice, for the encoding of $\mathcal{A}$ we do not use conjuncts Range$\mathcal{F}'$ and Order$\mathcal{F}'$ because the fragment $\mathcal{F}' = i$ is trivial.

$$\text{Exclusiveness} \wedge \text{MaxOne} \wedge \text{Range} \wedge \text{Order} \wedge \text{First}\mathcal{F}' \wedge \text{After}\mathcal{F}' \tag{4.20}$$

$$\text{Exclusiveness} \wedge \text{MaxOne} \wedge \text{Range} \wedge \text{Order} \wedge \text{Range}\mathcal{F}' \wedge \text{Order}\mathcal{F}' \wedge \text{First}\mathcal{F}' \wedge \text{After}\mathcal{F}' \tag{4.21}$$

Encoding One Occurrence of Loose-Ordering To encode the antecedent requirement $\mathcal{A} = \big(\mathcal{P} \ll i \mid \textit{Non-Repeated}\big)$ with a non-repeated context into LTL (i.e., to encode one occurrence of $\mathcal{L} = \mathcal{F}_1 < \cdots < \mathcal{F}_\ell < \mathcal{F}'$ with $\mathcal{F}' = i$ being a delimiter), one needs to state that (i) the $\mathcal{F}_k$s should occur only once in a defined order before *the first occurrences* of $\mathcal{F}' = i$; (ii) after the first occurrence of $\mathcal{F}'$ the names of $\mathcal{L}$ can occur in any order. For the names of $\mathcal{L}$ as before we apply Exclusiveness and First$\mathcal{F}'$. Since $\mathcal{L}$ is non-repeated, $\mathcal{F}_k$s are not required to occur before each occurrence of $\mathcal{F}'$, i.e., After$\mathcal{F}'$ is omitted. The constraints Range, MaxOne and Order are modified as explained below.

RangeNR: [7] only one name of a range $\mathcal{R}^k_j$ (for $k \in [1, \ell]$, for $j \in [1, m^k]$) can occur *before the first occurrence* of $\mathcal{F}' = i$, and after the occurrence of i the names of ranges can occur in any order. The constraint is encoded by Formula 4.22.

$$\bigwedge_{\substack{k\in[1,\ell],\ j\in[1,m^k] \\ \forall n^x,n^y\in\hat{\alpha}(\mathcal{R}^k_j):\ n^x\neq n^y}} \Big(\big(n^x \to (\neg n^y\ \mathcal{U}\ i)\big)\ \mathcal{U}\ i\Big) \tag{4.22}$$

MaxOneNR: the fragments $\mathcal{F}_k$s (their names) can occur only once before the first occurrence of $\mathcal{F}' = i$, and after the first occurrence of i they can occur in any order (Formula 4.23).

$$\bigwedge_{k\in[1,\ell],\ \forall n^x\in\hat{\alpha}(\mathcal{F}_k)} \Big(\big(n^x \to \bigcirc(\neg n^x\ \mathcal{U}\ i)\big)\ \mathcal{U}\ i\Big) \tag{4.23}$$

OrderNR: if the fragment $\mathcal{F}_k$ has started (i.e., its names occur), the preceding fragment $\mathcal{F}_{k-1}$ has lost its turn and cannot occur unless the resetting fragment $\mathcal{F}' = i$ occurs for the first time; after the occurrence of i the fragments $\mathcal{F}_k$s can occur in any order (Formula 4.24).

$$\bigwedge_{\substack{k\in[2,\ell] \\ \forall n^x\in\hat{\alpha}(\mathcal{F}_k),\ \forall n^y\in\hat{\alpha}(\mathcal{F}_{k-1})}} \Big(\big(n^x \to (\neg n^y\ \mathcal{U}\ i)\big)\ \mathcal{U}\ i\Big) \tag{4.24}$$

The LTL encoding of $\mathcal{A}$ with a non-repeated context $\nabla = \textit{Non-Repeated}$ is Conjunction 4.25.

$$\textsc{Exclusiveness} \wedge \textsc{MaxOneNR} \wedge \textsc{RangeNR} \wedge \textsc{OrderNR} \wedge \textsc{First}\mathcal{F}' \tag{4.25}$$

Example 4.5.5. LTL Encoding of an Antecedent Requirement – Consider an antecedent requirement

$$\mathcal{A} = \Big(\big(\{a, b^{[1,2]}\}, \wedge, \textit{Non-Shuffled}\big) < \big(\{c, d\}, \vee, \textit{Non-Shuffled}\big) \ll i \ \mid\ \nabla\Big).$$

Before encoding the property into LTL, let remove the range $b^{[1,2]}$ by introducing new names b^1 and b^2. The new vocabulary is $\hat{\alpha}(\mathcal{A}) = \{a, b^1, b^2, c, d, i\}$. The LTL encoding of the property is the conjunction of the following formulas:

$$\begin{aligned}
&\Box\big(\neg(a \wedge b^1)\big) \wedge \Box\big(\neg(a \wedge b^2)\big) \wedge \Box\big(\neg(a \wedge c)\big) \wedge \Box\big(\neg(a \wedge d)\big) \\
&\wedge \Box\big(\neg(a \wedge i)\big) \wedge \Box\big(\neg(b^1 \wedge b^2)\big) \wedge \Box\big(\neg(b^1 \wedge c)\big) \wedge \Box\big(\neg(b^1 \wedge d)\big) \\
&\wedge \Box\big(\neg(b^1 \wedge i)\big) \wedge \Box\big(\neg(b^2 \wedge c)\big) \wedge \Box\big(\neg(b^2 \wedge d)\big) \wedge \Box\big(\neg(b^2 \wedge i)\big) \\
&\wedge \Box\big(\neg(c \wedge d)\big) \wedge \Box\big(\neg(c \wedge i)\big) \wedge \Box\big(\neg(d \wedge i)\big)
\end{aligned} \tag{$\mathcal{A}$-Exclusiveness}$$

$$\begin{aligned}
&\Box\big(a \to \bigcirc(\neg a\ \mathcal{U}\ i)\big) \wedge \Box\big(b^1 \to \bigcirc(\neg b^1\ \mathcal{U}\ i)\big) \wedge \Box\big(b^2 \to \bigcirc(\neg b2\ \mathcal{U}\ i)\big) \\
&\wedge \Box\big(c \to \bigcirc(\neg c\ \mathcal{U}\ i)\big) \wedge \Box\big(d \to \bigcirc(\neg d\ \mathcal{U}\ i)\big)
\end{aligned} \tag{$\mathcal{A}$-MaxOne}$$

[7] "NR" stands for "*Non-Repeated*"

$$\Big((a \to \bigcirc(\neg a \,\mathcal{U}\, i))\,\mathcal{U}\, i\Big) \wedge \Big((b^1 \to \bigcirc(\neg b^1 \,\mathcal{U}\, i))\,\mathcal{U}\, i\Big) \wedge \Big((b^2 \to \bigcirc(\neg b^2 \,\mathcal{U}\, i))\,\mathcal{U}\, i\Big) \wedge \Big((c \to \bigcirc(\neg c \,\mathcal{U}\, i))\,\mathcal{U}\, i\Big) \wedge \Big((d \to \bigcirc(\neg d \,\mathcal{U}\, i))\,\mathcal{U}\, i\Big) \tag{$\mathcal{A}$-MaxOneNR}$$

$$\Box(b^1 \to (\neg b^2 \,\mathcal{U}\, i)) \wedge \Box(b^2 \to (\neg b^1 \,\mathcal{U}\, i)) \tag{$\mathcal{A}$-Range}$$

$$\Big((b^1 \to (\neg b^2 \,\mathcal{U}\, i))\,\mathcal{U}\, i\Big) \wedge \Big((b^2 \to (\neg b^1 \,\mathcal{U}\, i))\,\mathcal{U}\, i\Big) \tag{$\mathcal{A}$-RangeNR}$$

$$\Box(c \to (\neg a \,\mathcal{U}\, i)) \wedge \Box(c \to (\neg b^1 \,\mathcal{U}\, i)) \wedge \Box(c \to (\neg b^2 \,\mathcal{U}\, i)) \wedge \Box(d \to (\neg a \,\mathcal{U}\, i)) \wedge \Box(d \to (\neg b^1 \,\mathcal{U}\, i)) \wedge \Box(d \to (\neg b^2 \,\mathcal{U}\, i)) \tag{$\mathcal{A}$-Order}$$

$$\Big((c \to (\neg a \,\mathcal{U}\, i))\,\mathcal{U}\, i\Big) \wedge \Big((c \to (\neg b^1 \,\mathcal{U}\, i))\,\mathcal{U}\, i\Big) \wedge \Big((c \to (\neg b^2 \,\mathcal{U}\, i))\,\mathcal{U}\, i\Big) \wedge \Big((d \to (\neg a \,\mathcal{U}\, i))\,\mathcal{U}\, i\Big) \wedge \Big((d \to (\neg b^1 \,\mathcal{U}\, i))\,\mathcal{U}\, i\Big) \wedge \Big((d \to (\neg b^2 \,\mathcal{U}\, i))\,\mathcal{U}\, i\Big) \tag{$\mathcal{A}$-OrderNR}$$

$$((\neg i \,\mathcal{U}\, b^1) \vee (\neg i \,\mathcal{U}\, b^2)) \wedge (\neg i \,\mathcal{U}\, a) \wedge ((\neg i \,\mathcal{U}\, c) \vee (\neg i \,\mathcal{U}\, d)) \tag{$\mathcal{A}$-First$\mathcal{F}'$}$$

$$\Box\Big(i \to \bigcirc(\neg i \,\mathcal{U}\, a)\Big) \wedge \Box\Big(i \to \bigcirc((\neg i \,\mathcal{U}\, b^1) \vee (\neg i \,\mathcal{U}\, b^2))\Big) \wedge \Box\Big(i \to \bigcirc((\neg i \,\mathcal{U}\, c) \vee (\neg i \,\mathcal{U}\, d))\Big) \tag{$\mathcal{A}$-After$\mathcal{F}'$}$$

The LTL encoding of the property

$$\mathcal{A} = \Big(\big(\{a, b^{[1,2]}\}, \wedge, \textit{Non-Shuffled}\big) < \big(\{c, d\}, \vee, \textit{Non-Shuffled}\big) \ll i \mid \nabla\Big),$$

when $\nabla = \textit{Non-Repeated}$ (resp. $\nabla = \textit{Repeated}$) is Conjunction 4.26 (resp. Conjunction 4.27). Figures 4.14(a) and 4.14(b) show monitors produced by SPOT for the respective conjunctions provided to the tool in SPOT syntax.

$$\mathcal{A}\text{-Exclusiveness} \wedge \mathcal{A}\text{-MaxOneNR} \wedge \mathcal{A}\text{-RangeNR} \wedge \mathcal{A}\text{-OrderNR} \wedge \mathcal{A}\text{-First}\mathcal{F}' \tag{4.26}$$

$$\mathcal{A}\text{-Exclusiveness} \wedge \mathcal{A}\text{-MaxOne} \wedge \mathcal{A}\text{-Range} \wedge \mathcal{A}\text{-Order} \wedge \mathcal{A}\text{-First}\mathcal{F}' \wedge \mathcal{A}\text{-After}\mathcal{F}' \tag{4.27}$$

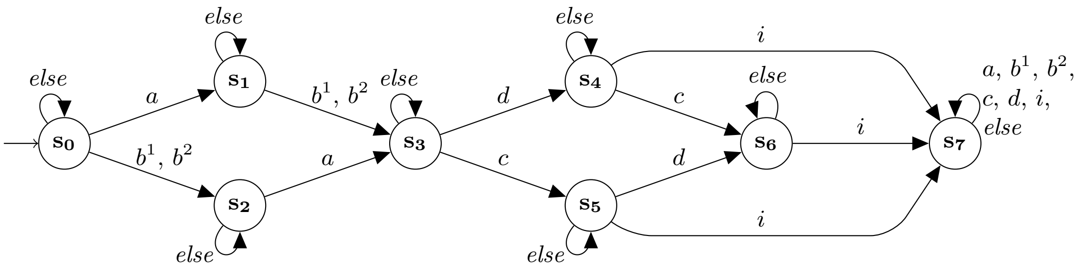
□

Testing with SPOT The correctness of the LTL encoding has been checked using the SPOT tool introduced in Section 4.5.3.1. For checking we selected different configurations of an antecedent requirement $\mathcal{A}$ and a timed implication constraint $\mathcal{T}$; they are presented in Tables 4.2 and 4.3 respectively. The forth (resp. third) column of Tables 4.2 (resp. Tables 4.3) provides references to the parts of this document where the interested reader can find LTL encodings of the respective loose-ordering properties. The fifth (resp. the forth) column defines the number of LTL and Boolean operators in the respective LTL formulas. The sixth (resp. the fifth) columns of the tables provide references to the figures illustrating corresponding SPOT monitors.

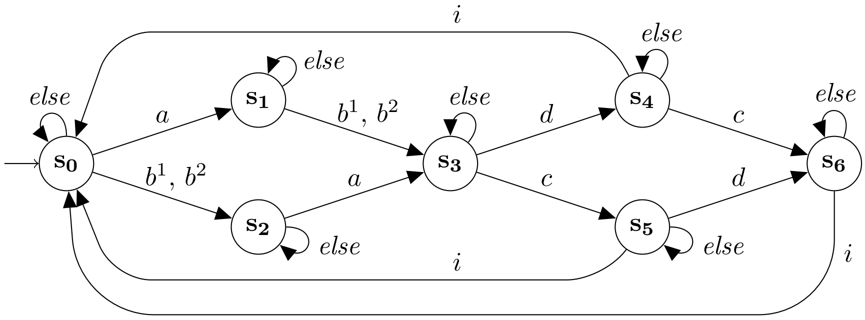
The testing consisted in (i) encoding the properties into LTL, (ii) providing SPOT with the obtained formulas, (iii) getting corresponding monitors, and (iv) ensuring that the state machines produced by the tool were correct with regard to the semantics of the encoded subset of the loose-ordering language.

4.5.3.3 The Number of Operators in LTL Encodings

The number of operators in the LTL encoding of $\mathcal{A} = \big(\mathcal{P} \ll i \mid \nabla\big)$ such that $\nabla = \textit{Repeated}$ (resp. $\nabla = \textit{Non-Repeated}$) is the sum of the LTL and Boolean operators appearing in each component of Conjunction 4.20 (resp. Conjunction 4.25). The number of operators in the LTL encoding of $\mathcal{T} = \big(\mathcal{P} \Longrightarrow \mathcal{Q} \mid t\big)$ is the sum of operators appearing in Conjunction 4.21. Thus, to compute the length of the encoding, we define the number of operators per each component of the conjunctions. We count only Boolean operators defined over LTL formulas.



(a) A non-repeated context ∇ = *Non-Repeated*



(b) A repeated context ∇ = *Repeated*

Figure 4.14 SPOT monitors for the LTL encoding of $\mathcal{A} = ((\{a, b^{[1,2]}\}, \wedge, \textit{Non-Shuffled}) < (\{c, d\}, \vee, \textit{Non-Shuffled}) \ll i \mid \nabla)$. *else* stands for any name which is not of $\hat{\alpha}(\mathcal{A}) = \{a, b^1, b^2, c, d, i\}$. Comma "," stands for disjunction. All names of $\hat{\alpha}(\mathcal{A})$ are mutually exclusive. Transitions which are not defined are forbidden.

Let $\mathcal{L} = \mathcal{F}_1 < \cdots < \mathcal{F}_\ell < \mathcal{F}'$. Let $\mathcal{F}_k = \big(\{\mathcal{R}^k_1, \ldots, \mathcal{R}^k_{m^k}\}, \sharp, \textit{Non-Shuffled}\big)$ for $k \in [1, \ell]$, and $\mathcal{R}^k_j = n^{[u^k_j, v^k_j]}$ for $j \in [1, m^k]$. Let $\mathcal{F}' = \big(\{\mathcal{R}'_1, \ldots, \mathcal{R}'_{m'}\}, \sharp, \textit{Non-Shuffled}\big)$, and $\mathcal{R}'_j = n^{[u'_j, v'_j]}$ for $j \in [1, m']$. Let $\hat{\alpha}(\mathcal{R})$ be a vocabulary of a range $\mathcal{R}$. Then the size of a vocabulary of $\mathcal{L}$ is defined by Formula 4.28.

$$|\hat{\alpha}(\mathcal{L})| = \sum_{k=1}^{\ell} \sum_{j=1}^{m^k} |\hat{\alpha}(\mathcal{R}^k_j)| + \sum_{j=1}^{m'} |\hat{\alpha}(\mathcal{R}'_j)| \tag{4.28}$$

The number of operators in the LTL encoding of the antecedent requirement $\mathcal{A}$ with a non-repeated context ∇ = *Non-Repeated* is defined by Sum 4.29. Here the summands are: ① (resp. ②) is the number of operators in the EXCLUSIVENESS (resp. RANGENR) after simplifications; ③ is the number of operators in the ORDERNR; ④ is the total number of operators in the components MAXONENR and FIRST$\mathcal{F}'$. We take into account that $\hat{\alpha}(\mathcal{F}') = \{i\}$, i.e., $\sum_{j=1}^{m'} |\hat{\alpha}(\mathcal{R}'_j)| = 1$.

$$\begin{aligned}
&① \sum_{k=1}^{\ell} \sum_{j=1}^{m^k} |\hat{\alpha}(\mathcal{R}^k_j)| \times \left[\sum_{k=1}^{\ell} \sum_{j=1}^{m^k} |\hat{\alpha}(\mathcal{R}^k_j)| + 1 \right] \\
&② \; + \; 4 \sum_{k=1}^{\ell} \sum_{j=1}^{m^k} \left[|\hat{\alpha}(\mathcal{R}^k_j)| \Big(|\hat{\alpha}(\mathcal{R}^k_j)| - 1 \Big) \right] \\
&③ \; + \; 4 \sum_{k=2}^{\ell} \left[\sum_{j=1}^{m^k} |\hat{\alpha}(\mathcal{R}^k_j)| \times \sum_{j=1}^{m^{k-1}} |\hat{\alpha}(\mathcal{R}^{k-1}_j)| \right] + \; ④ \; 8 \sum_{k=1}^{\ell} \sum_{j=1}^{m^k} |\hat{\alpha}(\mathcal{R}^k_j)|
\end{aligned} \tag{4.29}$$

No.	Configuration of $\mathcal{A}$	∇	LTL Encoding	|Operators|	SPOT Monitor
1.	$(a < b < c \ll i \mid \nabla)$	*Repeat.*	Appendix B.1.1	78	Figure B.1(b)
		Non-Rep.		62	Figure B.1(a)
2.	$\big((\{a,b,c,d\},\wedge) \ll i \mid \nabla\big)$	*Repeat.*	Appendix B.1.2	99	Figure B.4(b)
		Non-Rep.		75	Figure B.4(a)
3.	$\big((\{a,b^{[1,2]}\},\wedge) < (\{c,d\},\vee) \ll i \mid \nabla\big)$	*Repeat.*	Example 4.5.5	168	Figure 4.14(b)
		Non-Rep.		144	Figure 4.14(a)
4.	$\big((\{a,b\},\wedge) < (\{c,d^{[1,2]}\},\vee) \ll i \mid \nabla\big)$	*Repeat.*	Appendix B.1.3	168	Figure B.5(b)
		Non-Rep.		144	Figure B.5(a)
5.	$\big((\{a,b^{[1,2]}\},\vee) < (\{c,d\},\wedge) \ll i \mid \nabla\big)$	*Repeat.*	Appendix B.1.4	168	Figure B.7(b)
		Non-Rep.		144	Figure B.7(a)
6.	$\big((\{a,b\},\vee) < (\{c,d^{[1,2]}\},\wedge) \ll i \mid \nabla\big)$	*Repeat.*	Appendix B.1.5	168	Figure B.9(b)
		Non-Rep.		144	Figure B.9(a)
7.	$\big((\{a,b\},\wedge) < (\{c,d\},\vee) < (\{e,f\},\wedge) \ll i \mid \nabla\big)$	*Repeat.*	Appendix B.1.6	210	Figure B.11(b)
		Non-Rep.		177	Figure B.11(a)

Table 4.2 Configurations of $\mathcal{A} = (\mathcal{P} \ll i \mid \nabla)$ tested with SPOT tool. For all fragments $\sqcup\!\sqcup$ = *Non-Shuffled*.

No.	Configuration of $\mathcal{T}$	LTL Encoding	|Operators|	SPOT Monitor
1.	$\big(a \Longrightarrow (\{b,c\},\wedge) < (\{x,y\},\wedge) \mid t\big)$	Appendix B.2.1	169	Figure B.13
2.	$\big(a < (\{b,c^{[1,2]}\},\wedge) \Longrightarrow (\{x,y\},\wedge) \mid t\big)$	Appendix B.2.2	253	Figure B.15
3.	$\big(a \Longrightarrow (\{b,c\},\wedge) < (\{x,y^{[1,2]}\},\wedge) \mid t\big)$	Appendix B.2.3	271	Figure B.17
4.	$\big(a < (\{b,c\},\vee) \Longrightarrow (\{x,y\},\wedge) \mid t\big)$	Appendix B.2.4	163	Figure B.19
5.	$\big(a < (\{b,c^{[1,2]}\},\vee) \Longrightarrow (\{x,y\},\wedge) \mid t\big)$	Appendix B.2.5	247	Figure B.21
6.	$\big((\{a,b\},\wedge) < c \Longrightarrow (\{x,y^{[1,2]}\},\wedge) \mid t\big)$	Appendix B.2.6	271	Figure B.23
7.	$\big((\{a,b\},\vee) \Longrightarrow c < (\{x,y^{[1,2]}\},\wedge) \mid t\big)$	Appendix B.2.7	253	Figure B.25
8.	$\big((\{a,b\},\vee) < (\{c,d\},\wedge) \Longrightarrow (\{x,y^{[1,2]}\},\wedge) \mid t\big)$	Appendix B.2.8	378	Figure B.27

Table 4.3 Configurations of $\mathcal{T} = (\mathcal{P} \Longrightarrow \mathcal{Q} \mid t)$ tested with SPOT tool. For all fragments $\sqcup\!\sqcup$ = *Non-Shuffled*.

The number of operators in the LTL encoding of $\mathcal{A}$ with a repeated context ∇ = *Repeated* is Sum 4.30 where ① (resp. ②, ③) is the number of LTL operators in Exclusiveness (resp. Range, Order) component; ④ defines the total number of operators in the components MaxOne, First$\mathcal{F}'$ and After$\mathcal{F}'$.

$$
\begin{aligned}
&① \quad \sum_{k=1}^{\ell}\sum_{j=1}^{m^k} |\hat{\alpha}(\mathcal{R}_j^k)| \times \left[\sum_{k=1}^{\ell}\sum_{j=1}^{m^k} |\hat{\alpha}(\mathcal{R}_j^k)| + 1\right] \\
&② \quad + \; 5\sum_{k=1}^{\ell}\sum_{j=1}^{m^k} \left[|\hat{\alpha}(\mathcal{R}_j^k)|\Big(|\hat{\alpha}(\mathcal{R}_j^k)| - 1\Big)\right] \\
&③ \quad + \; 5\sum_{k=2}^{\ell} \left[\sum_{j=1}^{m^k} |\hat{\alpha}(\mathcal{R}_j^k)| \times \sum_{j=1}^{m^{k-1}} |\hat{\alpha}(\mathcal{R}_j^{k-1})|\right] \\
&④ \quad + \; 15\sum_{k=1}^{\ell}\sum_{j=1}^{m^k} |\hat{\alpha}(\mathcal{R}_j^k)|
\end{aligned}
\tag{4.30}
$$

The number of operators in the LTL encoding of $\mathcal{T} = \big(\mathcal{P} \Longrightarrow \mathcal{Q} \mid t\big)$ is defined by Sum 4.31. Here, ① defines the number of LTL operators in Exclusiveness component, ② and ③ are numbers of operators

in the components RANGE and RANGE$\mathcal{F}'$ respectively, ④ and ⑤ are respectively numbers of operators in the ORDER and the ORDER$\mathcal{F}'$, finally ⑥ is the total number of operators in the components MAXONE, FIRST$\mathcal{F}'$ and AFTER$\mathcal{F}'$.

$$
\begin{aligned}
① \quad & \left(\sum_{k=1}^{\ell}\sum_{j=1}^{m^k}|\hat{\alpha}(\mathcal{R}_j^k)| + \sum_{j=1}^{m'}|\hat{\alpha}(\mathcal{R}'_j)|\right) \times \left(\sum_{k=1}^{\ell}\sum_{j=1}^{m^k}|\hat{\alpha}(\mathcal{R}_j^k)| + \sum_{j=1}^{m'}|\hat{\alpha}(\mathcal{R}'_j)| - 1\right) \\
② \quad & + 5 \sum_{k=1}^{\ell}\sum_{j=1}^{m^k}\left[|\hat{\alpha}(\mathcal{R}_j^k)|\Big(|\hat{\alpha}(\mathcal{R}_j^k)| - 1\Big)\right] \times \sum_{j=1}^{m'}|\hat{\alpha}(\mathcal{R}'_j)| \\
③ \quad & + 5 \sum_{k=1}^{\ell}\sum_{j=1}^{m^k}|\hat{\alpha}(\mathcal{R}_j^k)| \times \sum_{j=1}^{m'}\left[|\hat{\alpha}(\mathcal{R}'_j)|\Big(|\hat{\alpha}(\mathcal{R}'_j)| - 1\Big)\right] \\
④ \quad & + 5 \sum_{k=2}^{\ell}\left[\sum_{j=1}^{m^k}|\hat{\alpha}(\mathcal{R}_j^k)| \times \sum_{i=1}^{m^{k-1}}|\hat{\alpha}(\mathcal{R}_i^{k-1})|\right] \times \sum_{j=1}^{m'}|\hat{\alpha}(\mathcal{R}'_j)| \\
⑤ \quad & + 5 \sum_{j=1}^{m'}|\hat{\alpha}(\mathcal{R}'_j)| \times \sum_{j=1}^{m^\ell}|\hat{\alpha}(\mathcal{R}_j^\ell)| \times \sum_{j=1}^{m^1}|\hat{\alpha}(\mathcal{R}_j^1)| \\
⑥ \quad & + 15 \sum_{j=1}^{m'}|\hat{\alpha}(\mathcal{R}'_j)| \times \sum_{k=1}^{\ell}\sum_{j=1}^{m^k}|\hat{\alpha}(\mathcal{R}_j^k)|
\end{aligned}
\tag{4.31}
$$

Formulas 4.29, 4.30 and 4.31 show that the length of the proposed LTL encoding for loose-ordering properties is *linear (quadratic) in the size of their vocabularies* (see the summand ① in the respective formulas). The formulas will be used in Chapter 7 to define the complexities of the PSL monitors [PF08; Fer11].

4.5.3.4 Remark: When Ranges are Shuffled

In Section 4.5.3.3 we provided the approximate length of the LTL encoding for a loose-ordering $\mathcal{L} = \mathcal{F}_1 < \cdots < \mathcal{F}_\ell < \mathcal{F}'$ under assumption that ranges of the fragments $\mathcal{F}_k$s and $\mathcal{F}'$ are not shuffled. To encode a fragment $\mathcal{F} = \big(\{\mathcal{R}_1, \ldots, \mathcal{R}_m\}, \sharp, \textit{Shuffled}\big)$ with shuffled ranges and the conjunctive semantics $\sharp = \wedge$ (resp. the disjunctive semantics $\sharp = \vee$), the idea would be to encode explicitly each sequence defined by $\mathcal{F}$ as a *nesting* of *next* "$\bigcirc$" operators.

EXAMPLE 4.5.6. – Consider a shuffled fragment $\big(\{n_1, n_2^2\}, \wedge, \textit{Shuffled}\big)$. It defines that exactly one occurrence of n_1 and exactly two occurrences of n_2 should occur; the order of occurrences is not fixed. To encode the fragment into LTL, one needs to encode explicitly all possible permutations of occurrences of n_1 and n_2:

$$\big(n_1 \wedge \bigcirc(n_2 \wedge \bigcirc n_2)\big) \vee \big(n_2 \wedge \bigcirc(n_1 \wedge \bigcirc n_2)\big) \vee \big(n_2 \wedge \bigcirc(n_2 \wedge \bigcirc n_1)\big)$$

□

Such encoding is as explosive as encoding into SERE (see Sec. 4.5.2). Moreover, as it was stated in Section 4.5.3.2, the semantics of the nested LTL formulas as well as the semantics of their conjunction is not clear, thus it is hard to get the encoding right.

Summary

In this chapter, we have defined *loose-ordering* specification primitives which allow definition of order and number non-determinism. Using loose-orderings we have defined the specification patterns: an *antecedent requirement* to define assumptions, and a *timed implication constraint* to specify guarantees of components. The properties of our case study serve examples for these new concepts. We have seen that although it is possible to encode loose-orderings into regular expressions and linear temporal logic, doing so may cause explosion either in the length of obtained formulas, or in the size of vocabularies. Therefore, using specification languages based on regular expressions and LTL (e.g., PSL) for expressing and checking loose-orderings may not be very efficient.

Chapter 5

Compositional Building of Recognizers

Contents

5.1 Introduction **89**
5.2 Recognition Principle **90**
5.3 Recognition Context **91**
5.4 Building Recognizers **92**
5.4.1 Elementary Recognizers of Ranges 93
5.4.2 Composite Recognizers 99
5.4.3 Recognizers of Loose-Ordering Properties 102
5.5 Validation **103**
5.5.1 Implementing Recognizers in LUSTRE 103
5.5.2 Obtaining Monitors from the LUSTRE Implementation 104

In this chapter, we define recognizers for loose-ordering properties. It is the first step towards efficient SystemC monitors. A recognizer is a machine which gets a sequences of names, one name per step, and delivers a Boolean output when any name violates a recognized property. We build recognizers in a compositional way: we propose primitive recognizers for ranges, the recognizers for fragments, loose-orderings, etc., are compositions of those primitive recognizers. The compositional structure facilitates the correctness checking of the obtained machines. We validate the recognizers and their compositions by an exhaustive testing of their encodings in LUSTRE.

5.1 Introduction

We want to check *at simulation* that the design satisfies loose-ordering properties defined in Chapter 4. To perform online checking, we define *recognizers* of the properties. Our recognizers are machines with *counters*, which (i) read sequences of inputs of I and/or outputs of O of a specified component, one name per step, (ii) (potentially) increase counters if names of interest occur, and (iii) deliver outputs when respective loose-ordering properties are *violated*. Properties are violated if any of the counters exceeds or falls behind its specified range. The names of $(I \cup O)$ which do not appear in the properties are ignored. Let $s = n^0 n^1 \ldots$ be a sequence of names of $(I \cup O)$. Online checking of a loose-ordering property consists in ensuring that the property holds on each prefix $s' = n^0 \ldots n^k$ of s (for any $k > 0$), i.e., the recognizer of the property never delivers error output. The first name n^k of s on which the recognizer complains corresponds to a step where violation of the property occurs.

Checking of properties relies on pattern matching of loose-orderings which can happen anywhere in a sequence $s = n^0 n^1 \ldots$. For instance, the antecedent requirement with non-repeated context $\mathcal{A} = (\mathcal{P} \ll i \mid \textit{Non-Repeated})$ holds, if the first occurrence of i is preceded by at least one occurrence of $\mathcal{P}$ (Fig. 5.1).

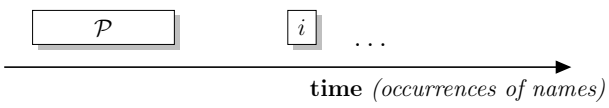


Figure 5.1 Example of a sequence satisfying $\mathcal{A} = (\mathcal{P} \ll i \mid \textit{Non-Repeated})$.

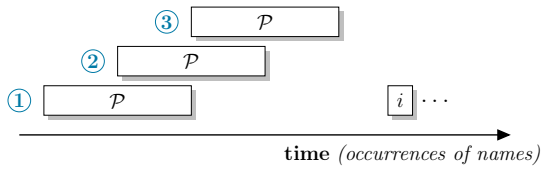


Figure 5.2 Overlapping of a loose-ordering $\mathcal{P}$: the first occurrence of $\mathcal{P}$ validates i.

To detect occurrences of a loose-ordering $\mathcal{P}$, one solution would be to start a new recognizer at each step as if $\mathcal{P}$ started there [Tom+09; GHR03]. Taking advantage of the restrictions of the loose-ordering language (see Table 4.1) we use only one *continuous* recognizer to detect occurrences of $\mathcal{P}$ (see Sec. 2.4).

A Continuous Recognizer of a Loose-Ordering We say that a loose-ordering $\mathcal{P}$ *holds* on a subsequence $s' = n^k \dots n^{k+m}$ of s, if (i) the projection of s' on names of $\mathcal{P}$ is a sequence defined by $\mathcal{P}$, and (ii) the first and the last letters of s' are names of $\mathcal{P}$ (resp. $\mathcal{Q}$). We call an *occurrence* of a loose-ordering $\mathcal{P}$ a subsequence on which $\mathcal{P}$ holds. If a subsequence $s' = n^k \dots n^{k+m}$ contains several overlapping occurrences of a loose-ordering (e.g., see 1, 2 and 3, in Fig. 5.2), the very first occurrence (1) is taken into account.

To detect occurrences of $\mathcal{P}$, we use only one *continuous* recognizer. The machine *re-initializes* when it fails to recognize a loose-ordering of a subsequence $s' = n^k \dots n^{k+m}$. Re-initialization means that the machine restarts by moving to its initial state and resets all its counters; when any name of $\mathcal{P}$ occurs, the re-initializing recognizer may start the recognition of $\mathcal{P}$ again. The definition of such a machine is possible due to the limitations of the loose-orderings language. We take advantage of the fact that *parts of $\mathcal{P}$ (fragments, ranges) do not share names*. It allows us to define (i) one counter per name of $\mathcal{P}$, and (ii) disjoint sets of counters for parts of $\mathcal{P}$. If the recognition of one part (a range of a fragment) fails, i.e., the respective counter (or counters) has a value which is not in a specified interval, the whole $\mathcal{P}$ should be restarted since there is no hope that $\mathcal{P}$ will ever hold on the current subsequence of names.

Limitations of Re-initializing Recognizers Our continuous recognizer of $\mathcal{P}$ works only if its complete re-initialization is possible, i.e., all counters can be reset. It means that the first fragment of $\mathcal{P}$ should have the *non-shuffled semantics*.

Compositional Building of Recognizers We built our recognizers in a compositional way. A composition is defined by the syntax tree of a loose-ordering property. The leaves of the tree correspond to elementary recognizers for ranges. The elementary recognizer has a *counter* and works in a *recognition context* defining which names should occur before or after respective ranges, etc. The recognition context is propagated downward in the syntax tree of the property. The elementary recognizers are either re-initializing, or they may report errors. In the former (resp. latter) case, occurrences of names of recognition context which violate a range make the recognizer to re-start (resp. to report an error).

A recognizer for any node which is not a leaf, i.e., a node representing a fragment, a loose-ordering, etc., is the composition of the recognizers for its children nodes. Our approach is in the spirit of [PF08; Bor+06], it facilitates the correctness checking of the composite recognizers [MAB06].

Organization of the Chapter In Section 5.2, we consider the recognition principle and give a brief overview of the compositional building of the recognizers for loose-orderings. In Section 5.3, we define a recognition context of the elementary recognizers of ranges. Section 5.4 presents a detailed definition of all types of the recognizers. In Section 5.5, we discuss the validation of the recognizers.

5.2 Recognition Principle

Recognizing a Range The recognizer of a range $\mathcal{R} = n^{[u,v]}$ works in a *recognition context*. The context defines which names should have occurred *before*, *immediately after*, *far after* the range, and names of *other ranges* of the same fragment. When $\mathcal{R}$ is being recognized: (i) names that should have happened before, lose their turn and should not occur again; (ii) names that go immediately after the range may stop the recognition of $\mathcal{R}$; (iii) names that are expected to appear far after $\mathcal{R}$ cannot stop recognition of $\mathcal{R}$ and they should not occur, (iv) names of other ranges of a parent fragment depending on the fragment's

semantics ⊔⊔ may appear either: (i) before and after a range (⊔⊔ = *Non-Shuffled*), or (ii) before, after and while the range (⊔⊔ = *Shuffled*). The recognizer has a *counter*; it increases when a name n of a range occurs. The names of the context should not occur when the recognizer is active and it has not recognized an occurrence of a range, i.e., the value of the counter is not in $[u, v]$. If the recognizer of a range is *continuous*, it re-starts when a violation happens; otherwise, the recognizer reports an error. When the elementary recognizer is stopped and it has detected an occurrence of a range, it delivers a Boolean output.

Recognizing a Fragment The recognizer of a fragment $\mathcal{F} = \big(\{\mathcal{R}_1, \ldots, \mathcal{R}_m\}, \sharp, ⊔⊔\big)$ is the synchronous parallel composition of the elementary recognizers of the ranges $\mathcal{R}_j$s. Each of those elementary recognizers has a context which is derived from $\mathcal{F}$ and other ranges. When the recognizer of $\mathcal{F}$ makes a step on an occurrence of a name appearing in $\mathcal{F}$, all elementary recognizers for ranges make a step, and one of them increases its counter. The *continuous recognizer* of $\mathcal{F}$ is made of the continuous recognizers of the $\mathcal{R}_j$s; such recognizer restarts if any of its ranges is violated. The *error reporting recognizer* of $\mathcal{F}$ is made of the elementary recognizers of the $\mathcal{R}_j$s reporting errors; such recognizer signals an error if any of the $\mathcal{R}_j$s is violated (i.e., the corresponding elementary recognizer delivers an error output). A fragment $\mathcal{F}$ is recognized (the range delivers a Boolean output), if all ($\sharp = \wedge$) or at least one ($\sharp = \vee$) of its ranges is recognized (the elementary recognizers deliver respective Boolean outputs).

Recognizing a Loose-Ordering The recognizer of a loose-ordering $\mathcal{L} = \mathcal{F}_1 < \cdots < \mathcal{F}_\ell$ produces an output when it detects an occurrence of $\mathcal{L}$. It ignores (loops) names which are not of $\mathcal{L}$. The recognizer is made by composing sequentially recognizers of the fragments $\mathcal{F}_k$s. When $\mathcal{L}$ is being recognized, only one recognizer of the $\mathcal{F}_k$s is active at a time; an occurrence of a name can make the recognizer of $\mathcal{F}_{k-1}$ ($k > 1$) to signal its termination, and the recognizer of $\mathcal{F}_k$ to start. An occurrence of $\mathcal{L}$ is successfully recognized, if the last fragment of $\mathcal{L}$ signals its termination.

The recognition of $\mathcal{L}$ is restarted, if (i) a loose-ordering $\mathcal{L}$ appears in $\mathcal{A} = \big(\mathcal{P} \ll i \mid \nabla\big)$ or it is the left loose-ordering of $\mathcal{T} = \big(\mathcal{P} \Longrightarrow \mathcal{Q} \mid t\big)$, i.e., $\mathcal{L} = \mathcal{P}$, (ii) any of the recognizers of its fragments $\mathcal{F}_k$s detects a violation. The continuous recognizer of $\mathcal{L}$ is made of the continuous recognizers of the $\mathcal{F}_k$s.

The recognition of $\mathcal{L}$ reports an error, if (i) a loose-ordering $\mathcal{L}$ is the right loose-ordering of $\mathcal{T} = \big(\mathcal{P} \Longrightarrow \mathcal{Q} \mid t\big)$, i.e., $\mathcal{L} = \mathcal{Q}$, (ii) the recognizer of $\mathcal{L}$ detects a violation of $\mathcal{L}$. The recognizer of $\mathcal{L}$ in this case is made of the error reporting recognizers of the $\mathcal{F}_k$s.

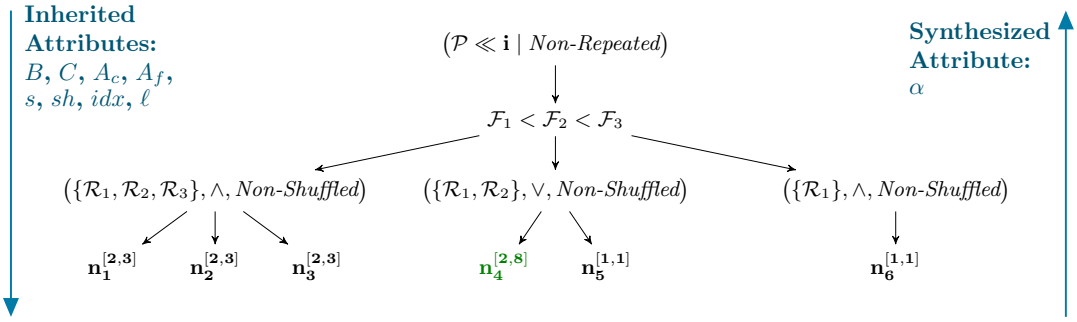
Recognizing Loose-Ordering Properties The recognizer of an antecedent requirement $\mathcal{A} = \big(\mathcal{P} \ll i \mid \nabla\big)$ is made of the continuous recognizer of $\mathcal{P}$. On each occurrence of a name of $\mathcal{A}$ the recognizer updates its state. If $\mathcal{A}$ has a non-repeated context (∇ = *Non-Repeated*), when the first i occurs the recognizer of $\mathcal{A}$ checks if an occurrence of $\mathcal{P}$ has been detected at least once before. If $\mathcal{A}$ has a repeated context (∇ = *Repeated*), when i occurs the recognizer checks if an occurrence of $\mathcal{P}$ has been detected since the previous occurrence of i.

The recognizer of a timed implication constraint $\mathcal{T} = \big(\mathcal{P} \Longrightarrow \mathcal{Q} \mid t\big)$ is sequentially composed of the continuous recognizer of $\mathcal{P}$ and the error reporting recognizer of $\mathcal{Q}$. When $\mathcal{P}$ is detected, recognition of $\mathcal{Q}$ is started. If the recognizer of $\mathcal{Q}$ is active, all attempts to re-start the recognizer are ignored.

5.3 Recognition Context

A recognizer of a range works in a *recognition context*, depending on where the range appears in the syntax tree of an antecedent requirement or a timed implication constraint. Consider for example the property of Figure 5.3. While recognizing $n_4^{[2,8]}$: (i) n_1, n_2 and n_3 should not occur, since they are supposed to have happened before. (ii) n_5 should not occur unless n_4 has occurred at least twice, this is because the parent fragment $\mathcal{F}_2$ disables shuffling. Notice that n_5 can occur both before and after the range $n_4^{[2,8]}$ since it belongs to $\mathcal{F}_2$. (iii) n_6 should not occur until n_4 has been observed at least twice, in which case it stops the recognition of the range $n_4^{[2,8]}$ (and starts the recognition of the appropriate range). (iii) i should not occur since it must take place after n_4 and it may not act as a stopping condition. Moreover, the recognizer for a range depends on whether its parent fragment has the conjunctive ($\wedge$) or disjunctive ($\vee$), shuffled (*Shuffled*) or non-shuffled (*Non-Shuffled*) semantics, and the position of the fragment in the parent loose-ordering (the fragment is the first, the second, etc).

The **context** for a range recognizer is therefore made of: (i) the semantics s in $\{\wedge, \vee\}$ and sh in $\{$*Non-Shuffled*, *Shuffled*$\}$ of the parent fragment; (ii) the index of the parent fragment $idx \in \mathbb{N}$ and the



Example for $\mathbf{n}_4^{[2,8]}$: $B = \{\mathbf{n_1}, \mathbf{n_2}, \mathbf{n_3}\}$, $C = \{\mathbf{n_5}\}$, $A_c = \{\mathbf{n_6}\}$, $A_f = \{\mathbf{i}\}$, $s = \vee$, $sh = \textit{Non-Shuffled}$, $idx = 2$, $\ell = 3$, $\alpha = \{\mathbf{n_4}\}$

Figure 5.3 A syntax tree of the loose-ordering property $\big((\{n_1^{[2,3]}, n_2^{[2,3]}, n_3^{[2,3]}\}, \wedge, \textit{Non-Shuffled}) < (\{n_4^{[2,8]}, n_5\}, \vee, \textit{Non-Shuffled}) < n_6 \ll i \mid \textit{Non-Repeated}\big)$.

number of fragments in a parent loose-ordering $\ell \in \mathbb{N}$ (idx and ℓ define the position of the fragment); (ii) the set B of names that should have occurred before (like n_1 in the above example); (iii) the set C of names of the other ranges of the parent fragment (like n_5 for $n_4^{[2,8]}$) (iv) the set A_c of names that will be required to happen just after the range, and may therefore act as delimiters for the end of the range (like n_6); (v) the set A_f of names that will be required to happen after, but cannot serve as delimiters (like i). We denote the context as a tuple $(B, C, A_c, A_f, s, sh, idx, \ell)$.

Different names of the recognition context of a range are treated differently depending on the place of the range in the syntax tree of a parent property. Thus, if the parent fragment of the range is a fragment of the left loose-ordering $\mathcal{P}$ of $\mathcal{A}$ (resp. $\mathcal{T}$) (like $(\{n_4^{[2,8]}, n_5\}, \wedge)$ in Fig. 5.3), all names which should not occur when the range is being recognized (like n_1) re-start the recognizer. If the parent fragment of the range is a fragment of the right loose-ordering $\mathcal{Q}$ of $\mathcal{T}$, all names which take place not in an appropriate time (e.g., names of B) make the recognizer to move to the error state, i.e., they are forbidden. The parameters of the recognition context sh, idx and ℓ define the type of an elementary recognizer that should be used for the recognition of a range (see Sec. 5.4.1).

The derivation of the contexts for the leaves (ranges) of the syntax tree of an antecedent requirement $\mathcal{A}$ or a timed implication $\mathcal{T}$ can be defined by means of an attribute grammar (see formal definition in Chapter 6). The list of its main attributes is shown in Figure 5.3. The derivation principle is the following: Starting from an antecedent requirement $\mathcal{A}$ or a timed implication $\mathcal{T}$, the contexts are propagated downwards in the syntax tree to the leaves (ranges). This can be expressed easily with one synthesized and eight inherited attributes shown in Figure 5.3. The synthesized attribute is α (denotes a set of names appearing in the formula of each node) since evaluation of the context for a particular node depends on names of other parts of the formula.

5.4 Building Recognizers

The recognizers of loose-ordering properties of either the "antecedent requirement" or "timed implication constraint" type are built of the *continuous* and *error reporting* recognizers of loose-orderings. The recognizers of loose-orderings are built from the elementary recognizers with counters of ranges of appropriate type. We distinguish *five* types of the recognizers of ranges: *error-reporting*, *re-initializing*, *resetting*, *stopping* and *mixed* resetting-stopping. The elementary recognizer of the error-reporting type has an error state where it moves if a range is violated. The composition of such elementary recognizers defines the error-reporting recognizers of fragments which are used to build the error-reporting recognizers of loose-orderings (Fig. 5.4). If any of those recognizers reports an error, the recognizer of a loose-ordering signals an error.

Other types of elementary recognizers of ranges are used to build the continuous recognizer of a loose-ordering $\mathcal{L} = \mathcal{F}_1 < \cdots < \mathcal{F}_\ell$ (Fig. 5.5).

1. The elementary recognizer of the *re-initializing* type resets its counter and moves to the idle state when names of the recognition context take place *before* a range is detected. The composition of the

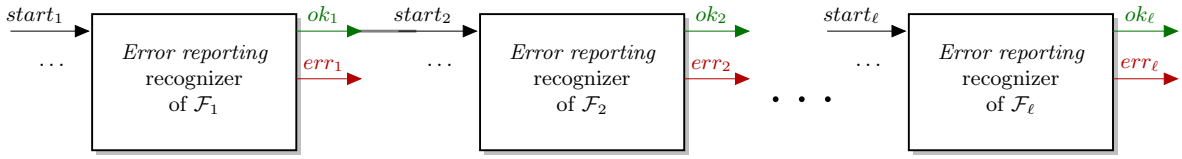


Figure 5.4 Building the error reporting recognizer of $\mathcal{L} = \mathcal{F}_1 < \cdots < \mathcal{F}_\ell$.

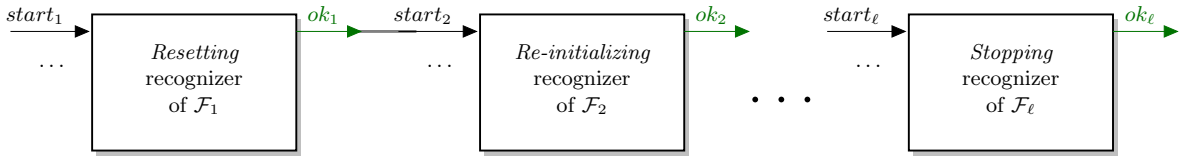


Figure 5.5 Building the continuous recognizer of $\mathcal{L} = \mathcal{F}_1 < \cdots < \mathcal{F}_\ell$.

elementary recognizers of this type is the continuous recognizer of a fragment of $\mathcal{L}$ which is neither the first, nor the last.

2. The *resetting* recognizers of ranges on occurrences of certain names of the context reset their counters but stay active waiting for their names to come. They are used to build the recognizer of the first fragment $\mathcal{F}_1$ of $\mathcal{L}$. Resetting occurs when, for instance, one of the elementary recognizers detects too many occurrences of names of the corresponding range of $\mathcal{F}_1$. In this case, the recognizers of other ranges of $\mathcal{F}_1$ should reset their counters, because the recognition of their ranges should be repeated.

3. The *stopping* recognizer of a range re-initializes if names of the recognition context occur before a range is recognized. The recognizer can be stopped by *any name of the context* (e.g., name of B, A_f, etc.) if that name occurs after the range is detected. The elementary recognizers of the stopping type are used to build the continuous recognizer of the last fragment of $\mathcal{L}$.

4. The *mixed resetting-stopping* recognizers of ranges can both reset and stop when some names of the recognition context occur. They are used if $\mathcal{L}$ has only one fragment.

The rest of the section is organized as follows. In Section 5.4.1, we define different types of the elementary recognizers of ranges in details. In Section 5.4.2, we build the composite recognizers of fragments and loose-orderings. In Section 5.4.3, we discuss definitions of the recognizers of loose-ordering properties.

5.4.1 Elementary Recognizers of Ranges

An elementary recognizer of a range $\mathcal{R} = n^{[u,v]}$ is an extended Mealy machine with a *counter* (see definition in Sec. 2.2.1.3). The counter is increasing when the name n of the range occurs. Inputs are names of the parent loose-ordering property of $\mathcal{R}$; outputs are emitted either when an occurrence of $\mathcal{R}$ is detected (i.e., $\mathcal{R}$ holds), or when the range is violated. The elementary recognizer loops when names which do not appear in the parent property occur. The behavior of our elementary recognizers depends on the recognition context of ranges, specifically on the semantics of a parent fragment of a range (shuffled $\sqcup\!\sqcup$ = *Shuffled* or non-shuffled $\sqcup\!\sqcup$ = *Non-Shuffled*) defined by the parameter s of the context, and the position of the parent fragment in the parent loose-ordering defined by the parameters idx and ℓ. We propose the following set of elementary recognizers:

1. $\mathbb{R}^{error}_{non\text{-}shuffled}$ (resp. $\mathbb{R}^{error}_{shuffled}$) is the elementary recognizer with the *error state* of ranges appearing in a fragment with the non-shuffled (resp. shuffled) semantics.
2. $\mathbb{R}^{re\text{-}init}_{non\text{-}shuffled}$ (resp. $\mathbb{R}^{re\text{-}init}_{shuffled}$) is the *re-initializing recognizer* of ranges appearing in a fragment with the non-shuffled (resp. shuffled) semantics, such that the recognizer can only re-initialize, it *does not reset* and *can be stopped only by names of* A_c appearing immediately after the range.
3. $\mathbb{R}^{stop}_{non\text{-}shuffled}$ (resp. $\mathbb{R}^{stop}_{shuffled}$) is the elementary recognizer of ranges appearing in a fragment with the non-shuffled (resp. shuffled) semantics, which can be *stopped by any name of the context*.
4. $\mathbb{R}^{reset}_{non\text{-}shuffled}$ is the elementary recognizer of ranges appearing in a fragment with the non-shuffled semantics, which can *reset*.
5. $\mathbb{R}^{reset,stop}_{non\text{-}shuffled}$ is the elementary recognizer of ranges appearing in a fragment with the non-shuffled semantics, which can *reset* and *can be stopped by any name of the context*.

We do not provide the continuous recognizer with a resetting ability of ranges appearing in a fragment with shuffled semantics. In the following sections we give detailed definition of the listed elementary recognizers.

5.4.1.1 Recognizers Reporting Errors

Figure 5.6 shows the elementary recognizer $\mathbb{R}^{error}_{non\text{-}shuffled}$ of a range $\mathcal{R} = n^{[u,v]}$ with the error state and context. $\mathbb{R}^{error}_{non\text{-}shuffled}$ recognizes a range which belongs to a fragment with the non-shuffled semantics. The recognizer is started with the input *start*, termination is signaled by the outputs *ok* or *nok*. In s_0, it is idle and waits to be started; s_5 is the error state; in s_1, it is started and waits for the first occurrence of its name n; in s_3, it is counting the occurrences with a counter `cpt`; in s_2, it is started and waits for the first occurrence of its name n, another range of the same fragment has started; in s_4, the minimum number of occurrences of n have been recognized, and another range of the same fragment has started. When the recognizer is in s_4 it means that the range has occurred, thus, any occurrence of the name n causes an error.

When the recognizer $\mathbb{R}^{error}_{non\text{-}shuffled}$ is started, occurrences of names that should have occurred before the range (names of B of the context) and occurrences of names that should happen after the range and do not serve delimiters (names of A_f) *always cause errors*. If names of other ranges of the parent fragment of $\mathcal{R}^{[u,v]}$ (names of C) or names which go immediately after the range (names of A_c) occur when (i) the recognizer is counting the occurrences of n in s_3 and (ii) the minimum number of occurrences of n was not detected, those names cause an error. If the range was detected (i.e., n occurred at least u times), occurrences of names of A_c stop the recognizer $\mathbb{R}^{error}_{non\text{-}shuffled}$.

Termination of $\mathbb{R}^{error}_{non\text{-}shuffled}$ is defined by the semantics `s` of the parent fragment of $\mathcal{R} = n^{[u,v]}$. If the semantics is disjunctive (`s`$= \vee$), the recognizer can be stopped before it detects any occurrence of n, otherwise the recognizer reports an error.

Figure 5.7 shows the elementary recognizer $\mathbb{R}^{error}_{shuffled}$ of ranges appearing in a fragment with the shuffled semantics; the recognizer has an error state. Unlike its counterpart in Figure 5.6, the recognizer while counting occurrence of its name n in s_3, accepts occurrences of names of other ranges of the same fragment (names of C).

5.4.1.2 Continuous Recognizers

Re-initializing Recognizers Figure 5.8 (resp. Figure 5.9) shows the continuous recognizer $\mathbb{R}^{re\text{-}init}_{non\text{-}shuffled}$ (resp. $\mathbb{R}^{re\text{-}init}_{shuffled}$) of ranges appearing in a fragment with the non-shuffled (resp. shuffled) semantics. It is symmetric to the elementary recognizer $\mathbb{R}^{error}_{non\text{-}shuffled}$ (resp. $\mathbb{R}^{error}_{shuffled}$) with an error state. The elementary recognizer reporting errors and the re-initializing continuous recognizer implement the same termination principle (see above). Unlike $\mathbb{R}^{error}_{non\text{-}shuffled}$ (resp. $\mathbb{R}^{error}_{shuffled}$), the continuous recognizer does not report an error when unexpected names of the recognition context occur (e.g., names of $\mathcal{A}_f$). Instead the machine *re-initializes* moving to the idle state s_0.

Stopping Recognizers Figure 5.10 (resp. Figure 5.11) shows the continuous recognizer $\mathbb{R}^{stop}_{non\text{-}shuffled}$ (resp. $\mathbb{R}^{stop}_{shuffled}$) of a range $\mathcal{R} = n^{[u,v]}$ belonging to a fragment with the non-shuffled (resp. shuffled) semantics. The recognizer is *re-initializing*, as its counterpart $\mathbb{R}^{re\text{-}init}_{non\text{-}shuffled}$ (resp. $\mathbb{R}^{re\text{-}init}_{shuffled}$) the machine moves to the idle state s_0 if names interrupting an occurrence of the range occur (names of B, A_f, A_c). If $\mathbb{R}^{stop}_{non\text{-}shuffled}$ detects a range $n^{[u,v]}$, it can be *stopped* by any name of a parent property. When it is stopped the machine produces a Boolean output. The elementary recognizer stops if it detects the maximum number of occurrences of n, and n occurs again. In this case the recognizer produces the output *stop*. Through input *stopC* the recognizer is notified if any of the recognizers of companion ranges has stopped due to the number of occurrence of names exceeding the defined ranges.

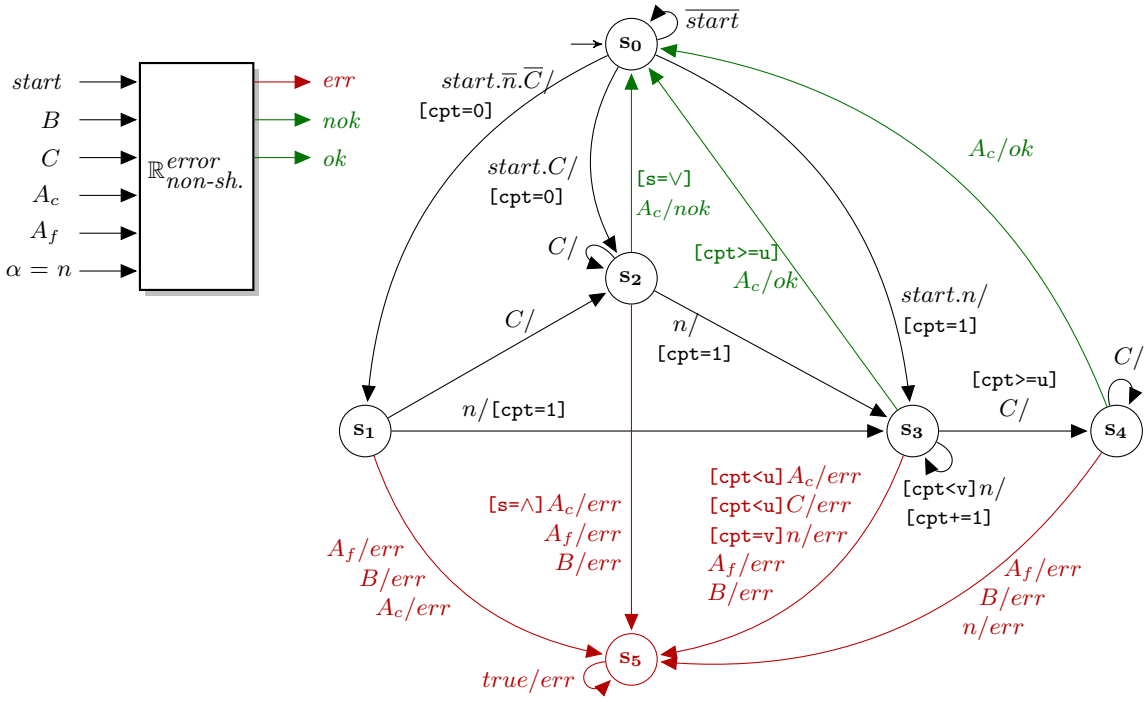


Figure 5.6 The error-reporting elementary recognizer $\mathbb{R}^{error}_{non\text{-}shuffled}$ of a range $\mathcal{R} = n^{[u,v]}$ appearing a fragment with the non-shuffled semantics ($\sqcup\!\sqcup$ = *Non-Shuffled*). `s`, B, A_c, A_f, C are the context. `cpt` is a counter; $\{start, n, B, A_c, A_f, C\}$ are inputs; $\{err, ok, nok\}$ are outputs. Each transition is of the form `[condition]`*input*/*output*`[action]` where *input* is a Boolean formula and *output* is a set.

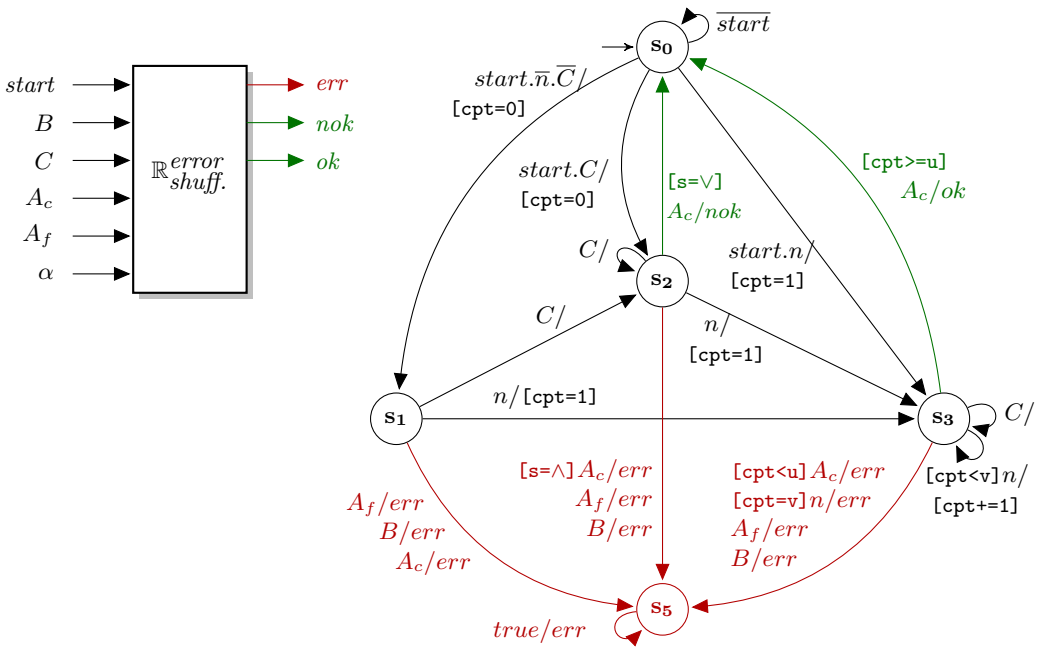


Figure 5.7 The error-reporting elementary recognizer $\mathbb{R}^{error}_{shuffled}$ of a range appearing in a fragment with the shuffled semantics ($\sqcup\!\sqcup$ = *Shuffled*).

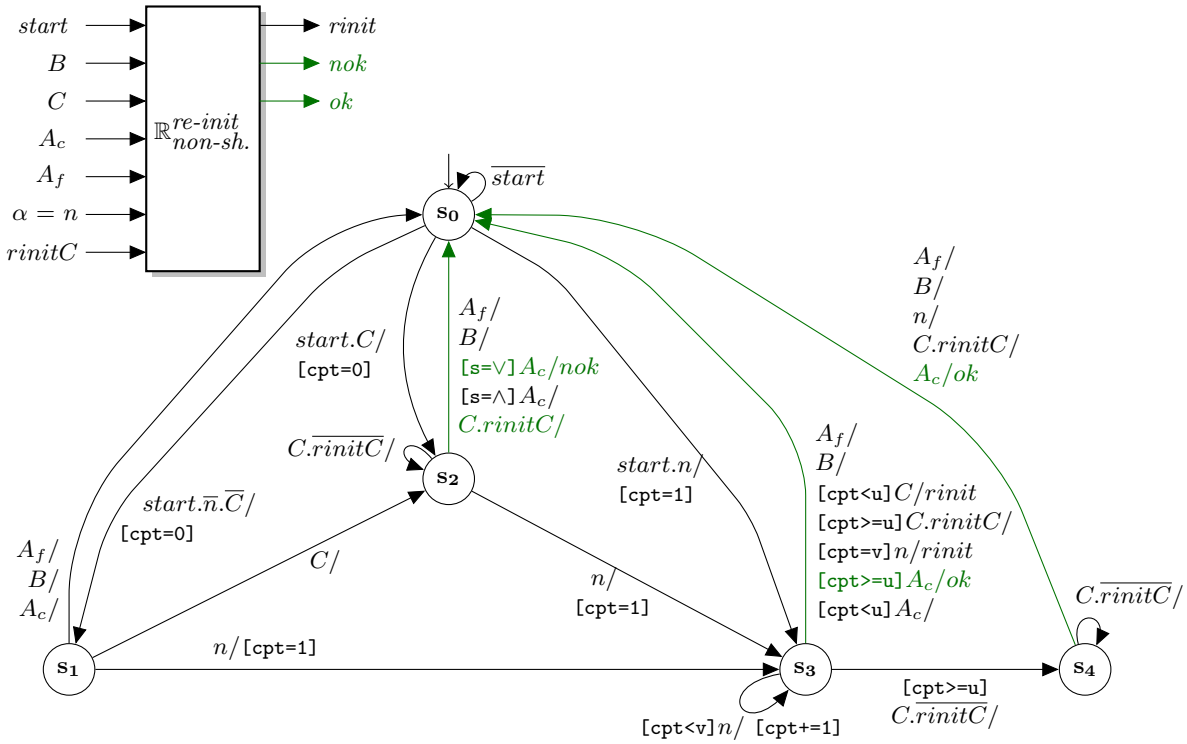


Figure 5.8 The re-initializing elementary recognizer $\mathbb{R}^{re\text{-}init}_{non\text{-}shuffled}$ of a range appearing in a fragment with the non-shuffled semantics (⧢ = *Non-Shuffled*).

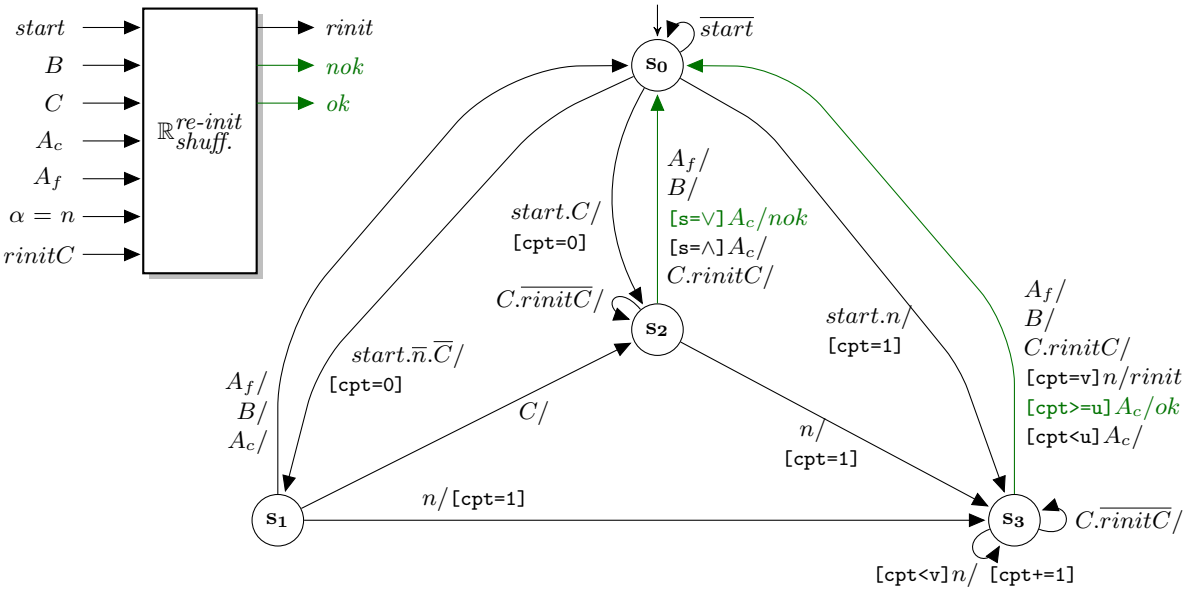


Figure 5.9 The re-initializing elementary recognizer $\mathbb{R}^{re\text{-}init}_{shuffled}$ of a range appearing in a fragment with the shuffled semantics (⧢ = *Shuffled*).

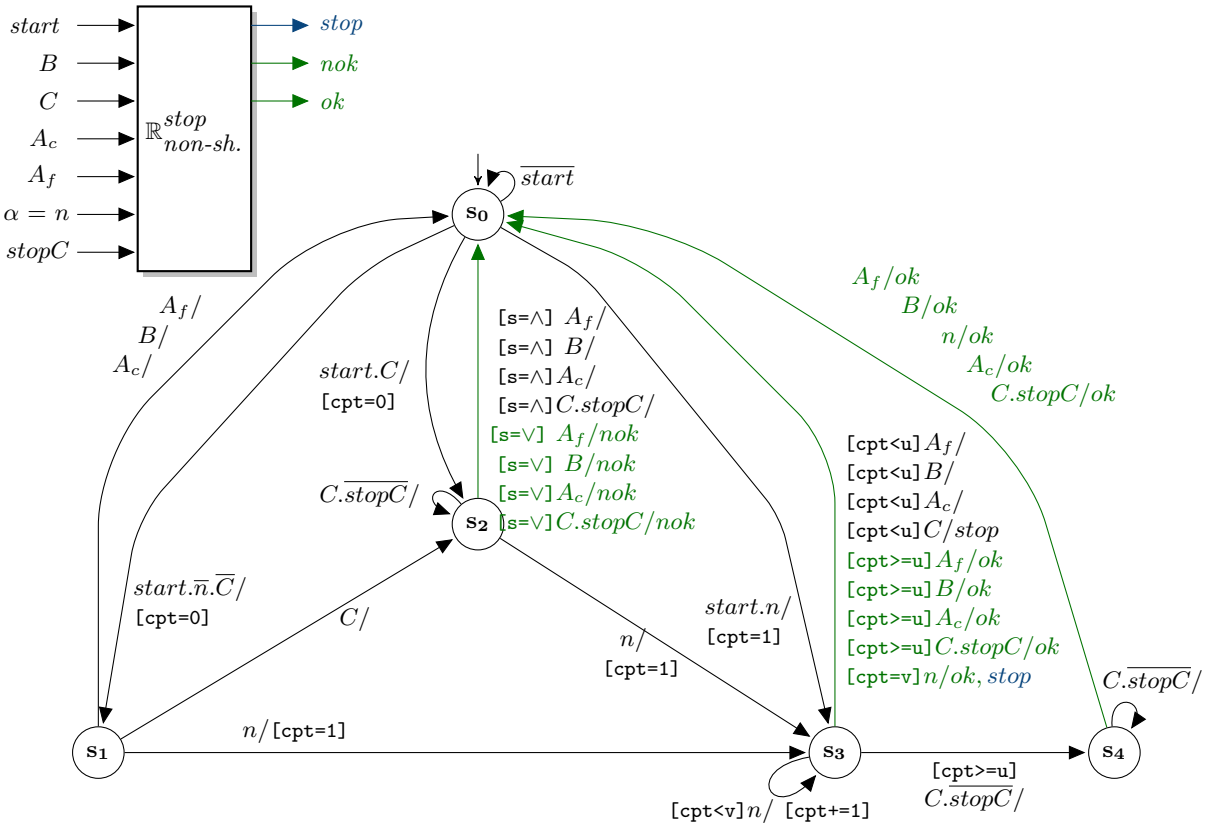


Figure 5.10 The stopping elementary recognizer $\mathbb{R}^{stop}_{non\text{-}shuffled}$ of a range appearing in a fragment with the non-shuffled semantics (⊔ = *Non-Shuffled*).

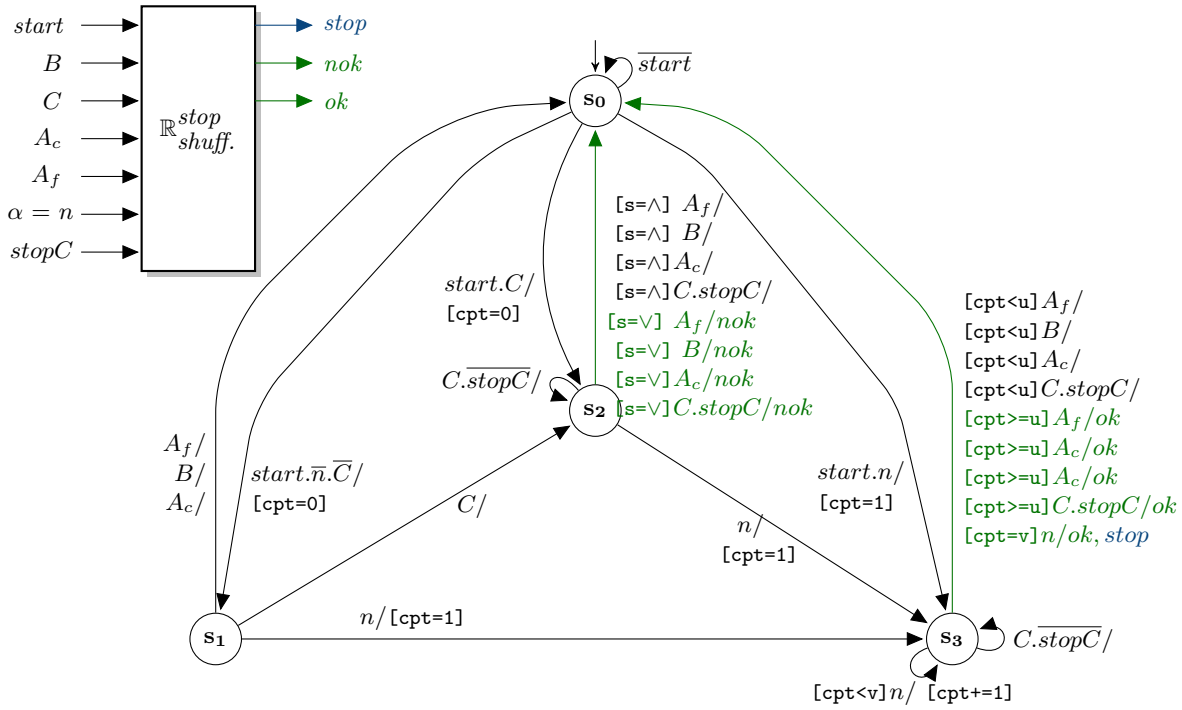


Figure 5.11 The stopping elementary recognizer $\mathbb{R}^{stop}_{shuffled}$ of a range appearing in a fragment with the shuffled semantics (⊔ = *Shuffled*).

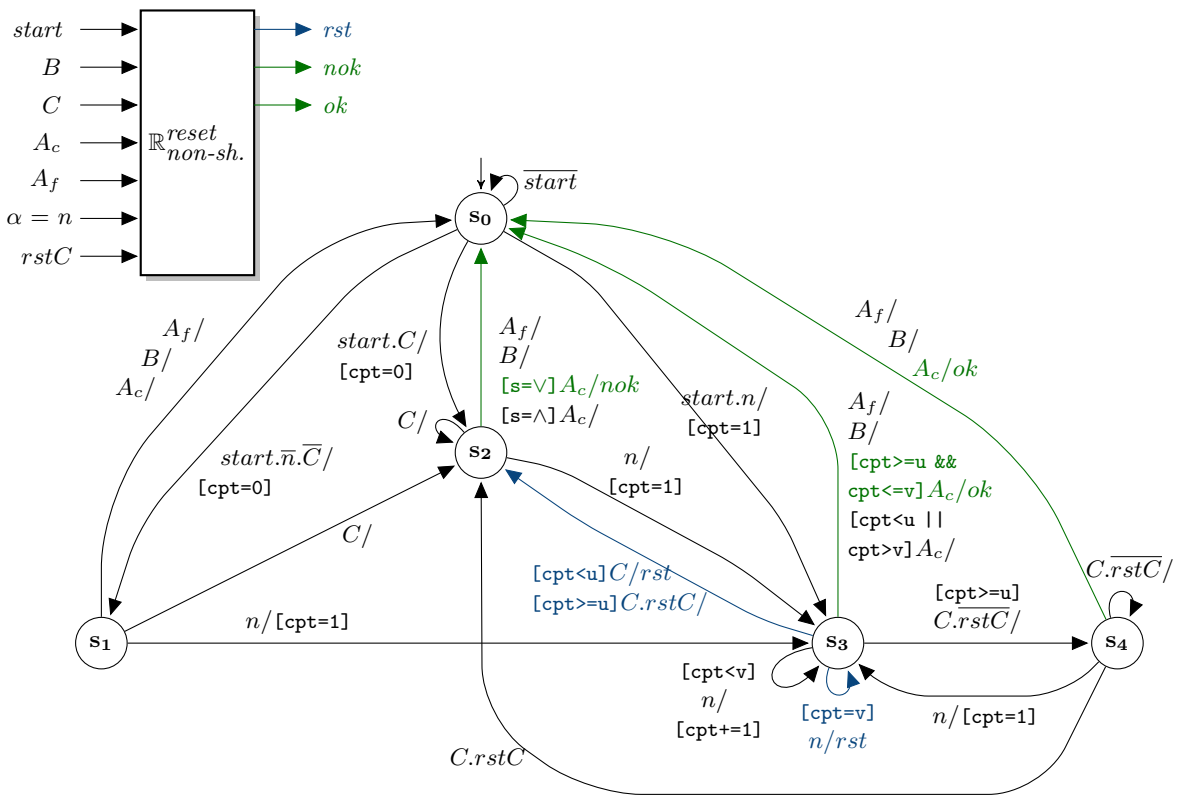


Figure 5.12 The resetting elementary recognizer $\mathbb{R}^{reset}_{non\text{-}shuffled}$ of a range appearing in a fragment with non-shuffled semantics (⊔ = *Non-Shuffled*).

Resetting Recognizers Figure 5.12 shows the continuous recognizer $\mathbb{R}^{reset}_{non\text{-}shuffled}$ of a range $\mathcal{R} = n^{[u,v]}$ belonging to a fragment with the non-shuffled semantics. The recognizer is *re-initializing*, as its counterpart $\mathbb{R}^{re\text{-}init}_{non\text{-}shuffled}$, the machine moves to the idle state s_0 if names interrupting an occurrence of the range occur (names of B, A_f, A_c). $\mathbb{R}^{reset}_{non\text{-}shuffled}$ can *reset*. Resetting means that the recognizer (potentially) discards already detected occurrences of n; the reset is activated with the input *rstC*. The reset may happen if:

(i) names of companion ranges (names of C) occur before the minimum number of n was detected,
(ii) the maximum number of occurrences of n was exceeded,
(iii) the maximum number of occurrences of names of companion ranges (names of C) was exceeded.

The recognizer $\mathbb{R}^{reset}_{non\text{-}shuffled}$ signals the reset with the output *rst*.

Mixed Resetting-Stopping Recognizers The recognizer $\mathbb{R}^{reset,stop}_{non\text{-}shuffled}$ combines stopping and resetting features of the continuous recognizers $\mathbb{R}^{stop}_{non\text{-}shuffled}$ (Fig. 5.10) and $\mathbb{R}^{reset}_{non\text{-}shuffled}$ (Fig. 5.12). We do not provide the definition of the machine here.

5.4.2 Composite Recognizers

The elementary recognizers of ranges are used to construct recognizers of fragments and loose-orderings. The continuous recognizer of a fragment (resp. a loose-ordering) is the composition of the continuous recognizers of their ranges. The recognizer of a fragment (resp. a loose-ordering) with the error state is the composition of the recognizers with error states of the ranges appearing in that fragment (resp. loose-ordering).

5.4.2.1 Recognizers of Fragments

Consider a fragment $\mathcal{F} = \big(\{\mathcal{R}_1, \ldots, \mathcal{R}_m\}, \sharp, \sqcup\!\sqcup\big)$, where $\sharp \in \{\wedge, \vee\}$ and $\sqcup\!\sqcup \in \{\textit{Non-Shuffled}, \textit{Shuffled}\}$. The recognizer of $\mathcal{F}$ is the synchronous parallel composition of the recognizers of the $\mathcal{R}_j$s. The type of elementary recognizers being composed (shuffled or non-shuffled) is defined by the semantics $\sqcup\!\sqcup$ of $\mathcal{F}$ and the position of $\mathcal{F}$ in a parent loose-ordering. Each of the elementary recognizers has a context depending on the other ranges of $\mathcal{F}$ (the set of names C) and the context inherited from $\mathcal{F}$ (the sets of names B, A_c, A_f and the semantics s). The recognizer of $\mathcal{F}$ signals termination with the output *ok* if all the recognizers for the $\mathcal{R}_i$s have signaled termination, i.e., produced *ok* or *nok*.

Recognizers Reporting Errors The recognizer of a fragment $\mathcal{F} = \big(\{\mathcal{R}_1, \ldots, \mathcal{R}_m\}, \sharp, \sqcup\!\sqcup\big)$, $\sharp \in \{\wedge, \vee\}$, $\sqcup\!\sqcup \in \{\textit{Non-Shuffled}, \textit{Shuffled}\}$ with the error state is the synchronous parallel composition of the elementary recognizers of the $\mathcal{R}_j$s reporting errors. Equation 5.1 (resp. Equation 5.2) defines the recognizer of $\mathcal{F}$ with the non-shuffled (resp. shuffled) semantics.

$$\mathbb{F}^{\textit{error}}_{\textit{non-shuffled}} = \mathbb{R}^{\textit{error}}_{\textit{non-shuffled},1} \times \cdots \times \mathbb{R}^{\textit{error}}_{\textit{non-shuffled},m} \tag{5.1}$$

$$\mathbb{F}^{\textit{error}}_{\textit{shuffled}} = \mathbb{R}^{\textit{error}}_{\textit{shuffled},1} \times \cdots \times \mathbb{R}^{\textit{error}}_{\textit{shuffled},m} \tag{5.2}$$

Here, "$\times$" is an operator of the synchronous product of Mealy machines (see Definition 4 in Sec. 2.2.1). For $j \in [1, m]$, $\mathbb{R}^{\textit{error}}_{\textit{non-shuffled},j}$ (resp. $\mathbb{R}^{\textit{error}}_{\textit{shuffled},j}$) is the elementary recognizer of ranges reporting errors (see Sec. 5.4.1.1). For instance, consider a fragment $\mathcal{F} = \big(\{n_4^{[2,8]}, n_5\}, \vee, \textit{Non-Shuffled}\big)$. Figure 5.13 shows the synchronous product of the elementary recognizers of the ranges $n_4^{[2,8]}$ and n_5. The recognizer of $\mathcal{F}$ signals an error with the output *err* when any of the recognizers of the $\mathcal{R}_j$s detects an error.

If a fragment $\mathcal{F}$ has the non-shuffled semantics ($\sqcup\!\sqcup = \textit{Non-Shuffled}$), at any time the composed elementary recognizers of ranges can be in one of the following *global states*: (i) all are idle (state s_0); (ii) all are waiting (in state s_1); (iii) exactly one recognizer is counting (state s_3) and all others are either still waiting for their names to come (state s_2) or have already recognized their ranges (state s_4). If the semantics $\mathtt{s}$ inherited from the parent fragment $\mathcal{F}$ is disjunctive (i.e. $\mathtt{s}{=}\vee$), each recognizers can be stopped before it has even started counting, provided that at least one of the other ranges of $\mathcal{F}$ has started recognizing a sequence of its name (e.g., see Fig. 5.13).

If a fragment $\mathcal{F}$ has the shuffled semantics ($\sqcup\!\sqcup = \textit{Shuffled}$) (e.g., $\mathcal{F} = \big(\{n_4^{[2,8]}, n_5\}, \vee, \textit{Shuffled}\big)$, see Fig. 5.14), then the composed elementary recognizers can be in the following global states: (i) all are idle (state s_0); (ii) all are waiting (in state s_1), (iii) some recognizers are counting their names (state s_3) and others are still waiting for their names to come (state s_2), (iv) all recognizers are counting their names (state s_3). The stopping mechanism of the recognizers is defined by the semantics $\mathtt{s}$ inherited from the parent fragment $\mathcal{F}$, it is defined by the rules described above.

Continuous Recognizers The continuous recognizer of a fragment $\mathcal{F} = (\{\mathcal{R}_1, \ldots, \mathcal{R}_m\}, \sharp, \sqcup\!\sqcup)$ is the synchronous parallel composition of the continuous elementary recognizers of the ranges $\mathcal{R}_j$s. The continuous elementary recognizers (re-initializing, stopping or resetting) have the same set of global states as their error reporting counterparts (see above). If the recognizer of $\mathcal{F}$ is made of the re-initializing elementary recognizers of ranges, the recognizer re-initializes when the recognizes of the ranges of $\mathcal{F}$ re-initialize. Equation 5.3 (resp. Equation 5.4) defines the *re-initializing recognizer* of $\mathcal{F}$ with the non-shuffled (resp. shuffled) semantics.

$$\mathbb{F}^{\textit{re-init}}_{\textit{non-shuffled}} = \mathbb{R}^{\textit{re-init}}_{\textit{non-shuffled},1} \times \cdots \times \mathbb{R}^{\textit{re-init}}_{\textit{non-shuffled},m} \tag{5.3}$$

$$\mathbb{F}^{\textit{re-init}}_{\textit{shuffled}} = \mathbb{R}^{\textit{re-init}}_{\textit{shuffled},1} \times \cdots \times \mathbb{R}^{\textit{re-init}}_{\textit{shuffled},m} \tag{5.4}$$

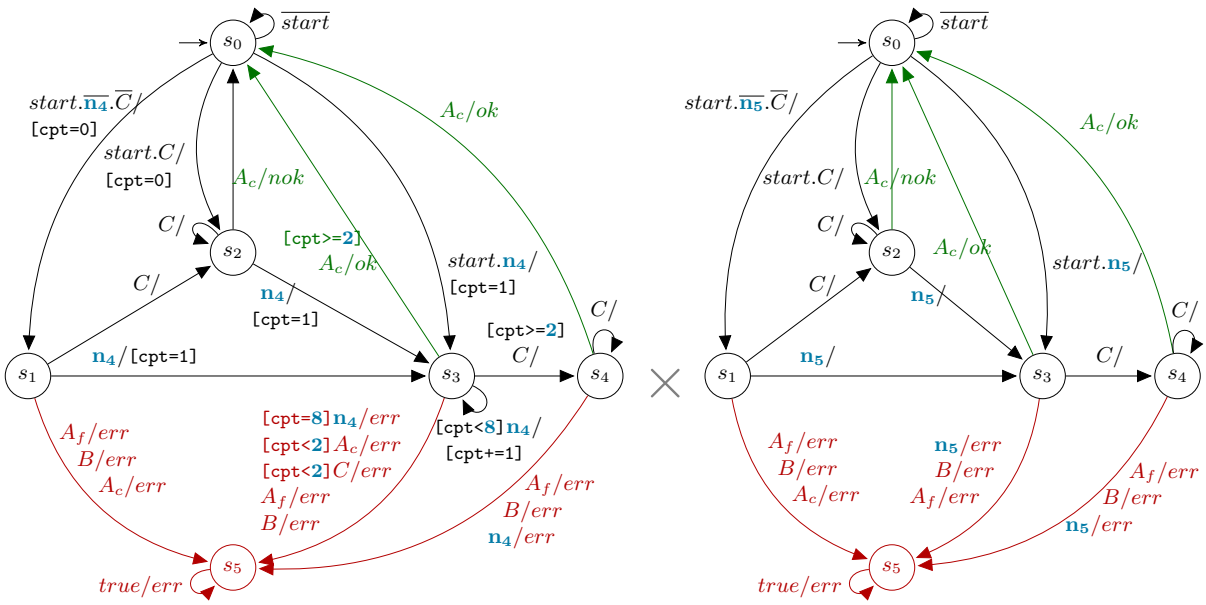


Figure 5.13 The synchronous product of the elementary recognizers of ranges of a fragment $\mathcal{F} = (\{n_4^{[2,8]}, n_5\}, \vee, \textit{Non-Shuffled})$ with error states.

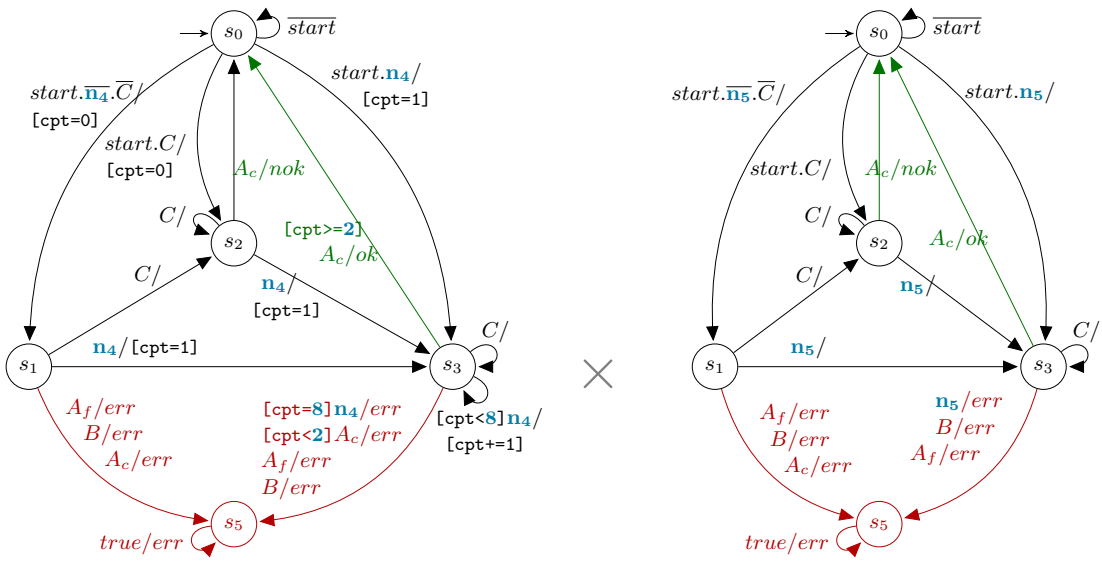


Figure 5.14 The synchronous product of the elementary recognizers of ranges of a fragment $\mathcal{F} = (\{n_4^{[2,8]}, n_5\}, \vee, \textit{Shuffled})$ with error states.

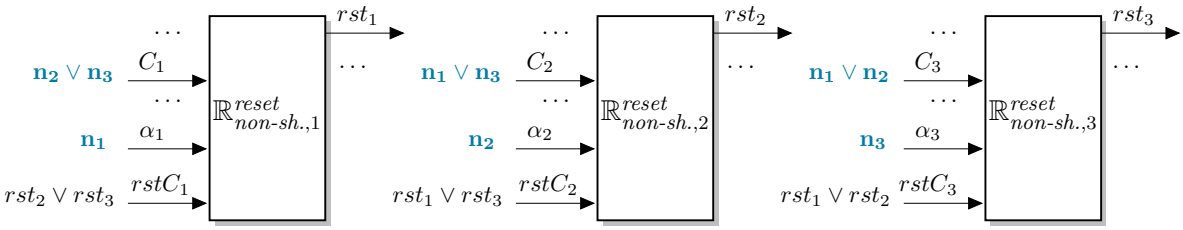
Here, for $j \in [1, m]$, $\mathbb{R}^{\textit{re-init}}_{\textit{non-shuffled},j}$ (resp. $\mathbb{R}^{\textit{re-init}}_{\textit{shuffled},j}$) is the continuous re-initializing recognizer of ranges (see Sec. 5.4.1.2).

The *resetting recognizer* of a fragment is the synchronous parallel composition of the resetting elementary recognizers of ranges. For example, Figure 5.15(a) shows a circuit diagram of the recognizer for a fragment:

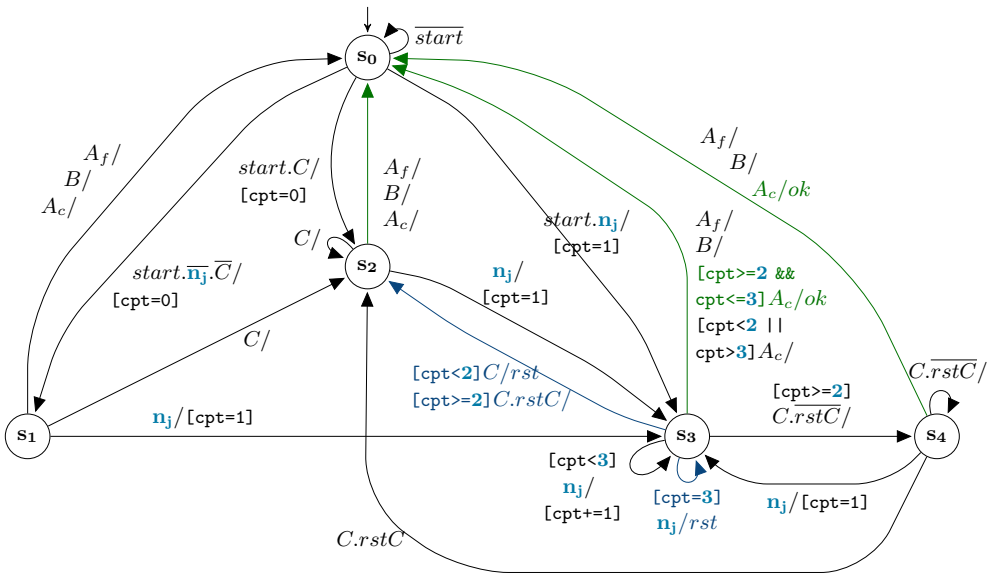
$$\mathcal{F} = \left(\{n_1^{[2,3]}, n_2^{[2,3]}, n_3^{[2,3]}\}, \wedge, \textit{Non-Shuffled}\right)$$

which is the synchronous parallel composition of the *resetting* elementary recognizers (Fig. 5.15(b)) of the ranges $n_j^{[2,3]}$ for all $j \in [1, 3]$. Resetting of the recognizers works as follows:

1. The elementary recognizer resets the recognizers of other ranges when, in state s_3, it detects more occurrences of its name than it is specified by a range (e.g., n_1 occurs 4 times). Those ranges need to occur again. The reset is done by producing *rst* output.



(a) A circuit diagram of the synchronous product.



(b) The resetting elementary recognizer of a range

Figure 5.15 The synchronous product of the resetting elementary recognizers of ranges of $\mathcal{F} = (\{n_1^{[2,3]}, n_2^{[2,3]}, n_3^{[2,3]}\}, \wedge, \textit{Non-Shuffled})$. $\mathbb{R}^{reset}_{non\text{-}shuffled}$ is a recognizer of $n_j^{[2,3]}$ for $j \in [1,3]$.

2. Any elementary recognizer of a range (e.g., $n_2^{[2,3]}$) can be reset either if it has already detected an occurrence of the range (i.e., the recognizer is in s_4), or if any of the companion ranges starts (e.g., $n_1^{[2,3]}$) before the recognizer detects the minimum number of occurrences of name of its range.

Formally, the *resetting recognizer* of a fragment with the non-shuffled semantics is defined in Equation 5.5.

$$\mathbb{F}^{reset}_{non\text{-}shuffled} = \left(\mathbb{R}^{reset}_{non\text{-}shuffled,1} \times \cdots \times \mathbb{R}^{reset}_{non\text{-}shuffled,m}\right) \backslash [rst_1, \ldots, rst_m] \tag{5.5}$$

Here, for $j \in [1,m]$, $\mathbb{R}^{reset}_{non\text{-}shuffled,j}$ is the resetting recognizer of ranges (see Sec. 5.4.1.2), "\" is the encapsulation operator (see Sec. 2.2.1.2 in the background chapter). We do not define the resetting recognizer of a fragment $\mathcal{F}$ with shuffled semantics.

Similarly, we define the *stopping recognizer* of a fragment with with non-shuffled (resp. shuffled) semantics in Equation 5.6 (resp. Equation 5.7).

$$\mathbb{F}^{stop}_{non\text{-}shuffled} = \left(\mathbb{R}^{stop}_{non\text{-}shuffled,1} \times \cdots \times \mathbb{R}^{stop}_{non\text{-}shuffled,m}\right) \backslash [stop_1, \ldots, stop_m] \tag{5.6}$$

$$\mathbb{F}^{stop}_{shuffled} = \left(\mathbb{R}^{stop}_{shuffled,1} \times \cdots \times \mathbb{R}^{stop}_{shuffled,m}\right) \backslash [stop_1, \ldots, stop_m] \tag{5.7}$$

Here, for any $j \in [1,m]$, $\mathbb{R}^{stop}_{non\text{-}shuffled,j}$ (resp. $\mathbb{R}^{stop}_{shuffled,j}$) is the stopping continuous recognizer of ranges (see Sec. 5.4.1.2).

Figure 5.16 shows interfaces (inputs and outputs) of the recognizers of fragments of different types.

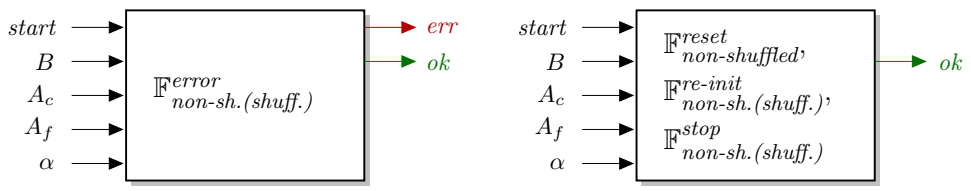


Figure 5.16 Circuit diagrams of the recognizers of fragments. Arrows from the left are inputs, arrows to the right are outputs.

5.4.2.2 Recognizers of Loose-Orderings

Consider a loose-ordering $\mathcal{L} = \mathcal{F}_1 < \cdots < \mathcal{F}_\ell$. The recognizer for $\mathcal{L}$ is made by composing *sequentially* the recognizers of the $\mathcal{F}_k$s: to start recognizing $\mathcal{L}$, we have to send *start* to the recognizer of fragment $\mathcal{F}_1$; for any $k \in [1, \ell - 1]$, the output *ok* of the recognizer of $\mathcal{F}_k$ is connected to the input *start* of the recognizer of $\mathcal{F}_{k+1}$. The output *ok* of the last fragment signals the stop of the recognizer of $\mathcal{L}$. Notice that, at any time only one of the fragments' recognizers can have its ranges in a non-idle state.

The **error reporting recognizer** of $\mathcal{L}$ is composed from the error reporting recognizers of the $\mathcal{F}_k$s. It signals an error when any of the recognizers of the $\mathcal{F}_k$s detects an error.

The **continuous recognizer** of $\mathcal{L}$ is composed from the continuous recognizers of the $\mathcal{F}_k$s, such that the recognizer of the first fragment $\mathcal{F}_1$ is *resetting*, the recognizer of the last fragment $\mathcal{F}_\ell$ is *stopping*, and the recognizers for other fragments are *re-initializing*.

Figure 5.17 shows a circuit diagram of the continuous recognizer of a loose-ordering

$$\mathcal{L} = \big(\{n_1^{[2,3]}, n_2^{[2,3]}, n_3^{[2,3]}\}, \wedge, \textit{Non-Shuffled}\big) < \big(\{n_4^{[2,8]}, n_5\}, \vee, \textit{Non-Shuffled}\big) < n_6.$$

Here, $\mathbb{F}^{reset}_{non\text{-}shuffled,1}$ is the recognizer of the first fragment defined in Figure 5.15(a) (see Sec. 5.4.2.1), $\mathbb{F}^{re\text{-}init}_{non\text{-}shuffled,2}$ is the re-initializing recognizer of the second non-shuffled fragment of $\mathcal{L}$, $\mathbb{F}^{stop}_{non\text{-}shuffled,3}$ is the stopping fragment of the third non-shuffled fragment of the loose-ordering.

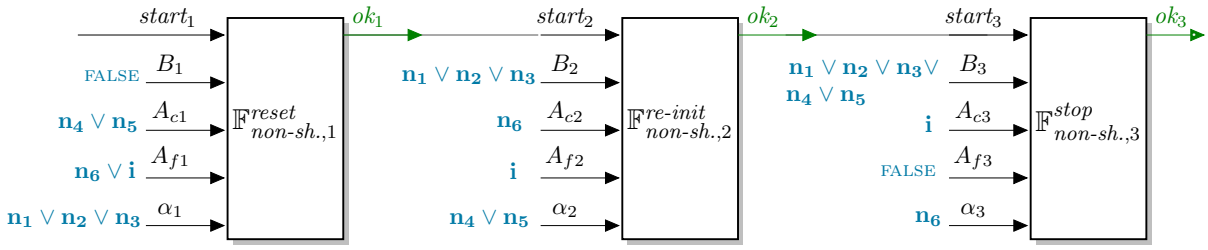


Figure 5.17 A circuit diagram of the recognizer of a loose-ordering $\mathcal{L}$.

5.4.3 Recognizers of Loose-Ordering Properties

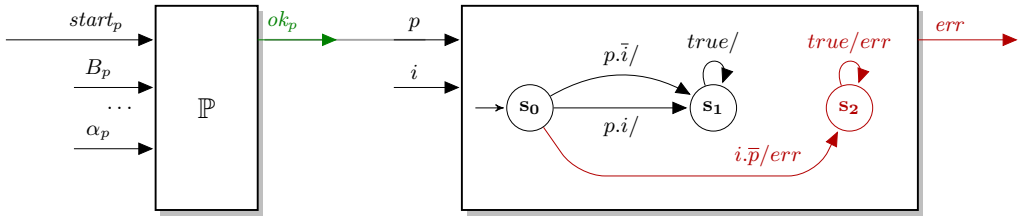
The recognizers of loose-ordering properties $\mathcal{A} = \big(\mathcal{P} \ll i \mid \nabla\big)$ and $\mathcal{T} = \big(\mathcal{P} \Longrightarrow \mathcal{Q} \mid t\big)$ use the continuous recognizer of a loose-ordering $\mathcal{P}$ to detect its occurrences. The recognizer of $\mathcal{A}$ uses this information to check the allowance of occurrences of i. The recognizer of $\mathcal{T}$ when $\mathcal{P}$ is detected may start the error reporting recognizer of a loose-ordering $\mathcal{Q}$.

5.4.3.1 A Recognizer of an Antecedent Requirement

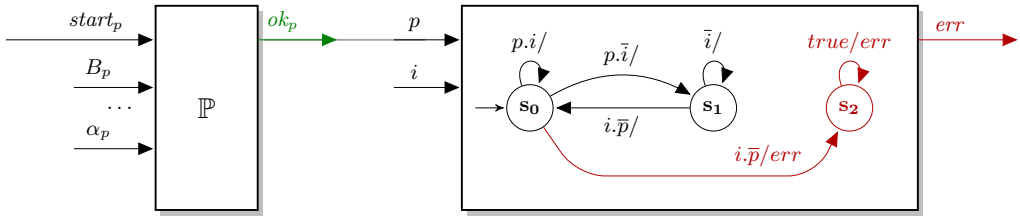
The recognizer of an antecedent requirement $\mathcal{A} = \big(\mathcal{P} \ll i \mid \nabla\big)$ with the non-repeated context $\nabla = \textit{Non-Repeated}$ whenever i occurs, checks if $\mathcal{P}$ has been recognized at least once before the first occurrence of i (see Fig. 5.18(a)). The recognizer of $\mathcal{A}$ with the repeated context $\nabla = \textit{Repeated}$ whenever i occurs, checks if $\mathcal{P}$ has been recognized at least once since the previous occurrence of i (Fig. 5.18(b)). If $\mathcal{P}$ has not been detected, both recognizers report errors. The recognizer of $\mathcal{A}$ is continuous.

5.4.3.2 A Recognizer of a Timed Implication Constraint

The recognizer of a timed implication constraint $\mathcal{T} = \big(\mathcal{P} \Longrightarrow \mathcal{Q} \mid t\big)$ is made by composing sequentially the continuous recognizer of $\mathcal{P}$ and the error reporting recognizer of $\mathcal{Q}$ (Fig. 5.19). When the condition is



(a) A non-repeated context: $\nabla = \textit{Non-Repeated}$



(b) A repeated context: $\nabla = \textit{Repeated}$

Figure 5.18 A circuit diagram of a recognizer of an antecedent requirement $\mathcal{A} = (\mathcal{P} \ll i \mid \nabla)$: $\mathbb{P}$ is the continuous recognizer of $\mathcal{P}$.

fulfilled (i.e., an occurrence of $\mathcal{P}$ is detected), the checking of promises is started (i.e., the recognizer of $\mathcal{Q}$ is started). When the recognizer of $\mathcal{Q}$ is active, it ignores the start. The recognizer of $\mathcal{T}$ reports an error if the loose-ordering $\mathcal{Q}$ is violated, i.e., the respective recognizer enters the error state. The recognizer of $\mathcal{T}$ is continuous.

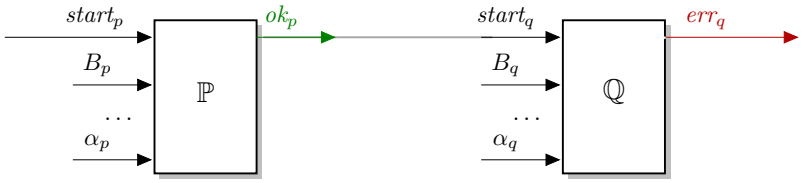


Figure 5.19 A circuit diagram of the recognizer for a timed implication constraint $\mathcal{T} = (\mathcal{P} \Longrightarrow \mathcal{Q} \mid t)$: $\mathbb{P}$ (resp. $\mathbb{Q}$) is the continuous (resp. error reporting) recognizer of $\mathcal{P}$ (resp. $\mathcal{Q}$).

5.5 Validation

The elementary recognizers with contexts of ranges, and the synchronous compositions defined in Sections 5.4.2 and 5.4.3 were programmed in LUSTRE[1] [Cas+87; Hal+91; EJ] . The correctness of these constructions with respect to the intuitive semantics given in Section 4.4 has been checked with a formal testing tool.

5.5.1 Implementing Recognizers in LUSTRE

The implementation principle is the following: Each elementary recognizer of ranges is implemented as a LUSTRE node. Inputs (resp. outputs) of the elementary recognizer are inputs (resp. outputs) of the node. The implementation is the set of Boolean equations encoding the recognizer. We define one Boolean variable per state, and one Boolean variable per output. We use LUSTRE *asserts* (see Sec. 2.2.1.4) to ensure that names of loose-ordering properties do not occur simultaneously. Figure 5.20 provides the LUSTRE implementation of the resetting continuous elementary recognizer of ranges.

We implement recognizers of fragments (resp. loose-orderings, an antecedent requirement, a timed implication constraint) with the dataflow connections of the LUSTRE nodes implementing the composed recognizers. For example, Figure 5.21 provides the LUSTRE implementation of the resetting continuous recognizer of a fragment

$$\mathcal{F} = \big(\{n_1^{[2,3]}, n_2^{[2,3]}, n_3^{[2,3]}\}, \wedge, \textit{Non-Shuffled}\big).$$

[1] LUSTRE is the synchronous language (see Sec. 2.2.1.4).

The synchronous product of the resetting elementary recognizers of the ranges $n_j^{[2,3]}$ (for all $j \in [1,3]$) corresponds to the dataflow connections of the LUSTRE nodes `range_reset_non_shuffled` implementing those recognizers (see Fig. 5.20).

5.5.2 Obtaining Monitors from the LUSTRE Implementation

The LUSTRE compiler generates a C implementation from the Lustre implementation. The library of monitors obtained in such a way can be then used for checking loose-ordering properties of the design at simulation. It can cause difficulties though, for instance, if one needs to instantiate a monitor which is not in a library. The monitors produced by the LUSTRE compiler are not compositional, i.e., the C implementation cannot be reused to build monitors by hand. Moreover, writing LUSTRE code is not a "usual" skill of the engineers. To overcome the problem, in the next chapter we provide a SystemC library of the primitive monitors of parts of loose-ordering properties (ranges, fragment, etc.) and show how monitors of arbitrary loose-ordering properties can be built from those primitive monitors.

```
1  node range_reset_non_shuffled(u, v: int;
2                                s, start, n, C, Ac, Af, B, rst_c: bool)
3  returns (s0, s1, s2, s3, s4, rst,  ok,  nok: bool);
4  var cpt: int;
5  let
6  -- ensuring that names do not occur simultaneously
7  assert(#(B, Ac));  assert(#(B, Af)); assert(#(B, C));  assert(#(B, n));
8  assert(#(Ac, Af)); assert(#(Ac, C)); assert(#(Ac, n)); assert(#(Af, C));
9  assert(#(Af, n));  assert(#(C,  n));
10
11 -- encoding transitions
12 s0 = true -> pre(s0 and not(start)) or
13              pre((s1 or s2 or s3 or s4) and (Ac or B or Af));
14
15 s1 = false-> pre(s0 and start and not(n) and not(C)) or
16              pre(s1 and not(n or C or Ac or B or Af));
17
18 s2 = false-> pre(s0 and start and C) or
19              pre((s1 or s2) and C) or
20              pre(s3 and C and cpt<u) or
21              pre(s3 and C and rst_c and cpt>=u) or
22              pre(s4 and C and rst_c) or
23              pre(s2 and not(Ac or B or Af or n));
24
25 s3 = false-> pre(s0 and start and n) or
26              pre((s1 or s2 or s3 or s4) and n) or
27              pre(s3 and not(C or Ac or B or Af));
28
29 s4 = false-> pre(s3 and C and not(rst_c) and cpt>=u) or
30              pre(s4 and C and not(rst_c)) or
31              pre(s4 and not(n or Ac or B or Af or C and rst_c));
32
33 -- the counter
34 cpt = 0 -> if pre(s0 and start and not n) then 0 else
35            if pre((s0 or s1 or s2 or s4) and n) then 1 else
36            if pre(s3 and n and cpt < v) then pre(cpt)+1 else
37              pre(cpt);
38
39 -- the outputs
40 ok  = (s3 and Ac and (cpt>=u or cpt<=v)) or (s4 and Ac);
41 nok = (s2 and not(s) and Ac);
42 rst = (s3 and C and cpt<u) or (s3 and n and cpt=v);
43 tel
```

Figure 5.20 LUSTRE implementation of the resetting continuous elementary recognizer $\mathbb{R}^{reset}_{non\text{-}shuffled}$ of ranges.

Summary

This chapter has presented efficient recognizers for loose-orderings. Their efficiency is enabled by the syntactic constraints put on the loose-ordering language. The constraints state that different parts of

```
node fragment_reset_non_shuffled_and(start,  n1, n2, n3, Ac, Af, B: bool)
returns (s01, s11, s21, s31, s41,
         s02, s12, s22, s32, s42,
         s03, s13, s23, s33, s43, ok: bool);
var rst1,  rst2, rst3,
    ok1, ok2, ok3,
    nok1, nok2, nok3,
    s: bool;
    u1, v1, u2, v2, u3, v3: int;
let
s=true;
u1=2; v1=3; u2=2; v2=3; u3=2; v3=3;

(s01, s11, s21, s31, s41, rst1, ok1,  nok1) =
      range_reset_non_shuffled(u1, v1, s, start, n1, n2 or n3, Ac, Af, B,
                               rst2 or rst3);

(s02, s12, s22, s32, s42, rst2, ok2, nok2) =
      range_reset_non_shuffled(u2, v2, s, start, n2, n1 or n3, Ac, Af, B,
                               rst1 or rst3);

(s03, s13, s23, s33, s43, rst3, ok3, nok3) =
      range_reset_non_shuffled(u3, v3, s, start, n3, n1 or n2, Ac, Af, B,
                               rst1 or rst2);

--due to conjunctive semantics:
ok = ok1 and ok2 and ok3;
tel
```

Figure 5.21 LUSTRE implementation of the resetting recognizer of $\mathcal{F} = (\{n_1^{[2,3]}, n_2^{[2,3]}, n_3^{[2,3]}\}, \wedge, \textit{Non-Shuffled})$.

loose-orderings (i.e., fragments, ranges, etc.) do not share names. It means that one can "cut" a sequence of names on fragments in a unique way, and define a recognition context for each of those fragments.

The recognizers of loose-orderings are built compositionally from the elementary recognizers for ranges. The elementary recognizers work in a recognizer context. In this chapter, we have shown the implementation of the recognizers and their compositions in LUSTRE; the implementation was used to ensure the correctness of the recognizers. In the next chapter, we show how to implement the recognizers in SystemC, and how to use the obtained SystemC monitors for capturing synchronization bugs of SystemC/TLM virtual prototypes.

Chapter 6

Monitoring Principles and SystemC Implementation

Contents

6.1 Introduction **107**
6.1.1 Challenges in Monitoring SystemC/TLM Models 107
6.1.2 Monitoring Loose-Ordering Properties 108
6.2 A Library of Primitive Monitors: SystemC Implementation **110**
6.2.1 Attribute Grammar of the Language of Loose-Orderings 110
6.2.2 SystemC Implementation 113
6.3 Monitors of Loose-Ordering Properties **118**
6.4 Monitorable SystemC/TLM Channels **119**
6.4.1 Observable Signal Channels 119
6.4.2 Observable SystemC/TLM Sockets 120

In this chapter, we present our principles for monitoring SystemC/TLM models. The chapter describes a direct translation of loose-ordering properties into efficient SystemC monitors. The translation relies on the attribute grammar of the loose-ordering language, and the recognizers of its parts (loose-orderings, fragments, ranges). We show how to build a monitor on the example of one of our running example's properties. In Chapter 7, we compare the time and memory complexities of the monitors obtained through our direct translation with the respective complexities of monitors corresponding to the PSL encoding of those properties.

6.1 Introduction

Our goal is to check at simulation time that a SystemC/TLM model of the system conforms its specification, i.e., it satisfies all defined properties. It can be done by means of *monitoring*. The idea is to define a *monitor* of a property and, when the simulation is running, make the monitor to check the property at certain *evaluation points*. Although monitoring is well studied for the RTL design [Men; Syna; Cad; Aba+00; MAB07b; MAB06], it remains challenging, when the design is defined at the TLM abstraction level.

6.1.1 Challenges in Monitoring SystemC/TLM Models

6.1.1.1 Non-Intrusive Monitors

Monitoring SystemC/TLM models should not be intrusive. This is because additional code may change the behavior of the examined SystemC/TLM model. This is particularly true in the case of monitors defined as SystemC modules. To make monitors non-intrusive, for instance, aspect-oriented approaches were investigated [NHd06; Kal+09; Kal+10]. Thus, [NHd06] was the first proposed methodology that allowed direct Assertion-Based Verification (ABV) of the TLM without additional abstract models. The key idea of the approaches [NHd06; Kal+09; Kal+10] is the following: Monitors are defined as *aspects*.

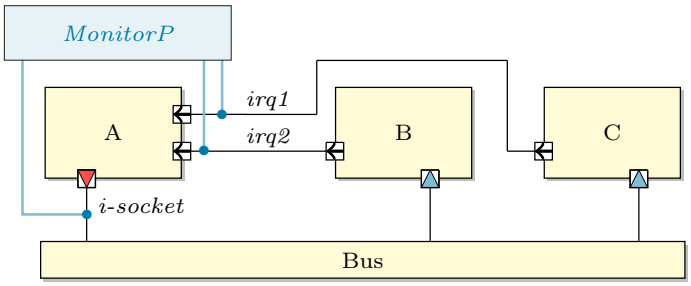


Figure 6.1 A TLM platform with a monitor: *MonitorP* is a monitor of a component A which checks a loose-ordering property P.

An aspect encapsulates methods, called advices. An advice is an action activated by an aspect before, after or around (in substitution of) the execution of the code addressed by the corresponding *join point*. A join point is a point in the control flow (e.g., a method execution). Monitors defined by aspects are weaved into SystemC code at compile time or at runtime by an aspect weaver [SGSP02]. Although aspect-oriented methods of checking properties are claimed to be non-intrusive, they are restrictive. They either can be applied to only certain categories of TL models [NHd06], or report results of checking relatively simple temporal properties consisting of only one temporal operator [Kal+09; Kal+10]. Aspect-oriented programming elements are available in HVLs [1] *e* [IJ04] and *OpenVera* [Synb] from Synopsys.

In [PF08; Fer11] authors propose a relatively non-intrusive approach. The original SystemC code undergoes few modifications that essentially correspond to (i) defining straightforward subclasses for the communication channels under observation, and (ii) using these inherited classes instead of their parent classes. The inherited classes should override all the methods related to the relevant events. Monitors are defined as C++ classes. They are registered in the inherited communication channels. At simulation, when events occur, the channels notify all registered monitors about those events.

6.1.1.2 Evaluation of Properties

The semantics of properties at the RT level is defined with regard to execution traces. At the synchronous RT level, the observations that constitute these traces are made of *clock edges*. Monitors defined at this level of abstraction (e.g., the *FoCs* checker generator [Aba+00; Foc; Aba+00; MH03] from IBM, the Horus technology [MAB07b]) evaluate properties at each edge of the clock.

When the hardware design is defined at *transactional level*, one needs to reconstruct the time scale from *sequence of events* in order to check properties. The concept of event is generally defined as "something happening during the evolution of the system" (e.g., sending/receiving a transaction). The meaning of event and sequence of events is settled depending on the type of the monitored properties. In [BFF05] and [BFP07] a sequence is defined based on the relation "happens before" proposed by Lamport in [Lam78] to order a sequence of events related to several processes of the design. [PF08] states that observation points are set when the properties needs to be reevaluated, i.e., each time a variable of the property has possibly been modified. The variable is modified, if an appropriate event occurs in the corresponding channel. A *sequence of events* is defined as a trace made of all the events that enable the observation of updated values for the variables involved in the property.

6.1.2 Monitoring Loose-Ordering Properties

Our monitoring model is inspired by [PF08; Fer11]. Our monitors are external to the SystemC/TLM design. They capture *events* occurring in the *communication channels* of TL components (Fig. 6.1). We consider communication channels for exchanging transactions and interrupts. The former are implemented by means of TLM initiator/target sockets (e.g., *i-socket* in Fig. 6.1). The latter are implemented by means of SystemC `sc_signal<bool>` channels (e.g., *irq1* and *irq2* in Fig. 6.1). Events are sending/receiving transactions/interrupts. A monitor may capture events of several communication channels of a component. Any TL component can (potentially) have several monitors. A monitor checks at simulation time the correctness of one loose-ordering property.

Let P be a loose-property of a component A. To enable online monitoring of P, one needs:

(i) to instantiate a monitor for P (e.g, *MonitorP*);

[1] HVL stads for *Hardware Verification Language*

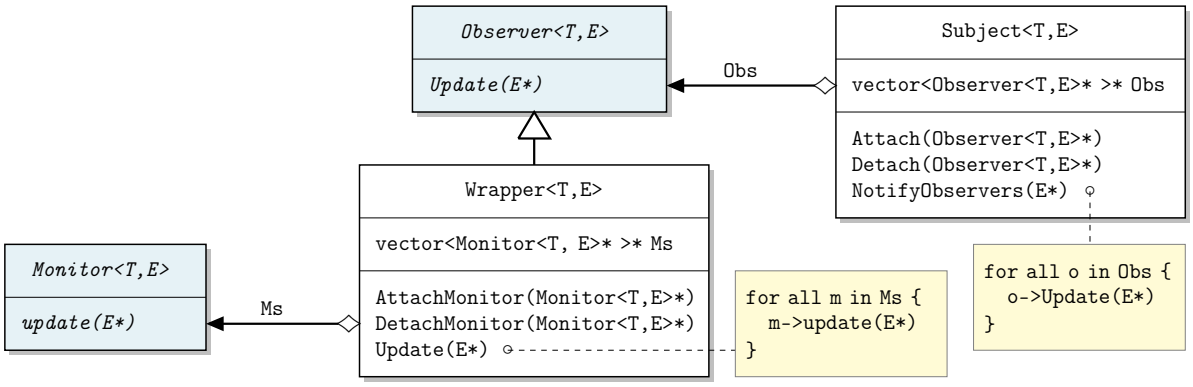


Figure 6.2 The monitoring model.

(ii) to instrument the SystemC/TLM model of A by substituting ordinary communication channels with their monitorable versions, i.e., channels which can register the monitor of P and notify it when communication events occur (in Fig. 6.1 *irq1*, *irq2* and *i-socket* are monitorable, see details below);

(iii) to define the correspondence between communication events and names of P.

At simulation time, (some) communication events may update the state of the monitor of P. If the monitor detects a violation of P, it stops the simulation and provides the information about an event(s) which caused the error. If P is of an "antecedent requirement" type, this information can be used by the designer to establish an initiator of a transaction or an interrupt that caused the error, i.e., the actual source of the design's malfunction. If P is of a "timed implication constraint" type, the description of the erroneous event may help to identify the exact point in the SystemC implementation of A which violated P.

6.1.2.1 The Monitoring Model

Figure 6.2 shows our model. It relies on the classical observer pattern [Gam+95] and it is inspired by [Pie07]. The abstract class `Subject` is inherited by monitorable channels. It maintains the set of observers `Obs` registered by means of the `Attach(...)` method. Observers are containers of monitors; they implement the ***Observer*** interface. When event e occurs, a channel (e.g., *irq1*) calls the `NotifyObservers(`e`)` method of the base `Subject` class to notify the registered observers about the event. When the method is invoked, the observers update states of their monitors. Monitors implement the interface ***Monitor*** which defines the `update(...)` method.

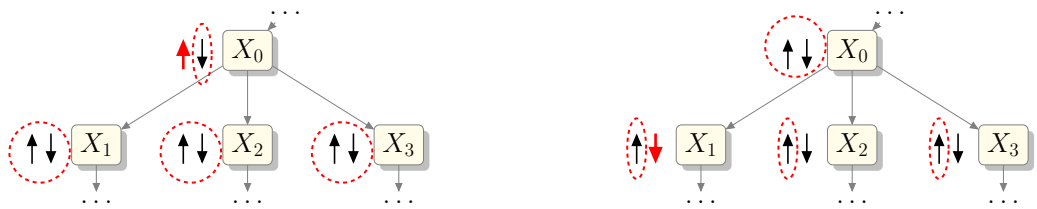
Our model is generic: all the abstract classes are C++ templates. They are parameterized by the *type of names* `T` of a loose-ordering property, and the *type of events* `E` occurring in the channels. Both types can be any user-defined structure.

6.1.2.2 Primitive Monitors of Loose-Ordering Properties

Our SystemC monitors of loose-ordering properties are compositional. We use the attribute grammar of the loose-ordering language proposed in Sec. 5.3; the SystemC implementation is a straightforward implementation of the grammar. Each node of the syntax tree of a loose-ordering formula is defined by a SystemC monitor. Leafs of the tree are SystemC monitors of ranges which implement corresponding elementary recognizers defined in Chapter 5 (see Sec. 5.4.1). Those monitors keep a recognition context. Based on the semantic rules of the attribute grammar, the constructed monitor of a loose-ordering property checks the well-formedness of the property. If the property is not well-formed, the SystemC monitor alerts and it cannot be used for on-line monitoring. If the property is well-formed, the recognition context for primitive monitors of ranges is evaluated. When the checking and initialization are finished, the monitoring can be started.

All monitors have a `step(...)` function. Its call is propagated downwards in the hierarchy of the monitors. The function is called each time when an event occurs, i.e., the state of the monitors needs to be updated. If any of the monitors detects an error, the composite monitor reports an error and provides information about the location of the detected malfunction.

To be used with our monitoring model (see Sec. 6.1.2.1 above), the primitive monitors of loose-orderings are embedded into classes which inherit the ***Monitor*** abstract class (Fig. 6.2). Those classes call



(a) Synthesized attributes of X_0 may depend on the attributes of X_1, X_2, X_3 and the inherited attributes of X_0

(b) Inherited attributes of X_1 may depend on the synthesized attributes of X_2, X_3 and all attributes of X_0

Figure 6.3 Semantic rules of an attributed grammar illustrated on a fragment of a derivation tree. $X_0, \ldots, X_3$ are nodes. The symbol "↓" (resp. "↑") denotes the inherited (resp. synthesized) attributes of X_i.

the `step(...)` method of the embedded monitors whenever the `update(...)` function inherited from the *`Monitor`* is invoked.

In the following sections we discuss in details SystemC implementation of the primitive monitors of loose-ordering properties (Sec. 6.2), their integration into our monitoring model (Sec. 6.3), and provide definition of monitorable communication channels (Sec. 6.4). For illustration purpose, we construct a monitor of one of the loose-orderings of the IPU and show how the monitor can be used to ensure correctness of the component.

6.2 A Library of Primitive Monitors: SystemC Implementation

To implement monitors of loose-ordering properties in SystemC, we use the attribute grammar of the loose-ordering language mentioned in Sec. 5.3. In this section we provide its formal definition. The grammar allows (i) to construct monitors compositionally, (ii) to generate monitors automatically, (iii) to compute recognition contexts for each part of the properties (ranges, fragments, etc.), (iv) to check the well-formedness of those monitors (i.e., the well-formedness of respective loose-ordering properties w.r.t. the definitions provided in Chapter 4). The grammar defines one production rule per *term* of the language (a range $\mathcal{R}$, a fragment $\mathcal{F}$, etc.). The attributes of the grammar are used to evaluate the recognition contexts of ranges and check the well-formedness. The SystemC implementation is the straightforward encoding of the grammar's production rules: classes represent terms of the grammar; they provide methods for the context evaluation and run-time monitoring. The evaluation of loose-ordering properties relies on the recognizers defined in Chapter 5.

6.2.1 Attribute Grammar of the Language of Loose-Orderings

The attribute grammar in Figure 6.4 defines the language of loose-orderings. It instantiates the abstract grammar for well-formed loose-ordering formulas provided in Figure 4.1 (see Chapter 4). The bold capital letters and symbols are non-terminals of the grammar. The set of production rules are highlighted in red.

Each non-terminal X is associated with two disjoint sets of attributes: $\uparrow I(X)$ stands for the set of inherited attributes of X, $\downarrow S(X)$ stands for the set of synthesized attributes of X. Each attribute takes a value from some associated domain (e.g., sets, integers). The attributes' values are defined by *rules* associated with the productions of the grammar. The rules of a production $X_0 \rightarrow X_1 \ldots X_k$ are as follows:

1. Each synthesized attribute in $S(X_0)$ is defined in terms of the attributes $I(X_0) \cup A(X_1) \cup \cdots \cup A(X_k)$ where for all $i \in [1, k]$, $A(X_i)$ is the set of all attributes of X_i (see Fig. 6.3(a)).
2. Each inherited attribute in $I(X_i)$ is defined in terms of the attributes in $A(X_0) \cup S(X_1) \cup \cdots \cup S(X_k)$ for all $i \in [1, k]$ (see Fig. 6.3(b))

Each production $X_0 \rightarrow X_1 \ldots X_k$ may have a set of conditions on the attributes' values. We denote the production rules of the attribute grammar as

$$
\begin{aligned}
X_0 \uparrow_{S(X_0)} \downarrow_{I(X_0)} \rightarrow & X_1 \ldots X_k \\
& < \textit{conditions over } A(X_0) \cup A(X_1) \cup \cdots \cup A(X_k) > \\
& [\ \textit{definition of } \forall x \in S(X_0) \cup I(X_1) \cup \cdots \cup I(X_k)\ ]
\end{aligned}
$$

$\mathcal{A} \uparrow_{\alpha} \rightarrow (\mathcal{P} \ll \mathbf{N} \mid \nabla)$
$< \alpha(\mathcal{P}) \cap \{val(\mathbf{N})\} = \varnothing; \quad val(\mathbf{N}) \in I >$
$[\ \alpha(\mathcal{A}) = \alpha(\mathcal{P}) \cup \{val(\mathbf{N})\};$
$B(\mathcal{P}) = \varnothing; \quad A_c(\mathcal{P}) = \{i\}; \quad A_f(\mathcal{P}) = \varnothing\ ]$

$\mathcal{T} \uparrow_{\alpha} \rightarrow (\mathcal{P} \Longrightarrow \mathcal{Q} \mid \mathbf{T})$
$[\ \alpha(\mathcal{G}) = \alpha(\mathcal{P}) \cup \alpha(\mathcal{Q});$
$B(\mathcal{P}) = \varnothing; \quad A_c(\mathcal{P}) = \sigma(\mathcal{Q}); \quad A_f(\mathcal{P}) = \alpha(\mathcal{Q}) \backslash \sigma(\mathcal{Q});$
$B(\mathcal{Q}) = \varnothing; \quad A_c(\mathcal{Q}) = \sigma(\mathcal{P}); \quad A_f(\mathcal{Q}) = \alpha(\mathcal{P}) \backslash \sigma(\mathcal{P})\ ]$

$\mathcal{P} \uparrow_{\alpha,\sigma} \downarrow_{B,A_c,A_f} \rightarrow \mathcal{L}$
$< \alpha(\mathcal{P}) \subseteq I \cup O >$
$[\ \alpha(\mathcal{P}) = \alpha(\mathcal{L}); \qquad \sigma(\mathcal{P}) = \alpha(\mathcal{L});$
$B(\text{resp. } A_c, A_f)(\mathcal{L}) = B(\text{resp. } A_c, A_f)(\mathcal{P}); \quad idx(\mathcal{L}) = 0\ ]$

$\mathcal{Q} \uparrow_{\alpha,\sigma} \downarrow_{B,A_c,A_f} \rightarrow \mathcal{L}$
$< \alpha(\mathcal{Q}) \subseteq O >$
$[\ \alpha(\mathcal{Q}) = \alpha(\mathcal{L}); \qquad \sigma(\mathcal{Q}) = \alpha(\mathcal{L});$
$B(\text{resp. } A_c, A_f)(\mathcal{L}) = B(\text{resp. } A_c, A_f)(\mathcal{Q}); \quad idx(\mathcal{L}) = 0\ ]$

$\mathcal{L} \uparrow_{\alpha,\sigma} \downarrow_{B,A_c,A_f,idx} \rightarrow$
$\mathcal{F}$
$[\ \alpha(\mathcal{L}) = \alpha(\mathcal{F}); \qquad \sigma(\mathcal{L}) = \alpha(\mathcal{F});$
$B(\mathcal{F}) = B(\mathcal{L}); \ A_c(\mathcal{F}) = A_c(\mathcal{L}); \ A_f(\mathcal{F}) = A_f(\mathcal{L}); \ idx(\mathcal{F}) = idx(\mathcal{L}) + 1\ ]$
$\mid \mathcal{F} < \mathcal{L}'$
$< \alpha(\mathcal{F}) \cap \alpha(\mathcal{L}') = \varnothing >$
$[\ \alpha(\mathcal{L}) = \alpha(\mathcal{F}) \cup \alpha(\mathcal{L}'); \quad \sigma(\mathcal{L}) = \alpha(\mathcal{F});$
$B(\mathcal{F}) = B(\mathcal{L}); \quad A_c(\mathcal{F}) = \sigma(\mathcal{L}'); \quad A_f(\mathcal{F}) = \alpha(\mathcal{L}') \backslash \sigma(\mathcal{L}') \cup A_c(\mathcal{L}) \cup A_f(\mathcal{L});$
$idx(\mathcal{F}) = idx(\mathcal{L}) + 1;$
$B(\mathcal{L}') = B(\mathcal{L}) \cup \alpha(\mathcal{F}); \ A_c(\mathcal{L}') = A_c(\mathcal{L}); \quad A_f(\mathcal{L}') = A_f(\mathcal{L});$
$idx(\mathcal{L}') = idx(\mathcal{L}) + 1\ ]$

$\mathcal{F} \uparrow_{\alpha} \downarrow_{B,A_c,A_f,idx} \rightarrow (\{\mathcal{X}\}, \sharp, \sqcup\!\sqcup)$
$[\ \alpha(\mathcal{F}) = \alpha(\mathcal{X});$
$B(\text{resp. } A_c, A_f)(\mathcal{X}) = B(\text{resp. } A_c, A_f)(\mathcal{F}); \quad C(\mathcal{X}) = \varnothing;$
$\beta(\mathcal{X}) = \varnothing; \quad s(\mathcal{X}) = val(\sharp); \quad sh(\mathcal{X}) = val(\sqcup\!\sqcup)\ ]$

$\mathcal{X} \uparrow_{\alpha} \downarrow_{B,A_c,A_f,C,\beta,s,sh} \rightarrow$
$\mathcal{R}$
$[\ \alpha(\mathcal{X}) = \alpha(\mathcal{R});$
$B(\text{resp. } A_c, A_f, s, sh)(\mathcal{R}) = B(\text{resp. } A_c, A_f, s, sh)(\mathcal{X}); \ C(\mathcal{R}) = \beta(\mathcal{X})\ ]$
$\mid \mathcal{R}, \mathcal{X}'$
$< \alpha(\mathcal{R}) \cap \alpha(\mathcal{X}') = \varnothing >$
$[\ \alpha(\mathcal{X}) = \alpha(\mathcal{R}) \cup \alpha(\mathcal{X}');$
$B(\text{resp. } A_c, A_f, s, sh)(\mathcal{R}) = B(\text{resp. } A_c, A_f, s, sh)(\mathcal{X}); \quad C(\mathcal{R}) = \beta(\mathcal{X}) \cup \alpha(\mathcal{X}');$
$B(\text{resp. } A_c, A_f, s, sh)(\mathcal{X}') = B(\text{resp. } A_c, A_f, s, sh)(\mathcal{X}); \quad C(\mathcal{X}') = \varnothing;$
$\beta(\mathcal{X}') = \beta(\mathcal{X}) \cup \alpha(\mathcal{R})\ ]$

$\mathcal{R} \uparrow_{\alpha} \downarrow_{B,A_c,A_f,C,s,sh} \rightarrow \mathbf{N}^{[\mathbf{U},\mathbf{V}]}$
$[\ \alpha(\mathcal{R}) = \{val(\mathbf{N})\}\ ]$

Figure 6.4 The attribute grammar defining the language of loose-orderings.

6.2.1.1 The Non-Terminals

The bold capital letters and symbols in Figure 6.4 are non-terminals. Their meaning is the following:

(i) $\boldsymbol{\mathcal{R}}$ (resp. $\boldsymbol{\mathcal{F}}$, $\boldsymbol{\mathcal{L}}$) defines a range (resp. a fragment, a loose-ordering);

(ii) $\boldsymbol{\mathcal{P}}$ (resp. $\boldsymbol{\mathcal{Q}}$) defines a loose-ordering made of both inputs and outputs (resp. only outputs);

(iii) $\boldsymbol{\mathcal{A}}$ (resp. $\boldsymbol{\mathcal{T}}$) defines an antecedent requirements (resp. a timed implication constraint);

(iv) the auxiliary non-terminal $\boldsymbol{\mathcal{X}}$ defines the set of ranges;

(v) the non-terminals $\mathbf{N}$, $\boldsymbol{\nabla}$, $\mathbf{T}$, $\mathbf{U}$, $\mathbf{V}$ represent respectively the names of the formulas of $(I \cup O)$, the context of an antecedent requirement (repeated or non-repeated), time of a timed implication constraint, lower and upper bounds of a range. Their definition is not shown for convenience. To get their respective values, we use the $val(...)$ function. $\mathbf{T}$, $\mathbf{U}$ and $\mathbf{V}$ define integers. $\boldsymbol{\nabla}$ defines values of the set $\{\textit{Non-Repeated}, \textit{Repeated}\}$;

(vi) the non-terminals $\sharp$, $\sqcup\!\sqcup$ define the semantics of a fragment: conjunctive or disjunctive, shuffled or non-shuffled; their values are accessed by means of the $val(...)$ function. $\sharp$ defines Boolean values, $\sqcup\!\sqcup$ defines values of the set $\{\textit{Non-Shuffled}, \textit{Shuffled}\}$.

6.2.1.2 The Attributes

The non-terminals corresponding to the terms of the loose-ordering language ($\boldsymbol{\mathcal{R}}$, $\boldsymbol{\mathcal{X}}$, etc.,) have inherited and synthesized attributes.

– *The inherited attributes:*

(i) the inherited attributes B, C, A_c and A_f correspond to the respective sets of names of a recognition context (see Sec. 5.3);

(ii) the inherited attributes s and sh of $\boldsymbol{\mathcal{R}}$ and $\boldsymbol{\mathcal{X}}$ define semantics of the parent $\boldsymbol{\mathcal{F}}$;

(iii) the inherited attribute β of the non-terminal $\boldsymbol{\mathcal{X}}$, representing a set of ranges $\{\mathcal{R}_1, \ldots, \mathcal{R}_m\}$, is used to compute the set of names C of *companion* ranges of a range $\mathcal{R}_j$ (for $j \in [1, m]$);

(iv) the inherited attribute idx of the non-terminals $\boldsymbol{\mathcal{F}}$ and $\boldsymbol{\mathcal{L}}$ is used to define the position of a fragment in a parent loose-ordering.

– *The synthesized Attributes:*

(i) the synthesized attribute α defines the vocabulary of a part of the formula represented by the respective non-terminal,

(ii) the synthesized attribute σ of the non-terminals $\boldsymbol{\mathcal{L}}$, $\boldsymbol{\mathcal{P}}$ and $\boldsymbol{\mathcal{Q}}$ representing a loose-ordering $\mathcal{F}_1 < \cdots < \mathcal{F}_\ell$ is used to evaluate the set of names A_c which immediately follow a fragment $\mathcal{F}_k$ of the loose-ordering (for $k \in [1, \ell]$).

6.2.1.3 The Semantic Rules and Conditions

The rules of the productions define the evaluation of the recognition context. The conditions ensure the well-formedness of the formula as it is defined in Section 4.4 (see Fig. 4.1). The conditions are checked after the vocabulary α is evaluated.

Example 6.2.1. – Consider an antecedent requirement

$$\Big(\big(\{n_1^{[2,3]}, n_2^{[10,15]}\}, \wedge, \textit{Non-Shuffled}\big) < n_3 \ll i \ \mid \textit{Non-Repeated}\Big).$$

Its parse tree is shown in Figure 6.5. For convenience (i) the non-terminals $\mathbf{N}$, $\boldsymbol{\nabla}$, $\mathbf{U}$ and $\mathbf{V}$ are omitted and their values are substituted directly in the respective nodes of the tree; (ii) indices are used to distinguish different occurrences (instantiations) of the non-terminals $\boldsymbol{\mathcal{L}}$, $\boldsymbol{\mathcal{F}}$, $\boldsymbol{\mathcal{X}}$ and $\boldsymbol{\mathcal{R}}$. In Figure 6.5 the occurrences of the non-terminals are shown the values of their attributes. The conditions are not illustrated since they all hold (the formula is well-formed). For each node t of the derivation tree the value of α synthesized at t is the union of the αs of the children nodes of t. The synthesized attribute σ of a node t' instantiating $\boldsymbol{\mathcal{L}}$ is equal to the set of names α of the left-most child node (which represents a fragment) of t'. All inherited attributes are propagated downward. They are re-evaluated (if needed) at each node according to the semantic rules defined in Figure 6.4. □

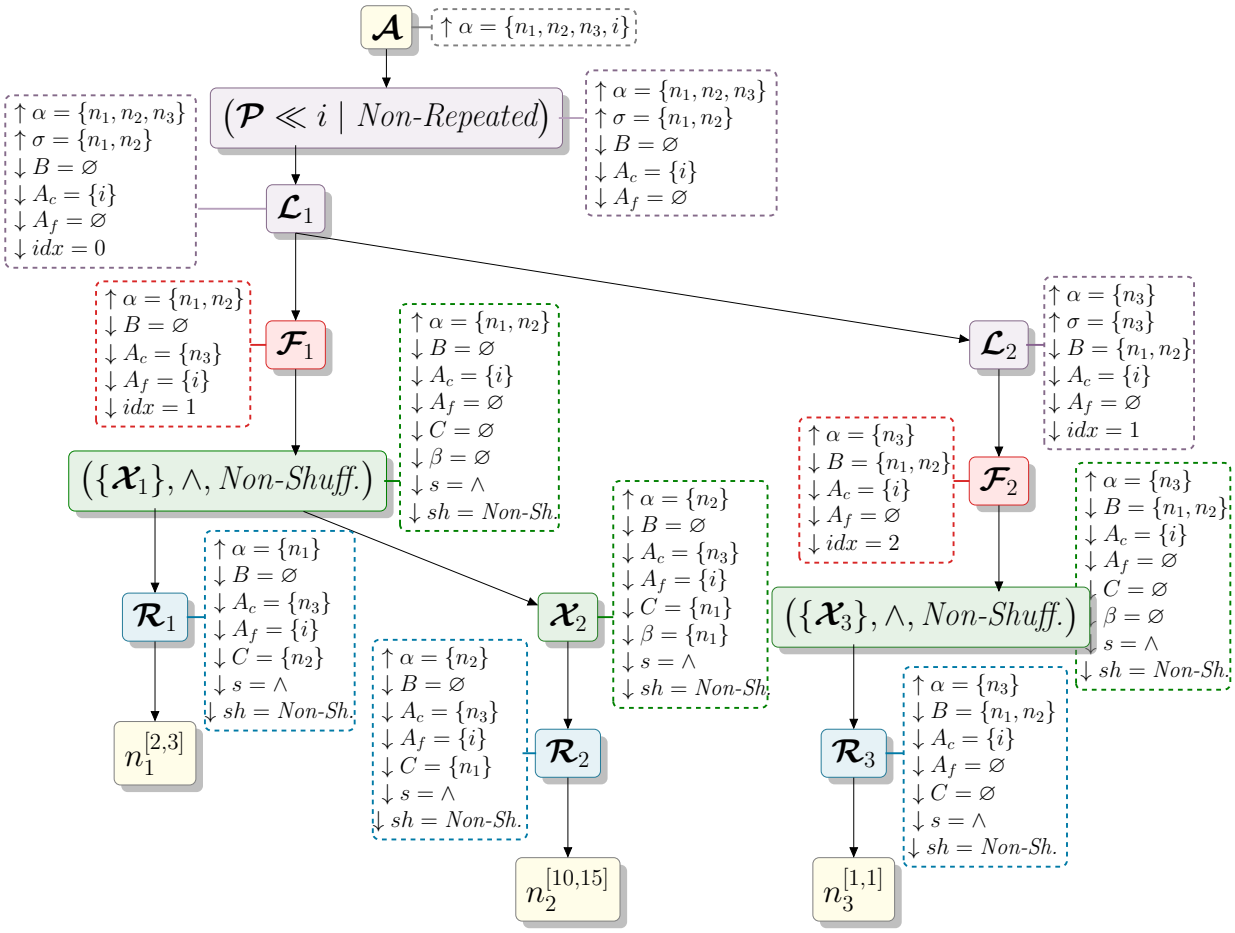


Figure 6.5 A derivation tree for an antecedent requirement $((\{n_1^{[2,3]}, n_2^{[10,15]}\}, \wedge, \textit{Non-Shuffled}) < n_3 \ll i \mid \textit{Non-Repeated})$. "$\uparrow x$" (resp. "$\downarrow y$") means that x (resp. y) is a synthesized (resp. inherited) attribute.

6.2.2 SystemC Implementation

The SystemC implementation of the primitive monitors of loose-orderings is straightforward. We define one class per non-terminal of the attribute grammar (see Fig. 6.6 and Fig. 6.7). Instantiations (objects) of those classes are primitive monitors of parts of a loose-ordering property (ranges, fragments, etc.). They keep pointers to the monitors of the children nodes of those parts in the syntax tree of a property (e.g., see Fig. 6.8). The implementation is generic; all the defined classes are C++ templates. They are parameterized with the *type of names* `T` constituting the loose-ordering properties.

6.2.2.1 Primitive Monitors and Recognition Context

The abstract class *`Range`* represents ranges; it corresponds to the non-terminal $\mathcal{R}$ of the attribute grammar. All the primitive monitors implementing elementary recognizers of ranges defined in Section 5.4.1 inherit from this abstract class (Fig. 6.6). Note that there is one primitive monitor per elementary recognizer. For instance, the class `RangeStopNonShuff` implements the continuous stopping recognizer of a range appearing in a fragment with non-shuffled semantics. Primitive monitors keep a pointer to a recognition context, a name of a range, a state of the monitor (it corresponds to the current state of the elementary recognizer), a counter, the parameters u and v of a range $n^{[u,v]}$.

A context is implemented by a class `Context`. It is a container of the sets of names defined by the recognition context (Sec. 5.3).

The class `RangesSet` defines the set of ranges; it corresponds to the non-terminal $\mathcal{X}$ of the grammar.

The class `Fragment` (Fig. 6.6) defines a monitor of a fragment; it corresponds to the non-terminal $\mathcal{F}$. The monitor has two Boolean variables `semantics` and `shuffle` defining the semantics of a fragment; the TRUE value of the former (resp. the latter) means that the fragment has *conjunctive* (resp. *shuffled*) semantics. The monitor possesses a pointer to a parent loose-ordering of the fragment, and an integer

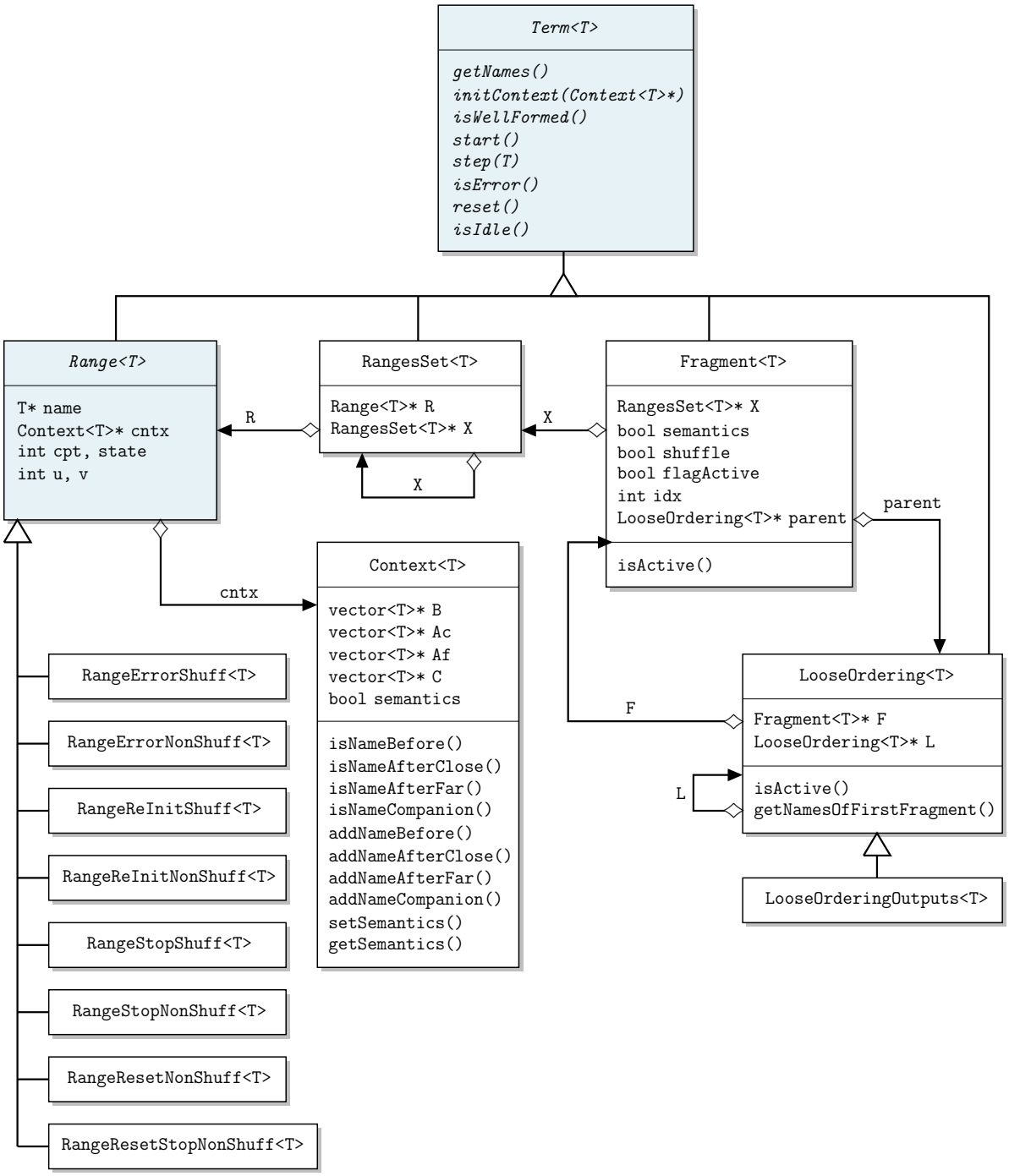


Figure 6.6 SystemC Implementation of the primitive monitors of loose-orderings: the class diagram. Part I.

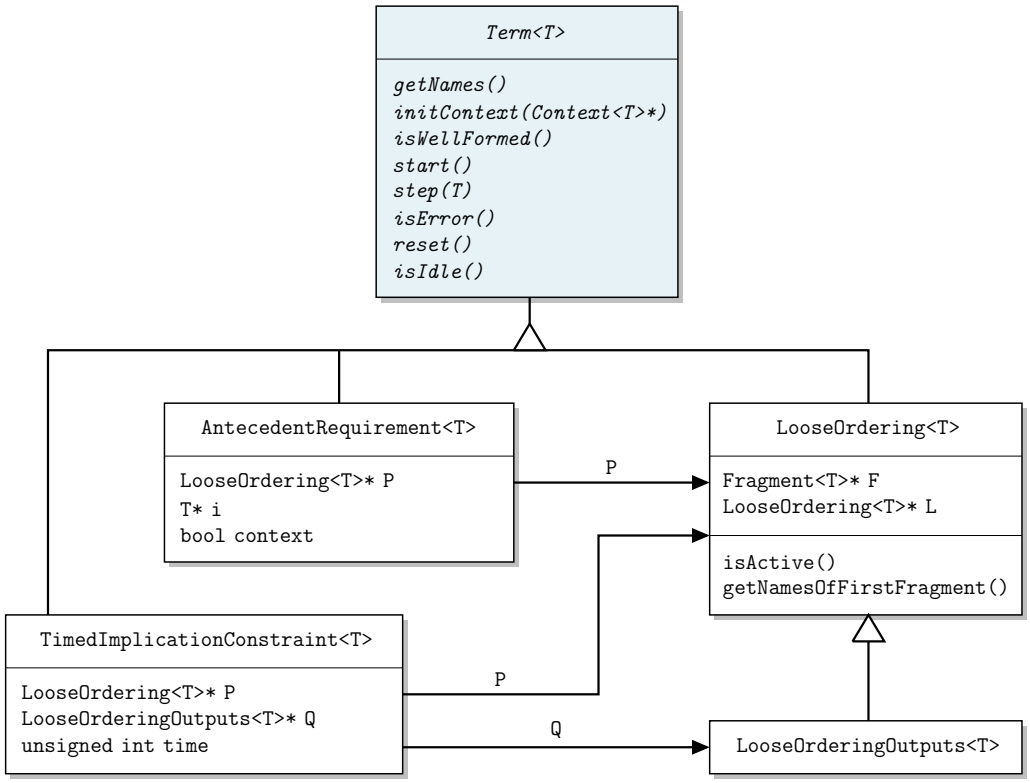


Figure 6.7 SystemC Implementation of the primitive monitors of loose-orderings: the class diagram. Part II.

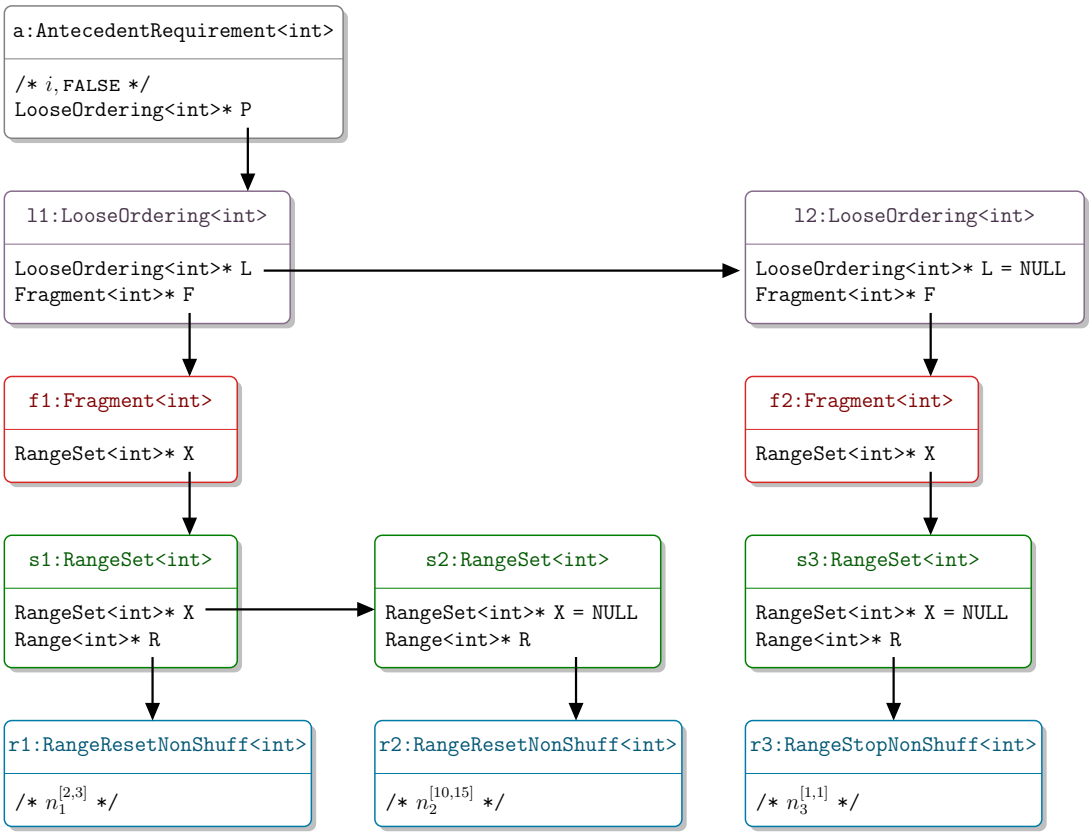


Figure 6.8 Structure of a monitor of the antecedent requirement $\big((\{n_1^{[2,3]}, n_2^{[10,15]}\}, \wedge, \textit{Non-Shuffled}) < n_3 \ll i \mid \textit{Non-Repeated}\big)$.

variable `idx` used to define the position of the fragment in the parent loose-ordering and to check the well-formedness of the monitor (see Sec. 6.2.2.5 below).

The `LooseOrdering` class defines a monitor of a loose-ordering; it corresponds to the non-terminals $\boldsymbol{\mathcal{L}}$ and $\boldsymbol{\mathcal{P}}$. The inherited class `LooseOrderingOutputs` defines a monitor of a loose-ordering made of only outputs; it represents the non-terminal $\boldsymbol{\mathcal{Q}}$ (Fig. 6.6).

The classes `AntecedentRequirement` and `TimedImplicationConstraint` correspond to the non-terminals $\boldsymbol{\mathcal{A}}$ and $\boldsymbol{\mathcal{T}}$ respectively (Fig. 6.7). They define monitors of loose-ordering properties. A monitor of an antecedent requirement $\mathcal{A} = \big(\mathcal{P} \ll i \mid \nabla\big)$ keeps a pointer to the checked name i, and a Boolean variable `context` defining the context; if it is TRUE, the context is *repeated*. A monitor of a timed-implication constraint $\mathcal{T} = \big(\mathcal{P} \Longrightarrow \mathcal{Q} \mid t\big)$ keeps an integer parameter `time` used to measure SystemC simulation time.

6.2.2.2 Interface of SystemC Monitors

Ranges, fragments, loose-orderings, etc. are *terms* of loose-ordering formulas. Their monitors implement the interface defined by the abstract class *`Tert`*:

(i) *`getNames()`*, *`initContext(Context<T>*)`*, *`isWellFormed()`* are methods used respectively to get names of a term, to compute a recognition context, to check conditions on a vocabulary of the term;
(ii) *`start()`* (resp. *`reset()`*) starts (resp. resets) a monitor;
(iii) *`isError()`* (resp. *`isIdle()`*) allows checking if a monitor is in error state (resp. idle);
(iv) the method *`step(T)`* checks if name of type `T` is allowed to occur; this is the method used for monitoring.

In addition to the inherited interface, some monitors define additional methods. Thus, the `Fragment` and `LooseOrdering` classes have a method `isActive()` which allows to check if a respective monitor is active.

6.2.2.3 Construction of a Monitor

A SystemC monitor of a loose-ordering property is constructed in a "bottom-up" way. First, primitive monitors of ranges are instantiated. Then, these monitors are used to construct the monitor of the property. Consider, for instance, the IPU of our running example (Sec. 3.2.1.3). It has a property of the "timed implication constraint" type stating that "*if face recognition starts properly with the defined image address, the IPU reads the analyzed image and images from the external gallery, and then sends an interrupt (the positive edge of the interrupt followed by the negative edge) within* 500 *nanoseconds*." (see T1-IPU, Sec. 3.4.2). This is formally defined by Formula 6.1.

$$\Big(\textit{set-img-addr} < \textit{start} \Longrightarrow \big(\{\textit{read-img}^{[100,19000]}, \textit{read-gl-img}^{[10000,2000000]}\}, \wedge, \textit{Shuffled}\big) \\ < \textit{set-irq-pos} < \textit{set-irq-neg} \mid 500ns\Big) \tag{6.1}$$

Figure 6.9 illustrates the construction of a monitor of the loose-ordering property defined by Formula 6.1. Here, for instance, lines 21 – 26 show the construction of a monitor of the fragment:

$$\big(\{\textit{read-img}^{[100,19000]}, \textit{read-gl-img}^{[10000,2000000]}\}, \wedge, \textit{Shuffled}\big).$$

The monitors of the ranges are objects of the class `RangeErrorShuff`, since the fragment appears in the right loose-ordering of the property and it has shuffled semantics ($\sqcup\!\sqcup = \textit{Shuffled}$). When the monitor of the fragment is instantiated (line 26), the Boolean parameters defining semantics of the fragment are specified.

We have implemented a tool which enables the automatic generation of monitors from loose-ordering properties. Our tool gets a definition of a property and, if the property is syntactically correct, returns a monitor.

6.2.2.4 Computation of Recognition Context

To compute the recognition context of primitive monitors of ranges, one needs to call the `initContext(Context<T>*)` method of the class `AntecedentRequirement` (resp. `TimedImplication-Constraint`). The invocation of the method is propagated downward in the tree of the monitors. At each level, the context is re-evaluated w.r.t. the semantic rules of the attribute grammar.

```
/*consturction of a monitor of the left loose-ordering p
/*the first fragment*/
Range<int>* r1p=new RangeResetNonShuff<int>(SET_IMG_ADDR,1,1);
RangesSet<int>* s1p=new RangesSet<int>(r1p, NULL);
//Boolean parameters of f1p: conjunctive, non-shuffled, first, not last
Fragment<int>* f1p=new Fragment<int>(s1p, true, false, true, false);

/*the second fragment*/
Range<int>* r2p=new RangeStopNonShuff<int>(START,1,1);
RangesSet<int>* s2p=new RangesSet<int>(r2p, NULL);
//Boolean parameters of f2p: conjunctive, non-shuffled, not first, last
Fragment<int>* f2p=new Fragment<int>(s2p, true, false, false, true);

LooseOrdering<int>* p2=new LooseOrdering<int>(f2p, NULL);
f2->setParent(p2);
LooseOrdering<int>* p=new LooseOrdering<int> (f1p, p2);
f1->setParent(p);

/*construction of a monitor of the right loose-ordering*/
/*the first fragment*/
Range<int>* r1q=new RangeErrorShuff<int>(READ_IMG, 100, 19000);
Range<int>* r2q=new RangeErrorShuff<int>(READ_GL_IMG, 1000,2000000);
RangesSet<int>* s2q=new RangesSet<int>(r2q, NULL);
RangesSet<int>* s1q=new RangesSet<int>(r1q, s2q);
//Boolean parameters of f1q: conjunctive, shuffled, first, not last
Fragment<int>* f1q=new Fragment<int>(s1q, true, true, true, false);

/*the second fragment*/
Range<int> r3q=new RangeErrorNonShuff<int>(SET_IRQ_POS, 1,1);
RangesSet<int>* s3q=new RangesSet<int>(r3q, NULL);
//Boolean parameters of f2q: conjunctive, non-shuffled, first, not last
Fragment<int>* f2q=new Fragment<int>(s3q, true, false, true, false);

/*the third fragment*/
Range<int> r4q=new RangeErrorNonShuff<int>(SET_IRQ_NEG, 1,1);
RangesSet<int>* s4q=new RangesSet<int>(r4q, NULL);
//Boolean parameters of f3q: conjunctive, non-shuffled, not first, last
Fragment<int>* f3q=new Fragment<int>(s4q, true, false, false, true);

LooseOrderingOutputs<int>* q3=new LooseOrderingOutputs<int>(f3q, NULL);
LooseOrderingOutputs<int>* q2=new LooseOrderingOutputs<int>(f2q, q3);
LooseOrderingOutputs<ing>* q=new LooseOrderingOutputs<ing>(f1q, q2);

TimedImplicationConstraint<int>* t=
                        new TimedImplicationConstraint<int>(p, q, 500);
```

Figure 6.9 Construction of a monitor of the loose-ordering property defined by Formula 6.1.

6.2.2.5 Well-Formedness Checking

When a monitor of a property is constructed, the *well-formedness of the property* can be checked by calling the function `isWellFormed()`. The method implements conditions of the attribute grammar on the vocabularies of different parts of the property.

The call of the method also ensures the *structural well-formedness of a monitor*. A monitor of a fragment is well-formed, if it is composed from the primitive monitors of ranges of the type corresponding to the semantics and the position of the fragment (see Sec. 5.4.2.1). For instance, a monitor of a fragment with shuffled semantics

$$\left(\{\mathit{read\text{-}img}^{[100,19000]}, \mathit{read\text{-}gl\text{-}img}^{[10000,2000000]}\}, \wedge, \mathit{Shuffled}\right),$$

which appears in the right loose-ordering of the property 6.1 should be constructed from monitors of ranges of the `RangeErrorShuff` type. When monitors are generated with our tool, they are correct by construction.

6.2.2.6 Evaluation of a Property

The evaluation of a property is performed by means of the function `step(name)`. A step of the monitor of the antecedent requirement $\mathcal{A} = \left(\mathcal{P} \ll i \mid \nabla\right)$ is shown in Figure 6.10. It consists of a step of the monitor

```
/* AntecedentRequirement */
// LooseOrdering<T>* P

void step(T name){
  P->step(name);
  if (name == i && !P->isError()
      && context == true){
    P->reset();
    P->start(); }}
```

Figure 6.10 A step of the `AntecedentRequirement<T>`.

```
1  /* TimedImplicationConstraint */
2  // LooseOrdering<T>* P
3  // LooseOrderingOutputs<T>* Q
4  void step(T name){
5    if (P->isActive()){
6      P->step(name);
7      if (P->isStop()){
8        P->reset();
9        Q->start();
10       Q->step(name);
11   }} else if (Q->isActive()){
12     Q->step(name);
13     if (Q->isStop()){
14       Q->reset();
15       P->start();
16       P->step(name); }}}
```

Figure 6.11 A step of the `TimedImplicationConstraint<T>`.

```
1  /* LooseOrdering */
2  // Fragment<T>* F
3  // LooseOrdering<T>* L
4
5  void step(T name){
6    if (F->isActive()){
7      F->step(name);
8      if (F->isStop()){
9        if (L != NULL){
10         L->start();
11         L->step(name);
12   }}} else if (L != NULL)
13     L->step(name);}
```

Figure 6.12 A step of the `LooseOrdering<T>`.

```
1  /* Fragment */
2  // RangesSet<T>* X
3
4  void step(T name){
5    X->step(name);}
6
7  /* RangesSet */
8  // RangesSet<T>* X
9  // Range<T>* R
10
11 void step(T name){
12   R->step();
13   if (X != NULL) X->step();}
```

Figure 6.13 A step of the `Fragment<T>`.

of a loose-ordering $\mathcal{P}$ (line 5) and, if the semantics of $\mathcal{A}$ is repeated ($\nabla = \mathit{Repeated}$), reset of the monitor of $\mathcal{P}$ (lines 6 – 9).

Figure 6.11 shows a step of the monitor of the timed implication constraint $\mathcal{T} = (\mathcal{P} \implies \mathcal{Q} \mid t)$. It performs a step on an active loose-ordering $\mathcal{P}$ (lines 5, 6) or $\mathcal{Q}$ (lines 11, 12). If the active loose-ordering $\mathcal{P}$ (resp. $\mathcal{Q}$) is stopped, it is reset and the alternating loose-ordering $\mathcal{Q}$ (resp. $\mathcal{P}$) starts and performs a step (lines 7 – 10 and lines 13 – 16 respectively).

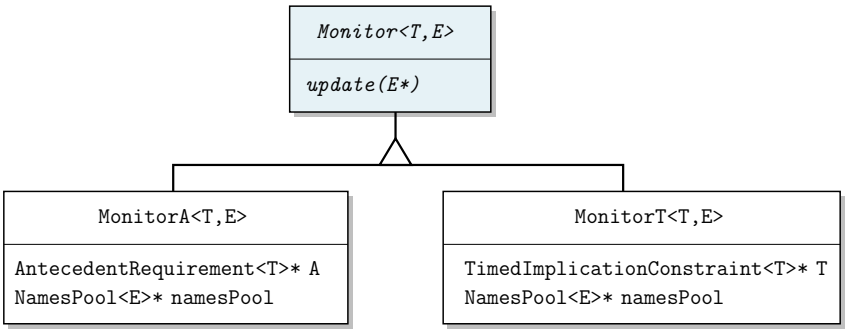
The monitor of the `LooseOrdering` type performs a step on the monitor of a fragment, if it is active (see Fig. 6.12, lines 6, 7). If the monitor of a fragment is *active* and the performed step stops it, the monitor of the following fragment is started through the start of the nested monitor of a loose-ordering (lines 8 – 11). If the monitor of a fragment is *passive*, the step is called on the monitor of the next fragment (though the nested monitor of a loose-ordering).

A step of the monitor of a fragment (see Fig. 6.13) consists of the steps of all monitors of its ranges through the nested monitors of the `RangesSet` type.

The primitive monitors of ranges implement the elementary recognizers defined in Section 5.4.1. When the method `step(name)` is called, those monitors trigger one transition of the corresponding elementary recognizers, updating the `state` and (possibly) increasing the counter `cpt`. To establish which transition should be taken, one defines if the `name` should occur before, immediately after, etc.

6.3 Monitors of Loose-Ordering Properties

The primitive monitors of loose-orderings defined in Section 6.2 are independent from our monitoring model (see Sec. 6.1.2.1). To monitor loose-ordering properties of the antecedent requirement and the timed implication constraint type, we define the monitors shown in Figure 6.14(a); they inherit from the *`Monitor`* abstract class. The class `MonitorA` (resp. `MonitorT`) possesses a pointer to the monitor of the `AntecedentRequirement` (resp. `TimedImplicationConstraint`) type. The monitors also have pools of names represented by the class `NamePool` which are used to map communication events to names of loose-ordering properties. The implementation is completely generic; our approach works whatever type of events `E` and type of names `T` are.



(a) Class diagram

```
1  /* MonitorA<T,E>  */
2  // AntecedentRequirement<T>* A
3  // NamesPool<E>* namesPool
4
5  void update(E* e){
6    T* name = namesPool->getName(e);
7    if (name != NULL){
8      A->step(&name); }
9    if (A->isError()){
10     cout<<"Violation of A\n";
11     exit(1); }}
```

(b) Update of the `MonitorA<T,E>`

```
1  /* MonitorT<T,E>  */
2  //TimedImplicationConstraint<T>* T
3  //NamesPool<E>* namesPool
4
5  void update(E* e){
6    T* name = namesPool->getName(e);
7    if (name != NULL){
8      T->step(&name);
9    } if (T->isError()){
10     cout<<"Violation of T\n";
11     exit(1); }}
```

(c) Update of the `MonitorT<T,E>`

Figure 6.14 The monitors of loose-ordering properties.

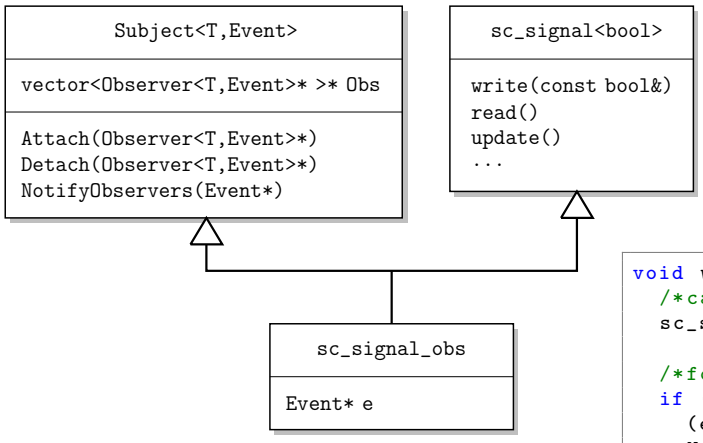
Figures 6.14(b) and 6.14(c) show the implementation of the `update(E* e)` method of the `MonitorA` and the `MonitorT` respectively. The implementation consists of (i) identifying the name of a property based on a descriptor of event `e` (line 11 in both figures), (ii) making a step of the monitor of the antecedent requirement (resp. the timed implication constraint), if the name is in the pool (lines 7, 8 in both figures), (iii) checking if after the performed step the monitor of the property entered the error state (lines 9, 11 in both figures).

6.4 Monitorable SystemC/TLM Channels

We are interested in monitoring loose-ordering properties occurring on the boarder (interface) of TLM components. Thus, the entities of a virtual prototype, where the (potentially) monitored events can occur, are those enabling inter-component communications. Specifically, they are: (i) SystemC signal channels modeling signal interrupts, (ii) TLM initiator/target ports which implement the communication by means of transactions. To enable run-time monitoring of the TL model's behavior, we define the observable version for each of the listed entities in the sense that any communication event (e.g., sending a transaction) is visible to the monitors of loose-orderings. The observable SystemC signal channels and TLM ports inherit the class `Subject<T,E>` (Sec. 6.1.2.1), thus they may have the set of monitors attached.

6.4.1 Observable Signal Channels

We define our own channel class `sc_signal_obs` as it is shown in Figure 6.15. The class inherits from the provided SystemC `sc_signal<bool>` channel and from the `Subject<T,Event>`. It possesses a descriptor of an event `Event* e` which may occur. An event occurring in a considered channel is defined as the change of the channel's value. The value changes any time the method `write()` of the `sc_signal<bool>` is called. To capture the change of the value, the method is redefined such that a value of the descriptor `e` is updated, and all registered observers of the `sc_signal_obs` are notified and provided with the event descriptor `e` (see Fig. 6.15(b)).



(a) The class `sc_signal_obs`

```
void write( const bool& value_){
  /*call write() of the base class*/
  sc_signal<bool>::write(value_);

  /*form a descriptor of event*/
  if (value_){
    (e->_irq_event).op = POS;
    NotifyObservers(e);
  } else {
    if (!init_flag){
      (e->_irq_event).op = NEG;
      NotifyObservers(e); }
    init_flag = 0;  }}
```

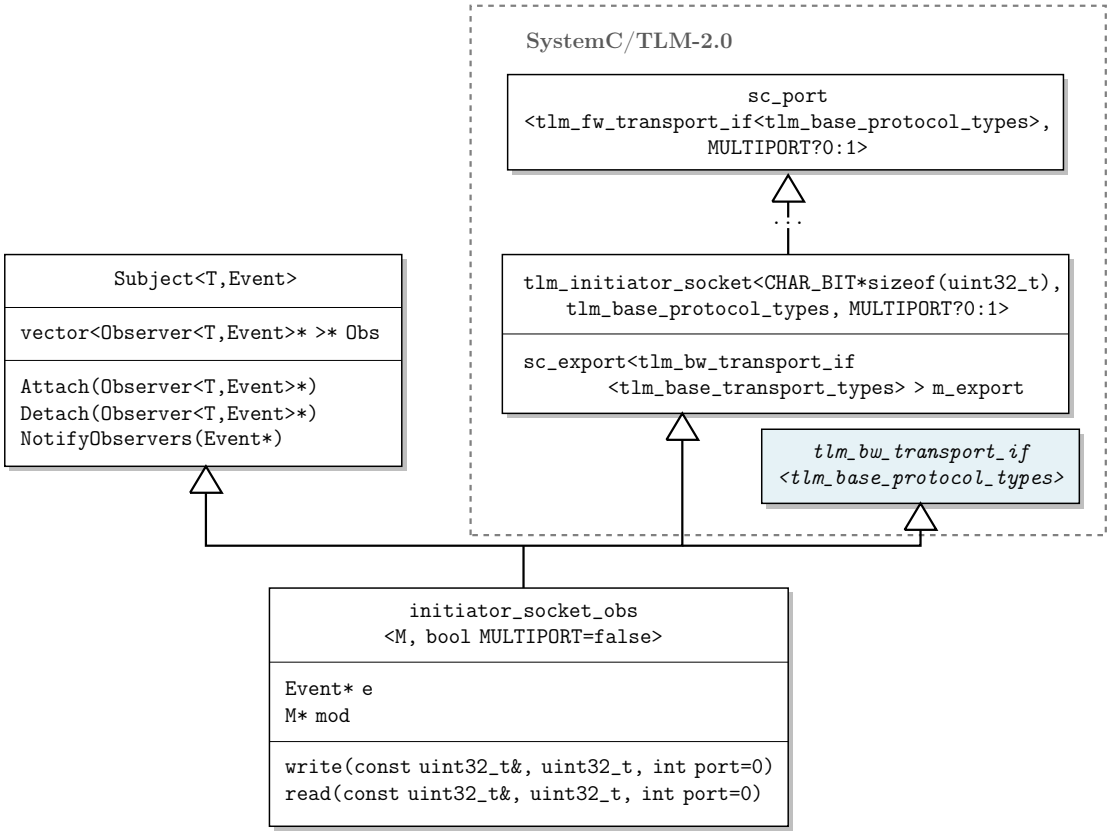
(b) Redefinition of `write()`

Figure 6.15 Observable SystemC channel.

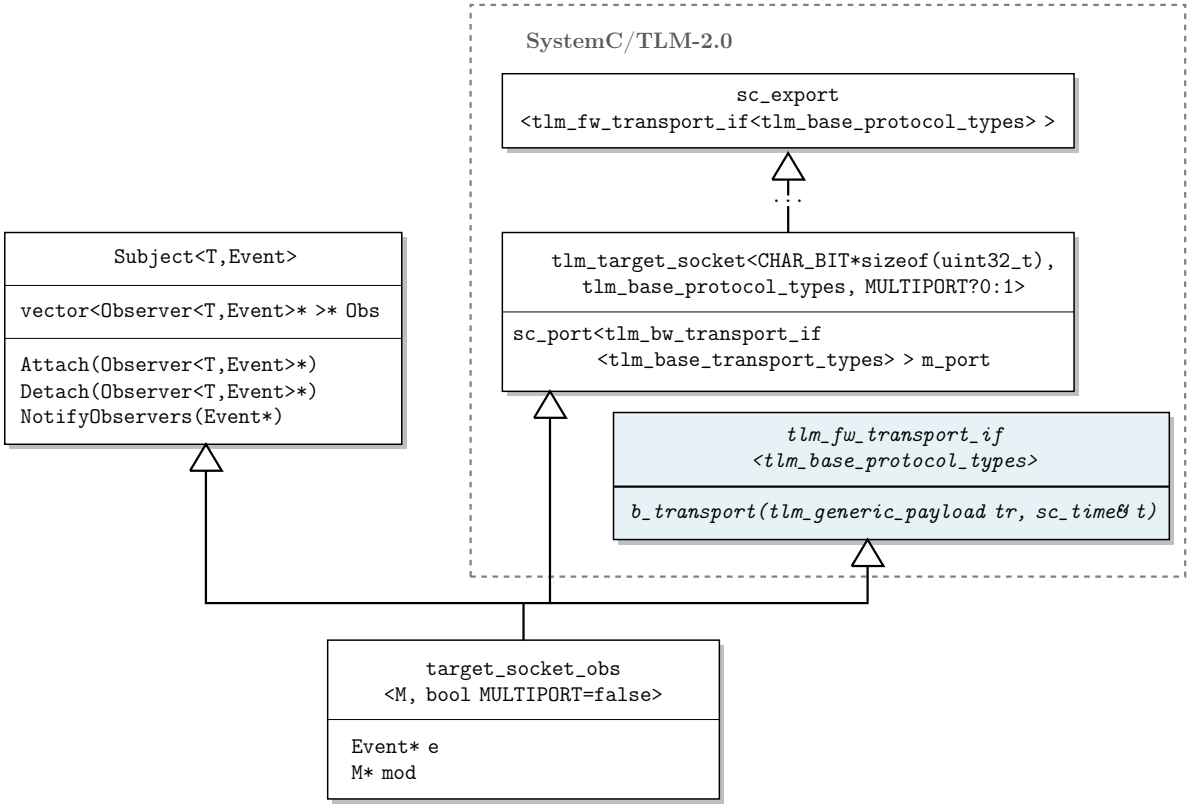
6.4.2 Observable SystemC/TLM Sockets

We define observable TLM sockets `initiator_socket_obs<M, bool MULTISOCKET=false>` (Fig. 6.16(a)) and `target_socket_obs<M, bool MULTISOCKET=false>` (Fig. 6.16(b)). The parameter `M` of the templates is the type of the component possessing the initiator (resp. target) socket. Provided that the implementation of the TLM model is in SystemC, `M` is always a SystemC module `sc_module`. The second parameter `MULTISOCKET` is used to specify whether the initiator (resp. target) socket can be connected to several target (resp. initiator) sockets[2]. By default the point-to-point connectivity is set, i.e., one socket can be connected to only one other socket. The defined classes have pointers to a parent module and a descriptor of events occurring in observable sockets.

[2]Strictly speaking the `MULTISOCKET` parameter determines if the respective SystemC socket (which is the base class of the TLM socket) is a *multisocket*. A multisocket can be connected to more than one SystemC channel (see Sec. 2.1.5).



(a) Observable initiator socket



(b) Observable target socket

Figure 6.16 Observable SystemC/TLM sockets.

```
/* initiator_socket_obs */
// using namespace tlm, sc_core
// Event* e
// sc_time time

tlm_response_status read(const uint32_t& addr, uint32_t& data, int port=0){
  tlm_generic_payload tr;

  /* form transaction */
  tr.set_command(TLM_READ_COMMAND);
  tr.set_address(addr);
  ...
  /* send transaction */
  (*this)[port]->b_transport(tr, time);

  /* form a descriptor of event */
  (e->_reg_event).addr = addr;
  (e->_reg_event).op = READ;
  (e->_reg_event).val=*(reinterpret_cast<uint32_t*>(trans.get_data_ptr()));

  /* notify monitors */
  NotifyObservers(e);
  return tr.get_response_status(); }
```

Figure 6.17 The `read` method of the class `initiator_socket_obs<sc_module>`.

6.4.2.1 Observable Initiator Socket

The observable initiator socket is derived from the TLM socket `tlm_initiator_socket<...>` and inherits the combined TLM transport interface *`tlm_bw_transport_if<...>`* (Fig. 6.16(a)). It provides two methods `write(const uint32_t&, uint32_t, int port=0)` and `read(const uint32_t&, uint32_t, int port=0)` (e.g., see Fig. 6.17) to forward respectively write and read transactions. These methods can be accessed by the parent module of the socket for communication purposes. The transactions are formed based on the arguments of the methods. The first (resp. the second) argument defines the address (resp. the data) of the transaction. The argument `port` defines the number of the connected target sockets to which the formed transaction should be sent (it is relevant only if the initiator socket can be connected to several target sockets, i.e., `MULTIPORT=true`).

When either the `write(...)` or `read(...)` method is called by the parent component, i.e., a communication event occurs, a descriptor of the event `e` is formed and all attached observers (monitors) are notified with the call of the method `NotifyObservers(e)`, which is inherited by the observable initiator socket from the `Subject<T, Event>`.

6.4.2.2 Observable Target Socket

The base class of the observable target socket is the `tlm_target_socket<...>` defined as a part of TLM-2.0. The observable target socket implements the TLM combined interface *`tlm_fw_transport_if<...>`* (Fig. 6.16(b)), specifically its virtual method *`b_transport(...)`* enabling blocking communication[3]. The implementation of this interface ensures that the previously described initiator socket can be connected to the defined target socket.

The invocation of the method `b_transport(...)` corresponds to occurrence of an input event of the parent component of the observable target socket. When `b_transport(tlm_generic_payload tr, sc_time& t)` is called, a descriptor of the occurring event is formed, the transaction `tr` is executed on the parent component of the socket, and the monitors attached to the target socket are notified provided with the defined event descriptor. The described implementation of the method is in Figure 6.18.

Summary

In this chapter, we have presented the direct translation of loose-ordering properties into SystemC monitors. We have enumerated monitoring principles of SystemC/TLM and have presented our model for monitoring the loose-ordering properties. We have defined the library of primitive SystemC monitors and

[3]Recall, the blocking transport interfaces of TLM-2.0 are provided for transactions which can be finished within a single functional call.

```
/* target_socket_obs */
// using namespace tlm, sc_core
// Event* e
// sc_time time
// sc_module* mod

void b_transport(tlm_generic_payload& tr, sc_time& t){
  (void) t;
  uint32_t addr = static_cast<uint32_t>(tr.get_address());
  uint32_t& data = *(reinterpret_cast<uint32_t*>(tr.get_data_ptr()));

  /*form descriptor of event, execute transaction*/
  (e->_reg_event).addr = addr;
  (e->_reg_event).val = data;
  switch(tr.get_command()) {
  case TLM_READ_COMMAND:
    tr.set_response_status(mod->read(addr, data));
    (e->_reg_event).op = READ;
    break;
  case TLM_WRITE_COMMAND:
    tr.set_response_status(mod->write(addr, data));
    (e->_reg_event).op = WRITE;
    break;
  case TLM_IGNORE_COMMAND:
    break;
  default:
    tr.set_response_status(TLM_COMMAND_ERROR_RESPONSE);
  }

  /*notify monitors*/
  NotifyObservers(e); }
```

Figure 6.18 The `target_socket_obs<sc_module>` class definition: the `b_transport()` method.

have shown how they could be used for construction of monitors of arbitrary properties. Our library is based on the attribute grammar of the loose-ordering language. Finally, we have presented the observable SystemC/TLM communication channels which can notify monitors about communication events.

In the next chapter, we compare the complexity of our direct monitors with the complexities of monitors available for PSL.

Chapter 7

Experiments

Contents

7.1 Comparing Complexities . . . 125
7.1.1 Experimental settings . . . 125
7.1.2 The ViaPSL Strategy: Complexities of the PSL Monitors . . . 126
7.1.3 The Drct Strategy: Complexities of the Direct SystemC Monitors . . . 126
7.1.4 Comparison . . . 127
7.2 Monitoring Loose-Ordering Properties . . . 128

The experiments presented in this chapter consist of two parts. The first part is the comparison of the time and memory complexities of the direct SystemC monitors of the loose-ordering properties with the respective complexities of the PSL monitors defined in [PF08; FP10]. To perform the comparison, we use the encoding of the loose-ordering properties into PSL as proposed in Chapter 4. The purpose of the second part of the experiments is to show that the synchronization bugs of the intercom system (see Chapter 3) can be detected with the direct SystemC monitors. We show that the monitors help in finding the sources of those bugs.

The chapter is organized as follows: In Section 7.1, we compute and compare the complexities of the monitors. In Section 7.2, we build the SystemC monitors for the loose-ordering properties of the intercom system, and show how the monitors detect the synchronization bugs. In Section 7.2 we also show how one can find the cause of the bugs based on the information provided by the monitors.

7.1 Comparing Complexities

We compare the time and memory complexities of the monitors for the loose-ordering properties. The *time complexity* is the total number of operations in an imperative language used to implement a monitor. The *memory complexity* is a monitor's memory dump (i.e., the number of bits).

7.1.1 Experimental settings

We consider different configurations of the loose-ordering properties. For each configuration P we build the direct SystemC monitor and the PSL monitor. We consider two strategies to obtain the monitors of P (see Fig. 7.1):

(i) Drct is the direct translation of P into SystemC as defined in Chapter 6;
(ii) ViaPSL first translates P into PSL (see Chapter 4, Sec. 4.5.1), then the built PSL encoding is translated into a SystemC monitor as described in [PF08].

For a fair comparison, we define the configurations of the loose-ordering properties only in the subset of the loose-ordering language encoded into PSL (Fig. 7.1). Recall that the encoded subset defines (i) the loose-ordering properties with only *non-shuffled* fragments, and (ii) the *strict alternation* of the loose-ordering $\mathcal{P}$ and a name i (resp. a loose-ordering $\mathcal{Q}$) of an antecedent requirement $\mathcal{A} = \big(\mathcal{P} \ll i \mid \nabla\big)$ (resp. a timed implication constraint $\mathcal{T} = \big(\mathcal{P} \Longrightarrow \mathcal{Q} \mid t\big)$).

In this chapter, we provide detailed results only for the encoding of the loose-orderings into PSL by means of the LTL operators. We do not analyze the Sequential Extended Regular Expressions (SERE)

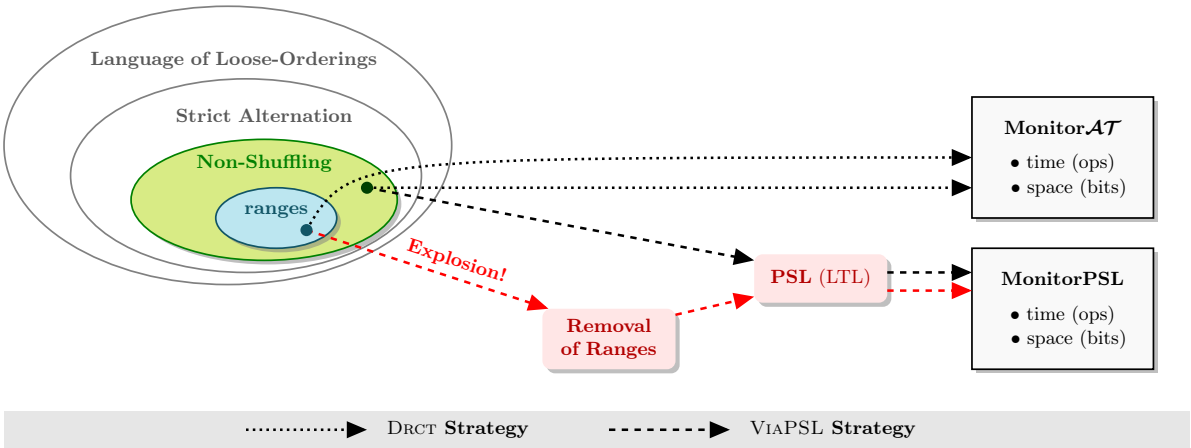


Figure 7.1 Experimental settings.

encoding because we do not have the actual implementation of the respective PSL monitors. In [BA14], it is mentioned that the complexity of the SERE monitors is proportional to the length (i.e., the number of SERE operators) of the respective SERE formulas. Provided that our encoding of the loose-ordering properties into SERE leads to the combinatorial explosion of the length of the SERE formulas (see Chapter 4, Sec. 4.5.2.3), we can assume that the time and memory complexities of the SERE monitors are also exponential.

In Section 7.1.2 (resp. Section 7.1.3) below, we compute the complexities of the direct SystemC monitors (resp. the PSL monitors). In Section 7.1.4, we present the obtained results and compare the computed complexities.

7.1.2 The VIAPSL Strategy: Complexities of the PSL Monitors

According to [PF08] the time and memory complexities of the monitors generated with the VIAPSL strategy are linear with regard to the length of the formula. As stated in [PF08; FP10], the PSL monitor of a loose-ordering property P is built by interconnecting the primitive monitors. These primitive monitors are associated with the PSL operators of P. To compute the time (resp. memory) complexity of the monitor of P, we need:

(i) to compute the number of operations Ops (resp. the number of bits $Bits$) per primitive PSL monitor,
(ii) to compute the number of primitive monitors N constituting the monitor of P (this is equal to the number of the PSL operators in P),
(iii) finally, to multiply Ops (resp. $Bits$) by N.

The Number of Operations and Bits of a Primitive Monitor Figure 7.2 shows the examples of the primitive PSL monitors from [PF08; Pie07]. Each primitive monitor performs approximately 13 operations. Each primitive monitor has approximately 8 Boolean variables which makes up to 64 bits. Thus, $Ops = 13$, $Bits = 64$.

The Number of Primitive Monitors The PSL monitor of a loose-ordering property P is built from N primitive monitors. N is equal to the number of PSL operators in P. If P is of the antecedent (resp. timed implication constraint) kind, N is defined by Sum 4.30 (resp. Sum 4.31) derived in Chapter 4 (see Sec. 4.5.3.3).

7.1.3 The DRCT Strategy: Complexities of the Direct SystemC Monitors

We compute the time (resp. memory) complexity of the direct SystemC monitor as follows: First, we compute the number of operations (resp. bits) of the elementary monitors of ranges, the composed monitors of fragments and loose-orderings, and the monitors of the antecedent requirement and the timed implication constraint (see Table 7.1). Then, for a loose-property P, we compute the number of monitors of each kind in the derivation tree of P. Finally, to find the time (resp. memory) complexity of P, we multiply the number of monitors of each kind by the respective number of operations (resp. bits), and sum

```
class mnt_always: public Monitor {
protected:
  bool reset_n, start, expr,
    checking, valid,
    start_always, start_t1,
    valid_t;
public:
 mnt_always(const char* n):
                  Monitor(n){
    valit_t=true;
  }
  void update(){
    //part 1
    start_always=start||start_t1;
    //part 2
    if((reset_n==false)||
       (start_always==false))
      valid_t=true;
    else
      if(expr==true)
        valid_t=true;
      else
        valid_t=false;
    //part 3
    if(reset_n==false)
      start_t1=false
    else if (start==true)
      start_t1=true;
    //part 4
    valid=valid_t;
    //part 5
    checking=start_always;
}};
```

(a) The PSL monitor of the "`always`" operator. (Source: [PF08])

```
class mnt_impl:public Monitor{
protected:
  bool reset_n, start, expr,
    checking, valid,
    start_impl, cond,
    valid_t;
public :
  mnt_impl(const char *n):
                Monitor(n){
    valid_t = true;
  }
  void update(){
    //part 1
    start_impl=start && cond;
    //part 2
  if ((reset_n==false)||
      (start_impl==false))
    valid_t = true;
  else
    if (expr == true)
      valid_t = true;
    else
      valid_t = false;




  //part 4
  valid = valid_t;
  //part 5
  checking = start_impl;
}};
```

(b) The PSL monitor of the "`->`" operator. (Source: [Pie07])

Figure 7.2 Primitive monitors for PSL operators.

Type of a monitor	Time (ops)	Space (bits)
`RangeErrorNonShuff`, `RangeResetNonShuff`, etc.	≈ 47	≈ 232
`RangeSet`	≈ 13	≈ 64
`Fragment`	≈ 6	≈ 96
`LooseOrdering`, `LooseOrderingOutputs`	≈ 23	≈ 64
`AntecedentRequirement`	≈ 10	≈ 40
`TimedImplicationConstraint`	≈ 23	≈ 64

Table 7.1 The time and memory complexities of the SystemC monitors.

up the results. To compute the complexities, we assume that the SystemC monitors, which are template classes (see Chapter 6), are instantiated for integer names of P.

7.1.4 Comparison

Table 7.2 lists the configurations of the loose-ordering properties we consider and the respective time and memory complexities of the monitors obtained with the DRCT and VIAPSL strategies. The summand Δ in the table stands for the complexity of the lexical analyzer which is supposed to be used together with the PSL monitors in order to remove ranges (see Chapter 4, Sec. 4.5.3.2). The provided results in Table 7.2 show that the time and memory complexities of the monitors of both strategies are linear with regard to the length of a loose-ordering property P. The complexities of the DRCT monitors do not

Configurations	Drct		ViaPSL(LTL)	
	time (ops)	space (bits)	time (ops)	space (bits)
$(n \ll i \mid \mathit{Repeated})$	≈ 99	≈ 496	$\approx 182 + \Delta$	$\approx 896 + \Delta$
$(n^{[100,19000]} \ll i \mid \mathit{Repeated})$	≈ 99	≈ 496	$\approx 28 \times 10^9 + \Delta$	$\approx 137 \times 10^9 + \Delta$
$\big((\{n_1, \ldots, n_4\}, \wedge) \ll i \mid \mathit{Non\text{-}Repeated}\big)$	≈ 279	≈ 1384	$\approx 676 + \Delta$	$\approx 3328 + \Delta$
$\big((\{n_1, \ldots, n_5\}, \wedge) \ll i \mid \mathit{Non\text{-}Repeated}\big)$	≈ 339	≈ 1680	$\approx 780 + \Delta$	$\approx 3840 + \Delta$
$\big((\{n_1, \ldots, n_{10}\}, \wedge) \ll i \mid \mathit{Non\text{-}Repeated}\big)$	≈ 639	≈ 3160	$\approx 1300 + \Delta$	$\approx 6400 + \Delta$
$\big((\{n_1, \ldots, n_{20}\}, \wedge) \ll i \mid \mathit{Non\text{-}Repeated}\big)$	≈ 1239	≈ 6120	$\approx 2340 + \Delta$	$\approx 11520 + \Delta$
$(n_1 \Rightarrow n_2 < n_3 < n_4 \mid t)$	≈ 379	≈ 1888	$\approx 936 + \Delta$	$\approx 4608 + \Delta$
$(n_1 \Rightarrow n_2 < \cdots < n_{10} \mid t)$	≈ 913	≈ 4624	$\approx 3510 + \Delta$	$\approx 17280 + \Delta$
$(n_1 \Rightarrow n_2 < \cdots < n_{20} \mid t)$	≈ 1803	≈ 9184	$\approx 9880 + \Delta$	$\approx 48640 + \Delta$
$(n_1 \Rightarrow n_2^{[100,19000]} < \cdots < n_{20} \mid t)$	≈ 1803	≈ 9184	$\approx 28 \times 10^9 + \Delta$	$\approx 137 \times 10^9 + \Delta$

Table 7.2 The comparison of the Drct and ViaPSL strategies. All fragments are non-shuffled.

depend on ranges. The time and memory complexities of the ViaPSL monitors explode when P contains ranges.

7.2 Monitoring Loose-Ordering Properties

In this section, we build the SystemC monitors for the loose-ordering properties of our running example provided in Section 3.4. The monitors are built as described in Chapter 6 (see Sec. 6.2.2.3). We use the monitors to check that all loose-ordering properties are satisfied. Tables 7.3 and 7.4 present the results. The first and the second columns of the tables are respectively the reference names and descriptions of the bugs. The third column provides the reference name(s) of the loose-ordering(s) property(ies) which is(are) violated. If a property is violated (e.g., A4-CPU), the respective monitor reports an error. The fourth column lists the causes of the bugs. We establish the cause of each bug based on the information provided by the monitors detecting the bugs. Some bugs were detected and localized by means of the monitors. Some bugs were injected in order to ensure that the monitors detect the synchronization malfunctions of the system.

Ref.	Description	Detection	Cause of the Bugs
BS1	The Button-Start does not respond whatever is the execution phase of the system.	A1-TMR2	The timer TMR2 (used by the GPIO) is started without the time scale being defined.
		T6-CPU	The CPU does not start the GPIO.
BS2	The display is always black.	T1-SEN	The SEN does not send an interrupt when it captures an image.
		T6-CPU	The CPU does not start the LCDC.
		T7-CPU T9-CPU	The CPU does not update the content of the LCDC's buffer.
BS3	The display shows nonsense.	T7-CPU T8-CPU	The SEN's buffer is read while the component is capturing an image.

Table 7.3 The detection and localization of the synchronization bugs which could appear in the TL model of the intercom system. Part I.

Ref.	Description	Detection	Cause of the Bugs
BS4	The system gets stuck at the face recognition: the BUTTON-START does not respond, the same (currently analyzed) image is shown on the display.	T1-IPU	The IPU does not send an interrupt when the face recognition terminates.
		T4-CPU	The CPU did not acknowledge a previously received interrupt from the GPIO.
BS5	While the system performs the face recognition, the display starts to show the images being captured by the sensor SEN at that moment of the system's execution.	T4-CPU	The CPU did not disable interrupts of the SEN before starting the face recognition.
BS6	For any user the system always returns the "access-denied" image.	A2-IPU	The face recognition is started and the size of the image is not defined.
		A2-IPU	The face recognition is started and the size of the image gallery is not defined.
BS7	The registered user gets the "access-denied" notification	A4-IPU	The confidence value of the IPU is read before the respective face recognition terminates.
		A3-IPU	The image address is not specified before the respective face recognition is started, and the old one is used.
BS8	The registered user sees on the screen a salutation image addressed to another user.	A5-IPU	The address of the reference image computed by the IPU is read before the respective face recognition terminates.
BS9	The unregistered user sees on the screen a salutation image addressed to someone else, and gets access into the building.	A4-IPU	The confidence value of the IPU is read before the respective face recognition terminates.
		A5-IPU	The image address is not specified before the respective face recognition is started, and the old one is used
BS10	Face recognition starts when the system shows either the "access-denied" or salutation image.	T4-CPU	The CPU did not disable interrupts of the GPIO before starting the face recognition.
BS11	After the first termination of the face recognition, the system's execution gets stuck: the display shows either salutation or "access-denied" image, the BUTTON-START does not respond.	A1-TMR1	The timer TMR1 (used by the CPU) is started without being configured with the time scale.
		T3-CPU	The CPU did not acknowledge a previously received interrupt from the IPU.
		T10-CPU	The CPU did not activate the shutter of the SEN when the time T elapsed.
BS12	The user does not see any notification image on the screen when the face recognition is finished.	T9-CPU	The CPU did not start the timer and immediately proceeded to the activation of the SEN's shutter.
BS13	The Bus fails to establish the target component for a transaction initiated by the CPU.	A5-IPU	The reference image of the IPU is read before it is computed at least once (i.e., before at least one face recognition terminates).
BS14	The Bus cannot establish the target component for a transaction initiated by the IPU.	A2-IPU A3-IPU	The face recognition is started without the image address being defined.
		A2-IPU	The face recognition is started without the address of the image gallery being defined.
BS15	The Bus cannot establish the target component for a transaction initiated by the LCDC.	A2-LCDC	The LCDC is started with an undefined address of its buffer.

Table 7.4 The detection and localization of the synchronization bugs which could appear in the TL model of the intercom system. Part II.

Summary

In this chapter, we have compared the complexities of the direct SystemC monitors for the loose-ordering properties and the PSL monitors. The obtained results show that the direct SystemC monitors are more efficient than the PSL monitors. We also have shown that the monitors of the loose-ordering properties can capture the synchronization bugs of the SystemC/TLM virtual prototype.

Part IV

Towards Generalized Stubbing with Contracts

Chapter 8

Towards Generalized Stubbing with Sequence Properties

Contents

8.1 Introducing the Notion of Generalized Stubbing . . . 133
8.1.1 Inspiration . . . 134
8.1.2 Generalized Stubbing for SoCs . . . 135
8.1.3 Constraints on Property Languages and Semantic Choices . . . 135
8.1.4 Expected Benefits of Generalized Stubbing and Contributions . . . 136
8.2 Runs of Components, Prefixes, Fragments . . . 136
8.3 Expressivity of Contracts vs Implementability of Stubs . . . 137
8.3.1 Semantics of the Implication Operator . . . 138
8.3.2 Efficient Implementation . . . 138
8.3.3 Productive Runs and a Necessary Condition for Bounds . . . 138
8.4 Implementation of an Assume Clause . . . 138
8.5 Restrictions on Property Languages of a Guarantee Clause and Semantic Choices . . . 139
8.5.1 Semantic Choices . . . 139
8.5.2 Continuous Recognition of π . . . 140
8.5.3 Defining Several Guarantee Clauses . . . 140
8.5.4 Bounds of a Cycle-Free Set of Guarantee Clauses . . . 141
8.5.5 Summary on Constraints . . . 142
8.6 Semantics of a System of Components . . . 142

In the first part of this work, we have presented the specification language based on loose-ordering properties used in contract specifications. Now we would like to move further: we generalize loose-orderings as much as possible to define an approach of generalized stubbing for systems-on-a-chip. The idea is that any component can be a stub, given as a contract. Contracts are formally defined by sequence properties, but not any specification language can be used here. We identify a set of constraints on languages defining sequences to be used for contracts. We search for a compromise between expressivity and implementability in a simulation framework.

In the second part of the document, we define the semantics of a contract and propose an execution mechanics which allows early simulation with stubs. This chapter discusses the compromises on property languages one should keep in mind when defining contracts. In the following Chapter 9, we introduce implementation by encoding contracts into input/output state machines; the encoding can be seen as the operation semantics of contracts. In Chapter 10, we show the implementation of the execution mechanics in SystemC, and the experiments with our running examples; this serves a proof of concept for stubbing.

8.1 Introducing the Notion of Generalized Stubbing

The idea of stubbing is not new. Stubs are popular in agile methodologies for software development [And03; Rub12; HF10] where they define a *mock implementation* of a design entity (e.g., a class, a function). In

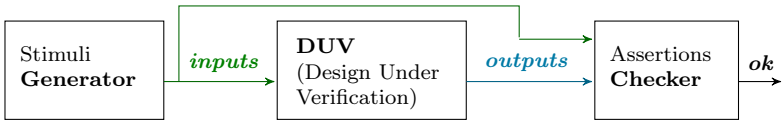


Figure 8.1 The verification framework for hardware designs.

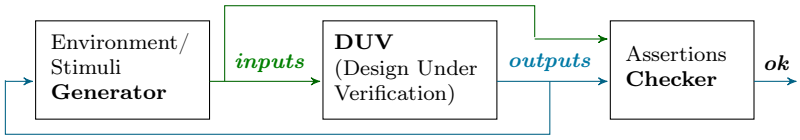


Figure 8.2 The testing framework for reactive systems.

hardware design stubs can play the role of *traffic generators* which produce stimuli for a hardware design (Sec. 2.1.3), or replace a missing component.

8.1.1 Inspiration

Our stubbing method is inspired by three concepts: (i) testing hardware designs, (ii) testing reactive systems, (iii) "design-by-contract" principle.

8.1.1.1 Testing Hardware Designs

The general method for **testing a hardware design** as presented in the Universal Verification Methodology (UVM) [AO14b], SystemC/SCV [Opeb], eRM [IJ04; Erm], SVM [Oli+12], etc. is illustrated in Figure 8.1. A *stimuli generator* is in charge of producing inputs for the hardware design under verification (DUV); an *assertion checker* is in charge of deciding whether the test passes. The *input stimuli* may be described by means of dedicated languages which support random variables and ranges (SystemC/SCV [Opeb], CRAVE [Hae+12; LD14]). Generating the stimuli from such a constraint-based description involves some sort of constraint solving, for Boolean and numerical constraints [LD14]. *Assertions* can be expressed with languages providing temporal logic constructs (e.g., PSL [CVK04]). Checking of assertions can be performed either at simulation time, or offline on a set of simulation traces.

8.1.1.2 Testing Reactive Systems

Figure 8.2 illustrates the **testing approach for reactive systems**, in which the *loop* between the system and its environment has to be taken into account. This is usually the case if the system under test is the implementation of some control law: the outputs of the system at some point in time influence (through the behavior of the *environment*) its future inputs. Generating the stimuli without taking this loop into account is likely to produce unrealistic inputs. This is why the method includes an abstract model of the environment, usually in the form of constraints, as in [Ray+98; JRB06].

The testing method is the following: the black box DUV and the environment play in turn. To avoid cyclic dependencies, the specification of the environment is such that the input to the system (the output of the environment) may only depend on the *previous* output of the system (the input of the environment). When the system produces an output vector o, it is transmitted to the environment specification, which produces the *next* input vector i for the system. The environment being modeled as a set of contraints C between the inputs and outputs, this means: (i) performing a partial evaluation of C with the known value o, which yields a simpler constraint C'; (ii) solving C' to produce a value for i. At any step, the inputs and outputs are given to the assertion checker, which plays the role of the test oracle. There are companies (e.g., Argosim [Arg]) which implement this method in their tools.

8.1.1.3 "Design-by-Contract" Principle

The **"design-by-contract" principle** [Mey92] has been successfully applied to object-oriented programming. Examples include Eiffel [Swi93], iContracts [Kra98], JASS [Bar+01], etc. JASS is particularly interesting because it proposed to write assume clauses as temporal logic properties: for instance, it is possible to express that an input parameter of a method should increase (considering the sequence of calls). The assume/guarantee principles have been formalized for various types of software or hardware systems,

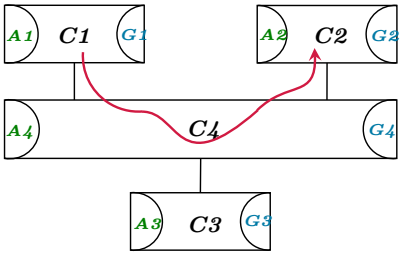


Figure 8.3 Executing a system made of stubs. In this example *C4* is a bus, a communication between *C1* and *C2* through the bus involves *G1*, *A4*, *G4* and *A2*.

for instance in [Ben+08] or [Bau+12]. Applications of (early) contracts include hardware optimization by exploiting *(sequential) don't care* sets [Dev91], modular verification [McM99], executable specifications for synchronous languages [MM04], etc.

8.1.2 Generalized Stubbing for SoCs

The idea of our generalized stubbing mechanics is shown in Figure 8.3. We consider a set of components exchanging data and synchronizing with each other; they are organized in a functional architecture. One of the components can be a bus (like $C4$ in the picture). Each of the components is (potentially) a stub for any other component: outputs of stubs-components (e.g., *C1* in Fig. 8.3) are inputs for other components (e.g., *C2*).

We choose to define stubs by *contracts*, i.e., an *assume clause* A, and a *guarantee clause* G (see Sec. 2.5). In our opinion *contracts* are very suitable for stubbing, since they match gracefully the checking-generating frameworks presented in Section 8.1.1. To simulate the system early in the design cycle, when only the contracts are given, but not the detailed implementation of the components (or at least, not all of them), the idea is the following: the guarantee clause G has to be used as a *generator*, while the assume clause A has to be used as a *checker*. Fully implemented components of a system can be seen as stubs, where the actual implementation plays the role of a generator.

The assume clause for a component C can depend on both the inputs and the outputs of C. An example constraint is: *a value computed by C can be read (the read operation is an input of C) only after the termination of the computation has been signaled by an interrupt (this is an output of C).* The guarantee clauses also depend on inputs and outputs. They define obligations of components. An example constraint is: *whenever the computation of the component has been requested (with an input), its termination will be signaled by an interrupt (an output) within some bounded delay.* All these properties specify *sequences* of events, and they are all *safety* (or *bounded liveness*) properties [AS87]. The properties can be defined, for instance, by means of the loose-ordering language (see Chapter 4). The system-environment loop presented in Section 8.1.1.2 is generalized to any number of components.

Following the principles of the TLM level of abstraction we consider *asynchronous* systems made of stubs connected by oriented point-to-point connections. Communication between stubs is enabled by FIFOs. Each stub has a *unique* input FIFO to preserve the order of inputs. Simulation of a system is managed by a *scheduler* which activates stubs. An activated stub makes a "step" either by getting actual input from the FIFO or by producing actual output and adding it to the FIFO of a target stub.

8.1.3 Constraints on Property Languages and Semantic Choices

To be used for stubbing, the language used to define assume (A) and guarantee (G) clauses should meet several requirements. We make a trade-off between *expressivity* of contracts and *implementability* of stubs, and accept to put constraints on property languages such that they can be used for our stubbing framework. Thus, to implement stubs efficiently, we require the languages to have recognizers of a bounded size. We also need to ensure the absence of backtracking when producing sequences in the generation part.

Besides that, we have the *semantic constraint*: we want all stubs of a system to do something (i.e., produce outputs). If one stub only consumes inputs and never produces outputs, our interpretation is that the stub is infinitely slow relatively to other stubs. We want to avoid such situations; we choose to produce only *productive* runs of components (the runs when the components produce outputs). This can be done by means of scheduling. There can be many scheduling policies; each scheduling allows to produce a certain subset of all productive runs of a component (Fig. 8.4). This leads to a coverage problem which

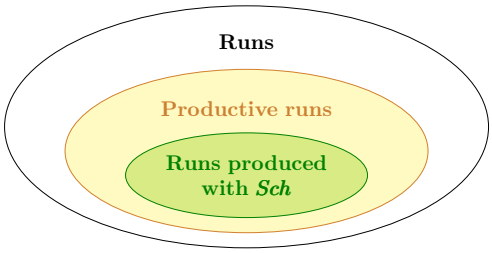


Figure 8.4 Runs of a component. *Sch* is a scheduling policy.

is out of scope of this work. To ensure that all runs are well covered, one could apply, for instance, a method similar to [Hel+06] based on dynamic partial-order reduction techniques [FG05].

We choose to define a guarantee clause G in the implication form $\pi \Rightarrow \sigma$ with some *bound*. The intuitive meaning of $\pi \Rightarrow \sigma$ is: when the left-had part π (a condition) is observed, the right-hand part σ (an obligation) should occur within a bound. The right-hand part σ defines sequences of outputs that are meant to be generated by a stub. Our choice is inspired by the implication operator $\alpha \Mapsto \beta$ of PSL and SVA, where the semantics of the operator is not well defined (see Sec. 2.3.2.4). In this chapter, we provide the clear semantics of $\pi \Rightarrow \sigma$.

8.1.4 Expected Benefits of Generalized Stubbing and Contributions

A structure of stubs with contracts facilitates the detection and localization of bugs at simulation time. Contracts provide a *blaming facility*: if a stub (its assertion checking part) detects an error (e.g., *A2* of *C2* in Fig. 8.3), the component that has produced the corresponding inputs should be blamed (*C1*). Moreover, assertion checkers constituting stubs can later be integrated in traditional testing framework for hardware designs (see Sec. 8.1.1.1). Non-deterministic production of outputs by stubs enables investigation of (potentially) more behaviors of the design than one may observe at simulation with TLM components.

The Contributions Our contributions consist of two linked parts interacting with each other. On the one hand we define generalized stubbing as general as possible. On the other hand we ensure that it is implementable and efficient. We do it by identifying constraints on property languages defining stubs.

The first part of contributions includes: (i) the generalized stubbing mechanics with sequence properties for early simulation with SystemC/TLM virtual prototypes; (ii) the operational semantics of stubs, and of a system made of stubs; (iii) the implementation of the execution mechanics in SystemC.

The second part of contributions includes: (i) the clarification of the semantics of properties of the implication kind $\pi \Rightarrow \sigma$, (ii) the identification of the set of constrains on property languages of $\pi \Rightarrow \sigma$ which enables online exploitation of those properties either for checking or stubbing.

Organization of the Material In Section 8.2, we define productive runs of components. Section 8.3 provides the detailed definition of stubs made of contracts and lists the constraints on property languages. In Section 8.6, we discuss the semantics of a system made of stubs. The operational semantics of stubs and a stub-based system is defined in Chapter 9. The implementation of the execution mechanics for stubs in SystemC and our experiments are discussed in Chapter 10.

8.2 Runs of Components, Prefixes, Fragments

We consider the behavior of components as *sequences* of some interesting elements, one element at a time. We call elements of such sequences *steps*. Consider a component C with an input/output interface defined by a pair (I, O), where I is the set of inputs, and O is the set of outputs (e.g., see Fig. 8.5).

Definition 13: Run — A run of a component C with the input/output interface (I, O) is an *infinite* sequence of inputs of I and outputs of O of a component; only one interface name can occur at a time.

One letter of a run corresponds to one step of C: (i) if a letter is an input in I, the corresponding step of C consists in getting one input from the environment, (ii) if a letter is an output in O, the corresponding

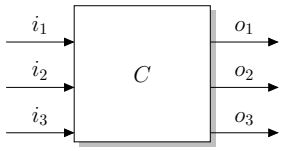


Figure 8.5 A component with inputs $I = \{i_1, i_2, i_3\}$ and outputs $O = \{o_1, o_2, o_3\}$.

step of C consists in producing one output. We need to express the fact that a component has a *productive* behavior, i.e., it actually does something. We do it by introducing the notion of a productive run.

Definition 14: Productive Run — A run $r = r^0 r^1 \dots$ of a component C is *productive* if (i) r contains outputs, (ii) only a bounded number of inputs can occur between any two consecutive outputs of r. Formally:

$$\begin{cases} \exists n \in \mathbb{N} : \ r^n \in O \\ \exists m \in \mathbb{N} : \ \forall i, j \in \mathbb{N} \Big((i \leq j) \wedge (r^i \in O) \wedge (r^j \in O) \ \wedge \ \forall k \in (i, j) : r^k \in I \Big) \rightarrow \Big(j - i \leq m \Big) \end{cases}$$

Definition 15: Projection — The projection of a sequence s on a vocabulary V is a sequence that results by removing all names which are not in V:

$$\rho(s, V) = \begin{cases} \epsilon, & \text{if } s = \epsilon, \\ \rho(u, V), & \text{if } s = \ell u \text{ and } \ell \notin V, \\ \ell \rho(u, V), & \text{if } s = \ell u \text{ and } \ell \in V. \end{cases}$$

Definition 16: Productive Run w.r.t. a Subset of Outputs — A run $r = r^0 r^1 \dots$ of a component C is *productive with respect to a subset* of outputs $O' \subseteq O$ if the projection of r on $I \cup O'$ is a productive run.

For instance, consider a component C shown in Figure 8.5. Runs of the form $i_1 o_1 (i_1 o_2)^* o_3$ are productive; moreover, they are productive runs w.r.t. $\{o_1, o_2, o_3\}$. Each run of the form $i_1 o_1 (i_1)^* o_3$ is not a productive run because the number of times i_1 can occur between o_1 and o_3 is not bounded.

Notice, the definition of a scheduling policy for a stub mentioned in Section 8.1.3 consists in choosing the maximum number of inputs which can occur between any two successive outputs of the stub's runs (for more details see Sec. 8.3).

When discussing property languages we will refer to *prefixes* and *fragments* of runs. They are defined below.

Definition 17: Prefix of a Run — A prefix of a run $r = r^0 r^1 \dots$ is a sequence $r' = r^0 \dots r^k$ for $k \in \mathbb{N}$.

Definition 18: Fragment of a Run — A fragment of a run $r = r^0 r^1 \dots$ is a sequence $r' = r^k \dots r^{k+m}$ for $k, m \in \mathbb{N}$.

8.3 Expressivity of Contracts vs Implementability of Stubs

When we use a contract with an assume clause A and a guarantee clause $\pi \Rightarrow \sigma$ with a bound for our stubbing framework, we can encounter several problems. Checking A is relatively easy: we only require that the property languages used for A have recognizers; this is discussed in more details in Section 8.4. The problems appear when we deal with $\pi \Rightarrow \sigma$:

1. The first problem we face is the absence of a *clear semantics* of the implication operator $\pi \Rightarrow \sigma$.
2. Second, we need an *efficient implementation* of $\pi \Rightarrow \sigma$ for our stubbing framework.
3. Finally, we should solve a kind of a scheduling problem to ensure that:
   - the implementation produces *productive runs*,
   - if there are several guarantee clauses $\pi_1 \Rightarrow \sigma_1, \dots, \pi_m \Rightarrow \sigma_m$, *bounds* of all $\pi_i \Rightarrow \sigma_i$s are preserved.

To solve the problems, we accept to put constraints on the property languages of π and σ. Our constrains are a compromise between generality of our stubbing framework and its implementability.

8.3.1 Semantics of the Implication Operator

Although properties of the implication form $\alpha \mapsto \beta$ are already part of some assertion languages (e.g., PSL [18505]), they cannot be directly applied for our stubbing framework because they may define liveness properties [AS87] (see Sec. 2.3.2.5). Moreover, semantics of $\alpha \mapsto \beta$ is not well-defined and it may depend on a vendor of a property checking tool implementing the operator. For instance, it is unclear how overlapping of α and β is treated.

We partially solve the boundedness problem of $\pi \Rightarrow \sigma$ by restricting the language used for σ to be *bounded*. We call a language bounded, if there exists a bound on the length of all sequences of the language. We define the semantics of the *overlapping* of π and σ, and the overlapping of several instances of π (see Sec. 8.5 below). Our choices can be discussed, they have consequences for expressivity of $\pi \Rightarrow \sigma$. Nevertheless, we think that having the clear semantics is always a good starting point for any specification. Moreover, we know exactly what we do and it helps us in investigating other problems related to $\pi \Rightarrow \sigma$ and our stubbing framework (see below).

8.3.2 Efficient Implementation

To use $\pi \Rightarrow \sigma$ for our stubbing framework, the idea is to recognize π and to generate σ, when a recognizer of π detects an occurrence of π. To provide the efficient implementation, we need a recognizer of π of a bounded size. We put a constraint on a property language of π and require it to have a *continuous recognizer* (see Sec. 2.4). Moreover, we need to ensure the *absence of backtracking* when producing σ. When there are several guarantee clauses $\pi_1 \Rightarrow \sigma_1 \ldots \pi_m \Rightarrow \sigma_m$, the need for backtracking can appear, for instance, if the generation of σ_k makes the generation of other σ_js ($j \neq k$) infeasible (i.e., the generation reaches a dead-end). We solve the problems by requiring that σ_is *do not share names*. Notice that this constraint also ensures that the guarantee clauses are not contradictory at the logical level.

8.3.3 Productive Runs and a Necessary Condition for Bounds

With the stubbing framework we want to produce productive runs. Moreover, when there are several guarantee clauses $\pi_1 \Rightarrow \sigma_1 \ldots \pi_m \Rightarrow \sigma_m$, we need to ensure that the bound of each $\pi_i \Rightarrow \sigma_i$ (for $i \in [1, m]$) is preserved. To ensure the productiveness of runs, one should perform the scheduling by choosing the *maximum number of inputs* K which can appear between any two consecutive outputs of σ_is. Notice that the productive runs produced with such scheduling are also productive w.r.t. to the outputs of each guarantee clause. In this work we do not investigate the impact of the value of K on the subset of produced productive runs; we assume that this value is given.

To ensure bounds of $(\pi_i \Rightarrow \sigma_i)$s, we require that $(\pi_i \Rightarrow \sigma_i)$s *do not form cycles* (the guarantee clauses do not depend on the outputs of each other). We define a cycle formally in Section 8.5.3. In Section 8.5.4, we show that, given the maximum number of inputs between any two consecutive outputs of σ_is, and provided that $(\pi_i \Rightarrow \sigma_i)$s are cycle-free, it is always possible to compute bounds of $(\pi_i \Rightarrow \sigma_i)$s statically.

In the sections below, we discuss in details assume (A) and guarantee $(\pi \Rightarrow \sigma)$ clauses. In Section 8.5, we analyze our semantic choices for $\pi \Rightarrow \sigma$, and discuss the accepted restrictions on property languages of π and σ.

8.4 Implementation of an Assume Clause

An assume clause A can define any language for which we can produce a recognizer. A is meant to be checked on successive *prefixes* of a run $r = r^0 r^1 \ldots$ at each step. If A does not constrain some interface name $x \in (I \cup O)$, occurrences of x are ignored by A. Checking consists in ensuring that the projection of any prefix r' of r on the names of A does not violate A. The first letter of r on which A does not hold corresponds to the step on which A fails.

Example 8.4.1. An assume clause – Consider a component C with the interface shown in Figure 8.5. Consider an assume clause A of C as a regular expression $(i_1 o_1)^*$. A prefix $r_1 = \underline{i_2} i_1 \underline{o_3 o_3} o_1 \underline{i_3} i_1 o_1$ or a run of C satisfies A (the underlined inputs/outputs are ignored). The last letter of a prefix $r_2 = \underline{i_2} i_1 \underline{o_3 o_3} o_1 \underline{i_3} i_1 o_1 o_1$ of a run of C violates A. □

When several assume clauses $A_1, \ldots, A_n$ are defined for the component C the implicit semantics is their conjunction.

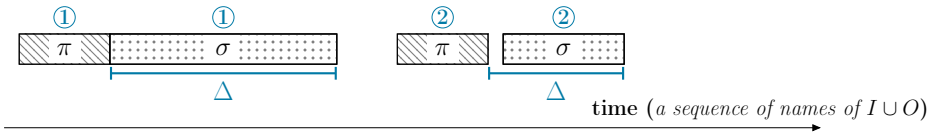


Figure 8.6 Intuitive semantics of a guarantee clause $\pi \Rightarrow \sigma$: if π holds, σ should hold afterwards within a known bound Δ.

To use an assume clause A for stubbing, we need a recognizer of A. The recognizer ignores names which are not constrained by A. It is always active. The first step at which the recognizer complains corresponds to a step where the violation of A occurs.

Example 8.4.2. An Assume Clause of the IPU – Recall the Image Processing Unit (IPU) from our running example. The component has an assume clause A which states: *before face recognition is started the IPU needs to be provided at least once with the address of the image to be analyzed (set-img-addr), the size of the image (set-img-size), the size of the gallery (set-gl-size) and the address of the gallery (set-gl-addr).* The property can be defined by means of the loose-ordering language (Sec. 4.4.3) as follows:

$$\Big(\big(\{\textit{set-img-add}, \textit{set-img-size}, \textit{set-gl-size}, \textit{set-gl-addr}\}, \wedge, \textit{Non-Shuffled}\big) \ll \textit{start} \mid \textit{Non-Repeated}\Big)$$

The recognizer of the assume clause A is defined in Section 5.4.3.1. □

8.5 Restrictions on Property Languages of a Guarantee Clause and Semantic Choices

We choose a guarantee clause to be of the form $\pi \Rightarrow \sigma$, where π is defined on $(I \cup O)$ and σ is defined on O. To explain the semantics of $\pi \Rightarrow \sigma$, we refer to fragments of a run on which π and σ hold.

Definition 19: α-fragment — Let α be a property. A fragment $r' = r^k \dots r^{k+m}$ of a run, on which α holds, is called an α-fragment. We say that α *holds* on a fragment $r' = r^k \dots r^{k+m}$ of a run if (i) the projection of r' on the names of α satisfies α, i.e., $\rho(r', \mathcal{V}_\alpha) \models \alpha$ where $\mathcal{V}_\alpha$ stands for names of α; (ii) the first and the last letters of r' belong to $\mathcal{V}_\alpha$.

An α-fragment is minimal in the sense that it has "borders" which are names of α. Any number of names which are not of α can occur during the α-fragment. We use α-fragments to explain the semantics of a guarantee clause $\pi \Rightarrow \sigma$.

Example 8.5.1. π-fragment – Consider a run $r = o_1 i_1 o_3 o_3 i_2 o_1 i_1 i_1 i_3 o_2 i_2 i_2 \dots$. Assume that π is a regular expression defining only the sequence $i_1 i_2$. The run r contains at least two π-fragments:

$$r = o_1 \underbrace{i_1\ o_3\ o_3\ i_2}_{\pi-\text{fragment}}\ o_1\ i_1\ \underbrace{i_1\ i_3\ o_2\ i_2}_{\pi-\text{fragment}}\ i_2 \dots$$

□

8.5.1 Semantic Choices

The meaning of a guarantee clause $(\pi \Rightarrow \sigma)$ is the following: if π holds, σ should hold afterwards within a known *bound* Δ (Fig. 8.6), and if π holds again, σ should be repeated. A bound Δ of σ is the *maximum number of steps* which may happen between the end of a π-fragment and the end of the respective σ-fragment. A σ-fragment can start either immediately after a corresponding π-fragment (e.g., see ① in Fig. 8.6), or after several steps (e.g., see ② in Fig. 8.6). A property language of σ should be *bounded*, i.e., there exists a bound on the length of all sequences of the language.

There are two possible overlapping situations. The first case is the overlapping of several π-fragments (e.g., see Fig. 8.7). We make a semantic choice: the first π-fragment (① in Fig. 8.7) is taken into account, and all others (e.g., ② and ③) are ignored.

The second case is the overlapping of π- and σ-fragments (Fig. 8.8). It may happen either because of shared names of π and σ, or because names of π occur in σ-fragments. We choose the following semantics. If we have a π-fragment (e.g., ① in Fig. 8.8), we should have a σ-fragment within a bound Δ. There are several cases:

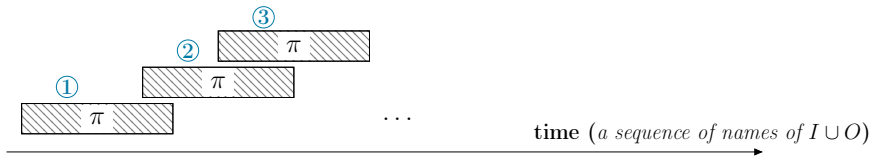


Figure 8.7 Example of overlapping π-fragments.

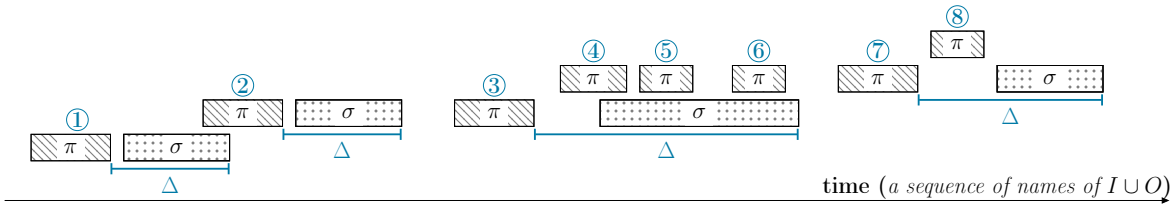


Figure 8.8 Overlapping of π- and σ-fragments: only π-fragments finishing after σ-fragments are taken into account.

(i) if another π-fragment finishes *after* the σ-fragment (e.g., ②, ③ and ⑦ in Fig. 8.8), it is taken into account (i.e., another σ-fragment will take place);
(ii) if another π-fragment finishes *during* the σ-fragment (e.g., ④, ⑤, ⑥), it is ignored;
(iii) if another π-fragment finishes *before* the σ-fragment (e.g., ⑧), it is ignored.

8.5.2 Continuous Recognition of π

To integrate $\pi \Rightarrow \sigma$ in our stubbing framework, we need to perform *efficient* online recognition of a property language of π. π can occur everywhere in a run. There are many ways to recognizer the language of π. One of them is to start a new instance of the recognizer at each step, as if π starts there (e.g., see [Tom+09; GHR03]). The drawback of this solution is that it may lead to the recognizer of an unbounded size [GHR03]; this may cause implementation difficulties.

We decide to restrict a property language of π to be a language which has a *deterministic efficient continuous recognizer* (see Sec. 2.4 in Chapter 2). Anyone who uses the framework should check that the language (s)he defines has this property. Although in Section 2.4 we describe continuous recognizers of regular languages, a complete study of the kind of languages one can use for the generalized stubbing is out of scope of this work. Notice that the language of loose-orderings defined in Chapter 4 can be used for π, since it has a continuous recognizer (see Chapter 5).

8.5.3 Defining Several Guarantee Clauses

A component C can (potentially) define several guarantee clauses ($\pi_i \Rightarrow \sigma_i$)s. The implicit semantics is conjunction. The clauses can be *conflicting* and/or form *cycles*

8.5.3.1 Conflicting Guarantee Clauses

We say that two guarantee clauses $\pi_i \Rightarrow \sigma_i$ and $\pi_j \Rightarrow \sigma_j$ are in conflict, if when both π_i and π_j are observed, there is no way to produce σ_i and σ_j.

Example 8.5.2. Conflicting Guarantee clauses – Consider a component C with an interface as in Figure 8.5. Consider two guarantee clauses $\pi_1 \Rightarrow \sigma_1$, $\pi_2 \Rightarrow \sigma_2$ of C such that: π_1 and π_2 define a singleton $\{i_1\}$; σ_1 (resp. σ_2) defines a singleton $\{o_1 o_2\}$ (resp. $\{o_2 o_1\}$). $\pi_1 \Rightarrow \sigma_1$ and $\pi_2 \Rightarrow \sigma_2$ are in conflict because satisfaction of one violates the other. When π_1 and π_2 are observed, we should start producing σ_1 and σ_2, and it is impossible. □

Moreover, when several σ_is share an output $o \in O$, we have to decide if each σ_i produces its "own" occurrence of o, or if one o is sufficient to satisfy all σ_is. We remove both problems by assuming that σ_is *do not share names*. This constraint also ensures that there is no need for backtracking in the implementation.

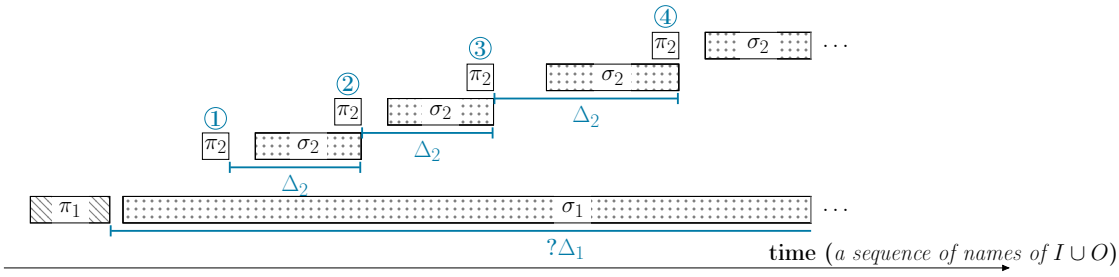


Figure 8.9 A bound Δ_1 of $\pi_1 \Rightarrow \sigma_1$ cannot be computed if $\pi_2 \Rightarrow \sigma_2$ is a cycle such that outputs of σ_2 are π_2-fragments.

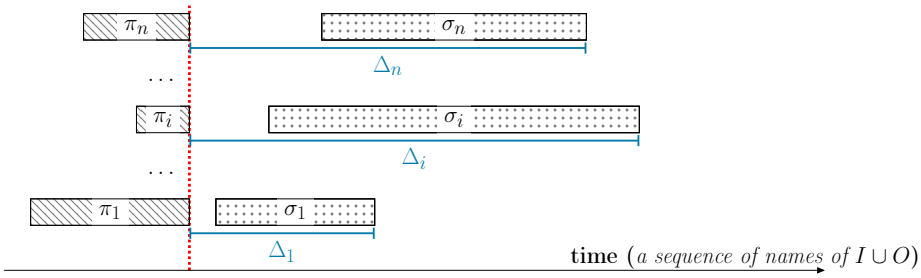


Figure 8.10 Scheduling of σ_i-fragments.

8.5.3.2 Cycles

When there are several guarantee clauses $(\pi_i \Rightarrow \sigma_i)$s such that some of them form *cycles*, it is not always possible to compute statically bounds for all σ_is.

Definition 20: Cycle — A *cycle* is a sequence of guarantee clauses $\{(\pi_i \Rightarrow \sigma_i)\}_m$, $m \geq 1$ such that π_i shares names with σ_{i-1}, and π_1 shares names with σ_m.

Example 8.5.3. Guarantee Clauses forming a cycle – Consider a component C with an interface as in Figure 8.5. Assume that guarantee clauses $\pi_1 \Rightarrow \sigma_1$ and $\pi_2 \Rightarrow \sigma_2$ are such that: π_1 (resp. π_2) defines the set of sequences $\{i_1, o_3\}$ (resp. $\{o_1, o_3\}$); σ_1 (resp., σ_2) defines a singleton $\{o_1 o_4\}$ (resp. $\{o_2 o_3\}$). It is easy to see that π_1 shares names with σ_2, and π_2 shares names with σ_1, thus $\pi_1 \Rightarrow \sigma_1$ and $\pi_2 \Rightarrow \sigma_2$ form a cycle. Moreover, $\pi_2 \Rightarrow \sigma_2$ is a cycle by itself due to the output o_3 shared by π_2 and σ_2. □

A bound Δ_1 of $\pi_1 \Rightarrow \sigma_1$ from Example 8.5.3 is undefined: during a σ_1-fragment $(o_1 o_4)$, infinitely many σ_2-fragments $(o_2 o_3)$ may take place due to π_2-fragments (o_3) finishing on the last output o_3 of σ_2 (Fig. 8.9). To ensure boundedness, we consider only *cycle-free sets of guarantee clauses*; absence of cycles is the necessary condition for bounds. Cycle-freeness defines *constraints on names* of different parts of guarantee clauses. In particular, it implies that π and σ do not share names.

8.5.4 Bounds of a Cycle-Free Set of Guarantee Clauses

We produce productive runs w.r.t. outputs of all guarantee clauses $(\pi_i \Rightarrow \sigma_i)$s (see Def. 14, Sec. 8.2). For that, we schedule outputs of σ_is by choosing the maximum number of inputs K which can occur between any two consecutive outputs of σ_i-fragments. K also defines the number of inputs which can occur between the end a π-fragment and the beginning of a corresponding σ-fragment.

When there is only one guarantee clause $\pi \Rightarrow \sigma$, we can statically compute a bound Δ: the bound is equal to the sum of the *longest sequence* defined by σ, and the *maximum number of inputs* when σ holds. We can find the length of the longest sequence of σ, since a property language of σ is bounded (see Sec. 8.5). The maximum number of inputs, when σ holds, is equal the product of K and the length of the longest sequence of σ.

When there are several guarantee clauses $(\pi_i \Rightarrow \sigma_i)$s, a σ_i-fragment can be postponed, since (potentially) many π_k-fragments can be observed at the same time. Therefore, the σ_i-fragments should be scheduled (e.g., see Fig. 8.10). We can statically calculate Δ_i for each σ_i, if $(\pi_i \Rightarrow \sigma_i)$s are cycle-free; the bound Δ_i of $\pi_i \Rightarrow \sigma_i$ is *less* then the *sum* of:

- the length of the longest sequence of σ_i,

- the maximum number of inputs which can happen during a σ_i-fragment corresponding to the longest sequence of σ_i,
- the sum of the lengths of the longest sequences of all σ_js ($j \neq i$) which can overlap with σ_i.

To get the third summand, we assume the worst case that *all* σ_js ($j \neq i$) can overlap with σ_i (this is equal to $m - 1$, if the number of guarantee clauses is m).

8.5.5 Summary on Constraints

To provide the efficient implementation of the generation part of a stub, and to ensure bounds of guarantee clauses ($\pi_i \Rightarrow \sigma_i$)s, we require that

- the property languages of π_is have deterministic continuous recognizers.

We also impose syntactic constraints on the property languages of π_is and σ_is as follows:

- different σ_is do not share outputs,
- the guarantee clauses ($\pi_i \Rightarrow \sigma_i$)s are *cycle-free*.

When all requirements are fulfilled, we can use ($\pi_i \Rightarrow \sigma_i$)s for our stubbing framework.

8.6 Semantics of a System of Components

We consider a system made of stubs which are connected with oriented point-to-point connections representing an asynchronous system. One way to represent the behavior of such a system is the *asynchronous interleaving* of its individual components (see Def. 6 in Sec. 2.2.2). Since asynchronous components are never alive at the same time, we enable communication between components by using FIFOs. We associate each component with a *unique* input FIFO to preserve order of inputs (e.g., see Fig. 8.12). If the architecture of a system is such that a component C_i is directly connected to a component C_j, (some) outputs of C_i are added to the FIFO of C_j. For instance, in Figure 8.12 stubs C_1 and C_2 are connected to a stub C_3, and C_3 is connected to C_1 and C_2.

Stubs get sequences of inputs and produce sequences of outputs (Fig. 8.11). At each step, a stub can either get one input, or produce one output. If an assume clause of a stub is violated, a stub may report the error.

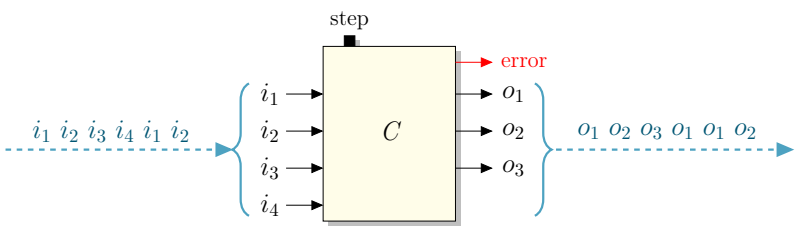


Figure 8.11 A stub receiving (resp. producing) a sequence of inputs (resp. outputs).

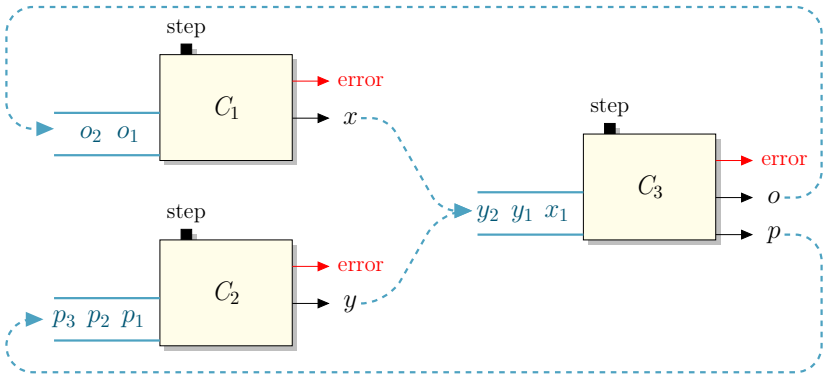


Figure 8.12 Simulation of a system made of stubs.

One path in the full asynchronous product of stubs corresponds to one simulation of the system. Simulation should be performed by a *fair scheduler*. It is a sequence of steps of stubs. At each step, a stub activated by the scheduler either gets the first input from its fifo (if the fifo is not empty), or produces one output. Outputs which a stub should produce are *dynamically* defined. The produced output is added to the input FIFO of a target component. For instance, in Figure 8.12 a stub C_1 (resp. C_2) produced an output x (resp. y) for a stub C_3 once (resp. two times), and C_3 produced an output o (resp. p) for C_1 (resp. C_2) twice (resp. three times). If either its FIFO is empty or a stub does not have outputs to provide, it does nothing. As soon as any of the stubs reports error, simulation stops.

When the simulation is running, FIFOs and stubs are monitored. When all the FIFOs are empty and the stubs do not have outputs to produce, the simulation stops. When the length of any of the FIFOs becomes too large, the simulation stops. The overflow of the FIFOs is likely, if there is a problem in the definition of the stubs. It should not normally happen, since the system is supposed to be run by the fair scheduler and contracts of the stubs are cyclic.

Summary

In this chapter, we have introduced our generalized stubbing framework and have explained how contracts can be used for stubbing. In this chapter, we have been trying to find compromises between expressivity of contracts and implementability of stubs. We have shown our semantic choices for contracts, and have identified constraints on property languages. We have given an intuition about the semantics of a system made of stubs. In the next chapter, we define the semantics of stubs and a stub-based system formally by encoding assume and guarantee clauses into Mealy machines and their synchronous composition. This is the first step toward the implementation of the stubbing mechanism in SystemC.

Chapter 9

Implementation by Encoding into Mealy Machines

Contents

9.1 Principles **145**
9.2 Encoding of a Contract **146**
9.2.1 Encoding of an Assume Clause 147
9.2.2 Encoding of a Guarantee Clause 148
9.2.3 Non-Deterministic Choice Between Inputs and Outputs and Its Interpretation . . 150
9.3 Encoding of the Semantics of a Stub-Based System **151**
9.3.1 Definitions 151
9.3.2 Semantics 151
9.4 Choices on Non-Determinism **153**

In this chapter, we describe the implementation of the generalized stubbing mechanics by encoding contracts into Mealy machines. We assume that contracts are *well-formed*, i.e., all constraints on the property languages of assume and guarantee clauses defined in Chapter 8 are fulfilled. The theoretical foundation of this chapter comprises the synchronous and asynchronous models for discrete concurrent systems defined in Chapter 2 (Sec. 2.2). The reader is assumed to be familiar with the respective material. The encoding is the operational semantics of both stubs and the system made of those stubs. It serves the basis for the SystemC implementation of the execution mechanics presented in Chapter 10.

9.1 Principles

We choose to encode contracts into Mealy machines. Each assume clause A and each guarantee clause $\pi \Rightarrow \sigma$ of a contract is encoded into a Mealy machine; we have one Mealy machine per clause. A Mealy machine encoding an assume clause A *consumes* inputs of a stub. A Mealy machine encoding a guarantee clause $\pi \Rightarrow \sigma$ *produces* outputs of a stub. The synchronous product of the consumer and producer machines is the encoding of a contract. One transition of the synchronous product corresponds to one step of a stub. The encoding has several sources of non-determinism:

1. When a stub makes a step, it can non-deterministically decide either to get an input, or to provide an output. We interpret this source of non-determinism in our model as the absence of the information about the relative speed of stubs.

2. A guarantee clause $\pi \Rightarrow \sigma$ can (potentially) define several output sequences; those sequences are produced non-deterministically.

When there are several guarantee clauses $(\pi_i \Rightarrow \sigma_i)$s, we need to ensure that their bounds are preserved. This is done in an operation way: we use *counters* of events to ensure that only a bounded number of inputs can occur between any two consecutive outputs of the σ_is.

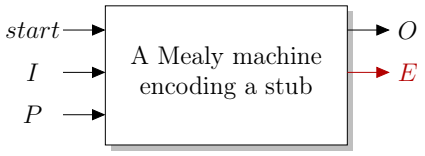


Figure 9.1 A Mealy machine encoding a stub. Arrows from the left (resp. to the right) are inputs (resp. outputs).

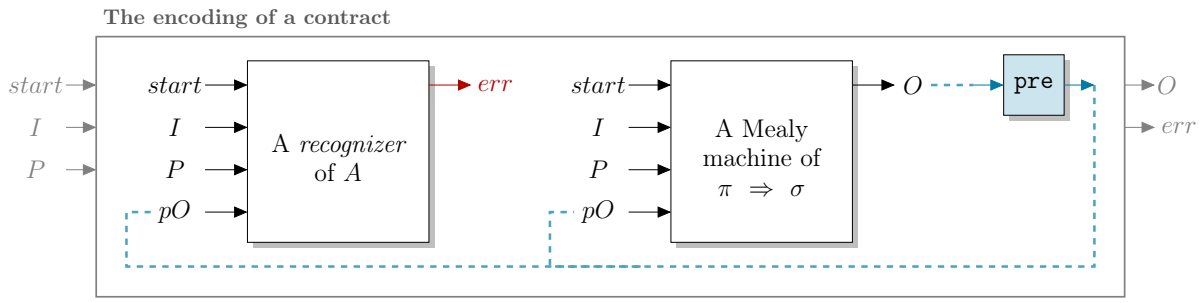


Figure 9.2 The encoding of a stub with an input/output interface (I, O). A is an assume clause, $\pi \Rightarrow \sigma$ is a guarantee clause.

The Structure of the Chapter In Section 9.2, we define in details the encoding of the contract into Mealy machines. In Section 9.3, we define the encoding of the stub-based system's semantics. Finally, in Section 9.4, we summarize the choices on non-determinism in the implementation.

9.2 Encoding of a Contract

We consider a stub C with an input/output interface (I, O). A machine encoding C is a synchronous parallel composition of the recognizers of assume clauses $A_1, \ldots, A_n$, and the Mealy machines encoding guarantee clauses $\pi_1 \Rightarrow \sigma_1, \ldots, \pi_m \Rightarrow \sigma_m$. Figure 9.1 shows inputs and outputs of the Mealy machine encoding C. The *inputs* are:

- inputs the stub (this is I),
- inputs enabling the production of outputs of the stub (this is P).

The *outputs* of the Mealy machine are:

- outputs of the stub (this is O),
- error outputs (this is E).

Elements of the set $P = \{\tau_1, \ldots, \tau_m\}$ are inputs which enable the production of outputs of the respective guarantee clauses $\pi_1 \Rightarrow \sigma_1, \ldots, \pi_m \Rightarrow \sigma_m$. When τ_i occurs (for any $i \in [1, m]$), an output of σ_i is (potentially) produced. Elements of the set $E = \{err_1, \ldots, err_n\}$ are error verdicts of the respective guarantee clauses $A_1, \ldots A_n$. If err_j is delivered (for any $j \in [1, n]$), the assume clause A_j is violated. One step of the Mealy machine encoding a stub C corresponds to one step of C: if an input of I occurs, the stub C gets *one* input; if an input of P occurs, the stub C produces *one* output. At each step, several error outputs can be potentially delivered. This happens if either the consumed input or the produced output violates any of the assume clauses A_js. Notice, only one input of I or P can occur at a time; moreover, inputs of I and P cannot occur simultaneously.

Figure 9.2 illustrates a circuit diagram of the synchronous product of a recognizer of an assume clause A and a Mealy machine encoding a guarantee clause $\pi \Rightarrow \sigma$. Dashed lines in the figure represent the synchronous product of Mealy machines with the encapsulation of respective signals (see the definition in Sec. 2.2.1.2). The recognizer of A gets sequences of inputs of I and outputs of O, and delivers an error output err when the assume clause is violated. The Mealy machine encoding $\pi \Rightarrow \sigma$ produces sequences of outputs of the stub according to the guarantee clause. Inputs of the Mealy machines of A and $\pi \Rightarrow \sigma$ are:

- inputs of the stub (I),
- outputs of the stub (this is pO in Fig. 9.2),
- inputs that enable the production of outputs (P).

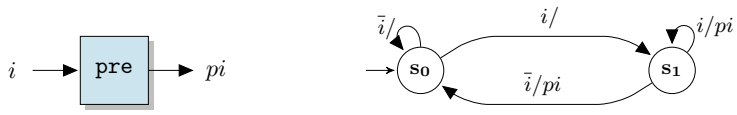


Figure 9.3 The Mealy machine delivering the previous value of its input.

To break any cyclic dependencies which may appear due to outputs of the stub, we use the Mealy machine which emits the *previous* value of its input (Fig. 9.3). This is similar to the `pre(x)` operator of LUSTRE (see Sec. 2.2.1.4). If A (resp. π) does not depend on names of $I' \subseteq I$ or $O' \subseteq O$, the Mealy machine encoding A (resp. $\pi \Rightarrow \sigma$) loops when it gets the ignored names.

In the sections below, we consider in details the recognizer of an assume clause A and the Mealy machine encoding a guarantee clause $\pi \Rightarrow \sigma$.

9.2.1 Encoding of an Assume Clause

We propose to encode an assume clause A into a Mealy machine which is a recognizer of the property language of A. When A is violated, the recognizer delivers an error output err. Figure 9.4 shows a circuit diagram of the recognizer. The inputs $pO = \{po_1, \ldots, po_k\}$ correspond to the outputs of $O = \{o_1, \ldots, o_k\}$ produced at the *previous step*. This fact should be taken into account by the recognizer: when $po_j \in pO$ and $i_\ell \in I$ occur simultaneously, it should be interpreted as a sequence where the output $o_j \in O$ occurs first, and then the input $i_\ell \in I$ takes place. By construction the elements of pO can occur *only* simultaneously with either inputs of I or inputs of P.

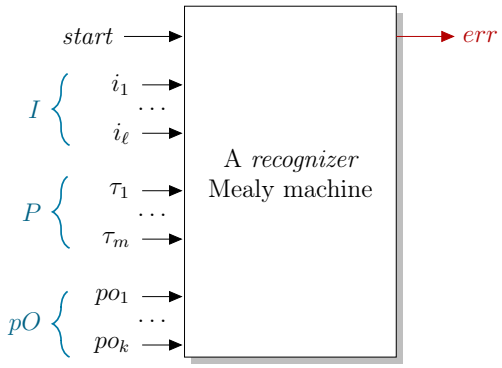


Figure 9.4 A circuit diagram of an error-reporting recognizer of an assume clause A.

.

EXAMPLE 9.2.1. A RECOGNIZER OF AN ASSUME CLAUSE – Consider an assume clause A which states that inputs i_1 and i_2 of a stub should alternate starting from the input i_1. The assume clause is defined as a regular expression $(i_1 i_2)^*$. Figure 9.5 shows the Mealy machine recognizing the property language of A. The state s_0 is the idle state. The state s_3 is the sink error state. □

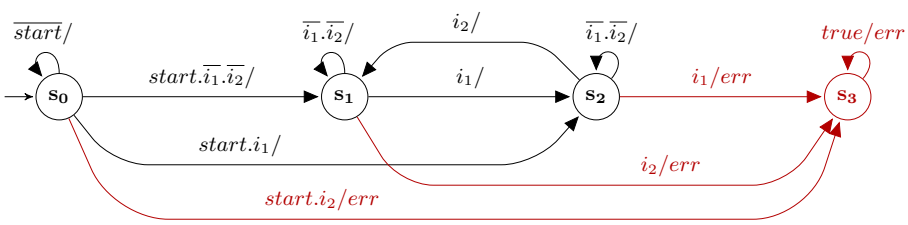


Figure 9.5 A recognizer of an assume clause A defined as $(i_1 i_2)^*$.

9.2.1.1 Encoding of Several Assume Clauses

When there are several assume clauses $A_1, \ldots, A_n$, the resulting Mealy machine is the synchronous product of Mealy machines encoding the A_is. All Mealy machines have inputs as shown in Figure 9.4. Error

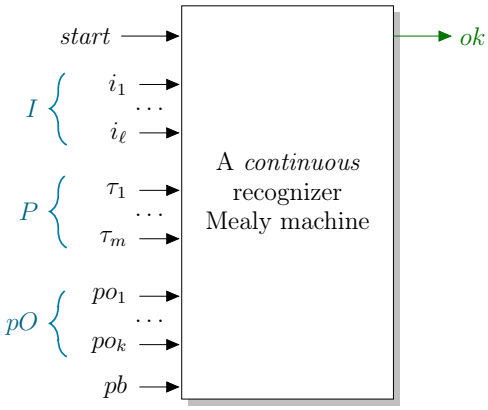


Figure 9.6 A circuit diagram of a continuous recognizer of π. Arrows from the left (resp. to the right) are inputs (resp. outputs).

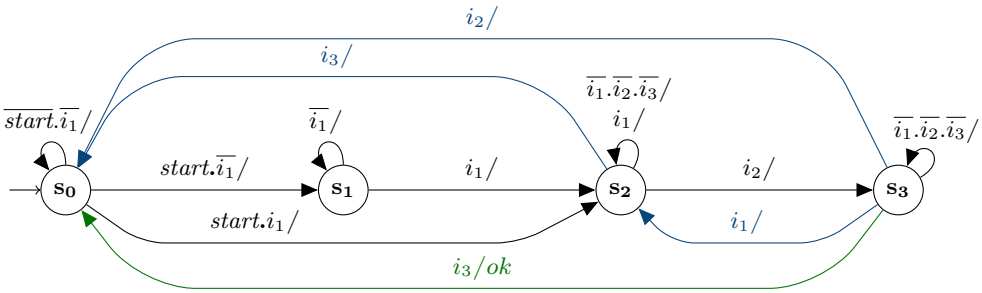


Figure 9.7 A deterministic continuous recognizer of π defining one sequence $\{i_1 i_2 i_3\}$.

outputs of the recognizers of the A_is are outputs of the synchronous product. They form the set $E = \{err_1, \ldots, err_n\}$ (see Fig. 9.1).

9.2.2 Encoding of a Guarantee Clause

The encoding of a guarantee clause $\pi \Rightarrow \sigma$ is made by composing sequentially the *continuous recognizer* of π, the *generator* of σ and the *counter machine*. The counter Mealy machine is used to ensure a bound of $\pi \Rightarrow \sigma$.

9.2.2.1 A Continuous Recognizer of an Antecedent π

Figure 9.6 shows a continuous recognizer of π. The Mealy machine has the same set of inputs as the recognizer of an assume clause. It gets sequences of inputs of I and outputs of O produced at the previous step (the inputs of pO), The synchronous product is enabled by adding the inputs of $P = \{\tau_1, \ldots, \tau_m\}$. The continuous recognizer loops when names which do not appear in π occur. The recognizer has a special input pb used for scheduling (the input is connected to the counter Mealy machine, see details in Sec. 9.2.2.4). Inputs of I can occur *only if the input pb is absent*. When the continuous recognizer detects an occurrence of π, it delivers a Boolean value ok.

Example 9.2.2. A Continuous Recognizer – Consider π defining a singleton $\{i_1 i_2 i_3\}$. Figure 9.7 illustrates a deterministic continuous recognizer of π. s_0 is the initial state. The Mealy machine moves when the names of π occur. If the recognizer detects matching with a prefix of the sequence $i_1 i_2 i_3$, it advances the recognition. Otherwise, the Mealy machine re-starts by moving either to the initial state s_0, if i_2 or i_3 occurs, or to s_2, if i_1 occurs. When the sequence $i_1 i_2 i_3$ is recognized, the Mealy machine emits the output ok and moves to the initial state s_0. The input *start* is always true; thus, the machine re-starts at the next step. □

9.2.2.2 A Generator of a Consequent σ

We choose to encode a generator of σ as a (potentially) non-deterministic Mealy machine. Our generator is illustrated in Figure 9.8. For convenience, here we suppose that there is only one guarantee clause

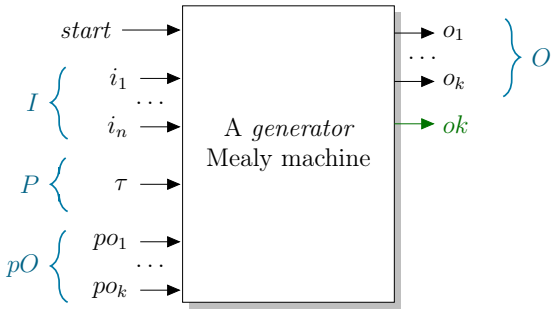


Figure 9.8 A circuit diagram of a generator Mealy machine of σ. Arrows from the left (resp. to the right) are inputs (resp. outputs).

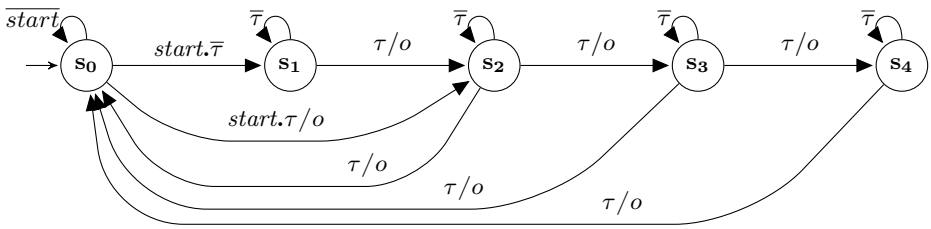


Figure 9.9 A Mealy machine generating sequences of a range $o^{[2,4]}$.

$\pi \Rightarrow \sigma$, i.e., (i) σ depends on all outputs O of the stub, and (ii) the set P is a singleton $\{\tau\}$ such that τ enables the production of outputs of O. The generator can be activated with the input *start*. When the machine is active and τ occurs, one of the outputs of O is emitted. When the generator finishes the production of an output sequence, it emits the output *ok* and moves to the idle state. To enable the synchronous product with the continuous recognizer of π, the generator can get as inputs all inputs I, all inputs pO, and, if there are several guarantee clauses, all auxiliary inputs $P\backslash\{\tau\}$. The Mealy machine generating σ loops on occurrences of inputs I, $P\backslash\{\tau\}$ and pO .

Example 9.2.3. A Generator – Consider σ as a range $o^{[2,4]}$. Figure 9.9 shows the Mealy machine generating the property language of σ. The machine is idle in the state s_0. In s_1, the machine is active and ready to start the generation. When the input τ occurs in the states s_1, s_2, s_3, s_4, the machine emits the output o. In s_2 (resp. s_3), the machine non-deterministically decides to produce a sequence of either two or three (resp. three or four) occurrences of o. When a sequence is generated, the Mealy machines moves to the idle state s_0. □

9.2.2.3 A Counter Mealy Machine

A circuit diagram of a counter Mealy machine is shown in Figure 9.10. The counter can be started (resp. stopped) with the input *start* (resp. *stop*). When the counter is active, the input *start* is ignored, the Mealy machine counts occurrences of *step*. The counter has a parameter K. When the number of steps is equal to or greater than K, the counter emits the output *bound*. The machine can be reset with the input *reset*. In this case the machine stays active, but re-starts the counting of steps from zero.

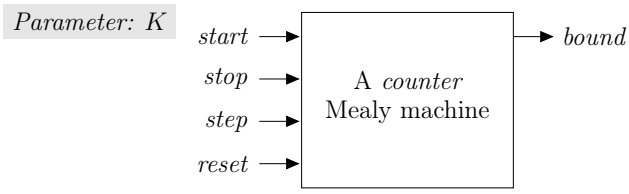


Figure 9.10 A circuit diagram of a counter Mealy machine. Arrows from the left (resp. to the right) are inputs (resp. outputs). The machine emits *bound*, when *step* occurs more than K times without *reset*.

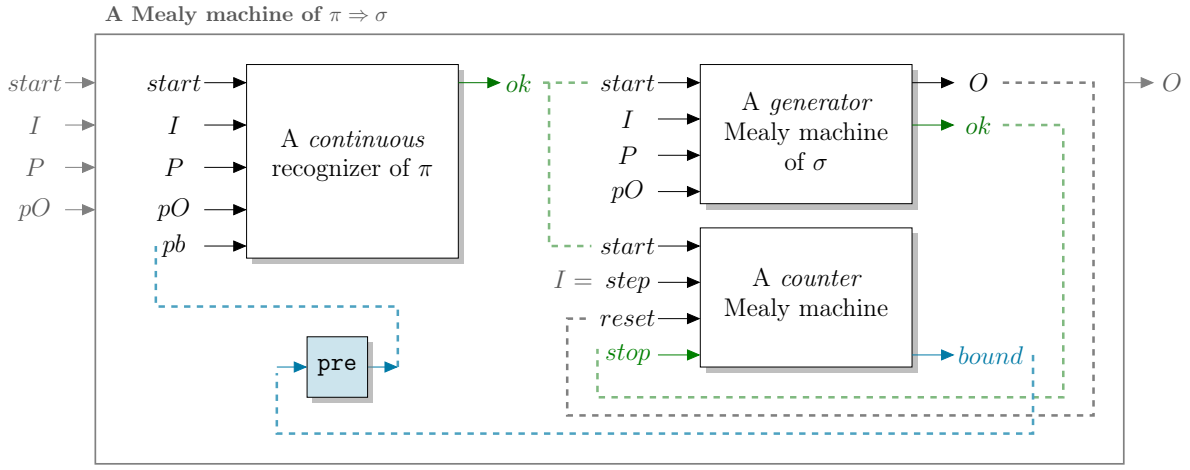


Figure 9.11 The encoding of a guarantee clause $\pi \Rightarrow \sigma$.

9.2.2.4 The Synchronous Product

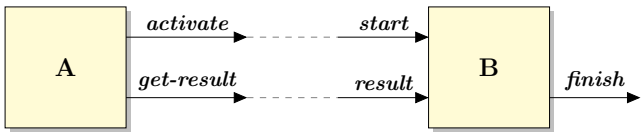
Figure 9.11 shows a circuit diagram of the synchronous parallel composition of the continuous recognizer of π, the generator of σ, and the counter Mealy machine. When the continuous recognizer detects an occurrence of π and emits the output *ok*, the counter machine and the generator machine of σ are started. When the generator finishes the production of σ and emits the output *ok*, the counter machine is stopped. Neither the counter nor the generator can be re-started when they are active. The counter machine is reset whenever the generator of σ produces an output from O; the counter increases whenever any input from I occurs. If the counter detects the maximum allowed number of inputs (which is specified by the value of its parameter K), the counter emits *bound*. This output is connected to the input *pb* of the continuous recognizer of π through the Mealy machine `pre`. By choosing the value of K one can perform the scheduling of a stub and ensure the boundedness of $\pi \Rightarrow \sigma$.

9.2.2.5 Encoding Several Guarantee Clauses

When there are several guarantee clauses $\pi_1 \Rightarrow \sigma_1, \ldots, \pi_m \Rightarrow \sigma_m$, the encoding is the synchronous product of the Mealy machines encoding all the $(\pi_i \Rightarrow \sigma_i)$s. Each guarantee clause $\pi_i \Rightarrow \sigma_i$ (for all $i \in [1, m]$) has its own τ_i. This ensures that no spurious synchronizations between the σ_is occur when outputs are produced. These inputs of the Mealy machines form the set $P = \{\tau_1, \ldots, \tau_m\}$.

9.2.3 Non-Deterministic Choice Between Inputs and Outputs and Its Interpretation

In the proposed implementation, at each step of a stub the choice between getting an input, or producing an output can be done non-deterministically (if the production is not forced by the counter machine, see Sec. 9.2.2.4). This non-deterministic choice can result in the violation of assume clauses of the stub. For instance, a stub **B**, with the inputs ***start***, ***result*** and the output ***finish*** (see Fig. 9.12), can either get an input (e.g., ***result***) from its fifo, or provide an output (***finish***). Suppose that **B** assumes to get ***result*** only after at least one ***finish***. If a step of **B** consists in getting the input ***result*** from the fifo before ***finish*** is produced, the assume clause of the stub is violated. If a step of **B** consists in producing the output ***finish***, the simulation continues normally. Our model is untimed and we interpret this kind of non-determinism as the unknown relative speed of stubs.



A guarantees: ***activate***, and then ***get-result***

B assumes: at least one ***finish*** before ***result***

Figure 9.12 Specification of two connected stubs.

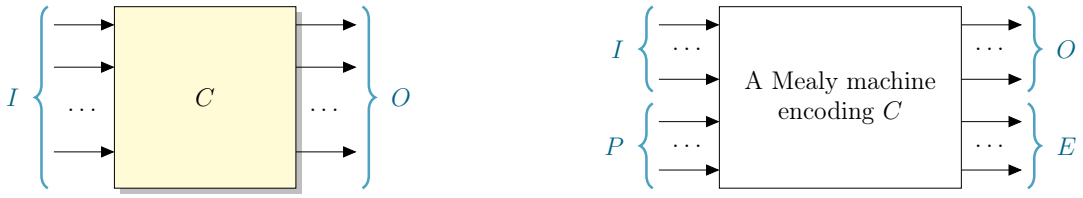


Figure 9.13 An input/output interface of a stub C (from the left), and the interface of a Mealy machine encoding C (from the right).

9.3 Encoding of the Semantics of a Stub-Based System

We consider a system S made of stubs $C_1, \ldots, C_n$ connected by oriented point-to-point connections and exchanging data. The semantics of the system S is the total asynchronous interleaving of Mealy machines encoding the C_is. We enable the communication between the asynchronous stubs by means of FIFOs. Each stub has a single input FIFO. The FIFO ensures that the order of received inputs of I is preserved. Each stub consumes inputs from its FIFO, and puts produced outputs to FIFOs of target stubs. One path of the asynchronous product corresponds to one simulation of the system S. Each transition of the path corresponds to a step of one of the stubs C_is of the system. The first transition where any of the stubs delivers an error output(s) indicates a step where an error occurs. To choose one path (to perform a simulation of a system), one needs to use a scheduler.

9.3.1 Definitions

Oriented point-to-point connections of the stubs C_is are defined by the system's *architecture*.

Definition 21: Architecture — Let S be a system made of stubs $C_1, \ldots, C_n$. An *architecture* of the system S is a function *arch* defined as follows: for each stub C_i, for each output o of the C_i, $arch(o) = C_j$ results in a stub C_j to which C_i is connected via the output o.

The asynchronous stubs C_is communicate by means of FIFOs.

Definition 22: Fifo — A FIFO F is a sequence of pairs (i, val), where i corresponds to one input port of an associated stub, and $val \in Dom$ is a value of the input i (Dom is a domain of values, it can be any user-defined type).

In the sequel, we will denote the size of a FIFO F as $|F|$. To define the semantics of a system made of stubs, we use three functions:

- $peek\ (F) = (i, val)$ returns (but does not remove) the first pair (i, val) of the FIFO,
- $add\ \big(F, (i, val)\big) = F'$ adds a pair (i, val) to the FIFO F and returns the obtained FIFO F',
- $poll\ (F) = F'$ removes the first pair from F and returns the obtained FIFO F'.

9.3.2 Semantics

The encoding of the system S is the asynchronous product of the Mealy machines encoding the stubs C_is. A Mealy machine encoding a stub C_i is the synchronous product of Mealy machines encoding assume and guarantee clauses of C_i.

9.3.2.1 Recall: Properties of the Synchronous Product

An input/output interface of a stub C is a pair (I, O), where I is the set of inputs, and O is the set of outputs (Fig. 9.13). Consider C with assume clauses $A_1, \ldots, A_n$, and guarantee clauses $\pi_1 \Rightarrow \sigma_1, \ldots, \pi_m \Rightarrow \sigma_m$. The Mealy machine encoding C has the following interface (Fig. 9.13):

- Inputs are:
  - the actual inputs of the stub C (this is I),
  - auxiliary inputs which enable production of outputs of C (this is P).
- Outputs are:
  - the actual outputs of the stub C (this is O);
  - error outputs (this is E).

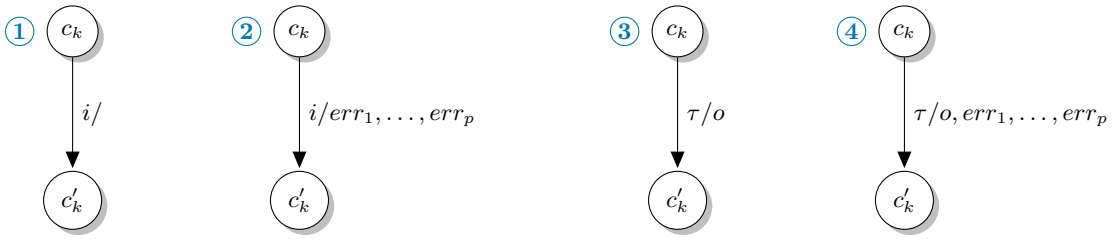


Figure 9.14 The types of transitions of the Mealy machine encoding a stub C: $i \in I$, $\tau \in P$, $\{err_1, \ldots, err_p\} \subseteq E$, $o \in O$.

The elements of the set $P = \{\tau_1, \ldots, \tau_n\}$ enable the production of pairwise disjoint sets of outputs of C; this is because σ_is do not share outputs (see Sec. 8.5.5). The elements of the set $E = \{err_1, \ldots, err_m\}$ are error outputs of the Mealy machine encoding C. They are delivered if assume clauses of C are violated.

Due to the way we build the synchronous product of the recognizers of assume clauses and the generators of guarantee clauses (see Sec. 9.2), the Mealy machine encoding C has transitions of only four types (Fig. 10.1):

1. only one input of I occurs, no outputs are produced;
2. only one input of I occurs, error outputs of E are delivered;
3. only one input of P occurs, only one output of O is produced;
4. only one input of P occurs, only one output of O and (potentially) several error outputs of E are produced.

9.3.2.2 Semantics of a System

Let the function *arch* be the architecture of the system S, and let $F_1, \ldots, F_n$ be the input FIFOs of the stubs $C_1, \ldots, C_n$ respectively. A state of the system is made of the states of all its stubs C_is and all input FIFOs F_is. We denote a state of a component C_i as c_i and a state of a FIFO F_i as f_i (for all $i \in [1, n]$). We denote a state of the system as $s = \big[(c_1, \ldots, c_n),\ (f_1, \ldots, f_n)\big]$. We denote a transition of the system as (s, ℓ, s'), where ℓ is a label of a transition.

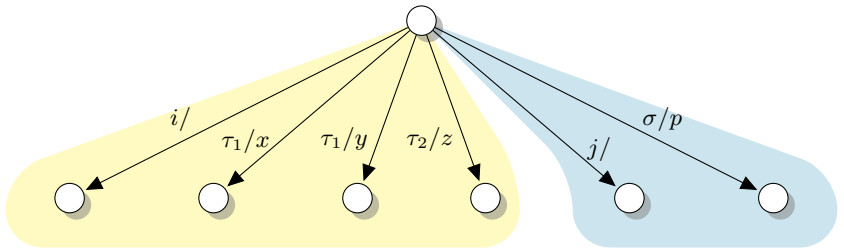
The semantics of the system S is define as follows: Rule 9.1 defines a step of the system S when one of its stubs C_k get an actual input from a corresponding input FIFO F_k and no errors occur, i.e., a transition of the type 1 is triggered. In this case: (i) C_k changes a state, (ii) the first element of F_k is removed, and (iii) states of all other stubs and FIFOs are unchanged.

$$\frac{\exists (c_k, i/, c'_k) \text{ in } C_k,\ |f_k| \neq 0,\ \mathit{peek}\ (f_k) = (i, \mathit{val})}{\exists \Big(\big[(c_1, \ldots, c_k, \ldots, c_n),\ (f_1, \ldots, f_k, \ldots, f_n)\big],\ i/,\ \big[(c_1, \ldots, c'_k, \ldots, c_n),\ (f_1, \ldots, f'_k, \ldots, f_n)\big] \Big) \text{ in } S \quad \text{such that } f'_k = \mathit{poll}(f_k)} \tag{9.1}$$

$$\frac{\exists (c_k, \tau/o, c'_k) \text{ in } C_k,\ \mathit{arch}(o) = C_j}{\exists \Big(\big[(c_1, \ldots, c_k, \ldots, c_n),\ (f_1, \ldots, f_j, \ldots, f_n)\big],\ \tau/o,\ \big[(c_1, \ldots, c'_k, \ldots, c_n),\ (f_1, \ldots, f'_j, \ldots, f_n)\big] \Big) \text{ in } S \quad \text{such that } f'_j = \mathit{add}(f_j, o)} \tag{9.2}$$

Rule 9.2 defines a step of the system S when one of its stubs C_k produces an actual output and no errors occur, i.e., a transition of the type 3 is taken. In this case: (i) C_k changes its state, and (ii) the produced output is added to the input FIFO of a target component C_j, and (iii) states of all other stubs and FIFOs are unchanged.

The system has a global error state. If any of the stubs C_is makes a step and reports an error(s), i.e., a transition either of the type 2 or of the type 4 is triggered, the system moves to the global error state and the simulation stops.



(a) A state of a system with several outgoing transitions.

(b) Input/output interfaces of two stubs of a system

Figure 9.15 A state of a system (a) with several outgoing transitions corresponding to steps of stubs (b). τ_1, τ_2 (resp. σ) enables production of outputs of a stub A (resp. B).

9.4 Choices on Non-Determinism

By construction the Mealy machine encoding the semantics of a system made of stubs is such that each state of the Mealy machine can have several outgoing transitions triggered by occurrences of inputs (transitions of the types 1 and 2, see Fig. 10.1), and/or several outgoing transitions producing outputs of the stubs (transitions of the types 3 and 4). Each transition corresponds to a step of some stub of the system. There are several sources of non-determinism that should be controlled:

1. There is a non-deterministic choice between stubs to be activated; this corresponds to the choice of a subset of the transitions (Fig. 9.15). This choice should be fair in order to avoid overflow of the stubs' fifos.

2. When a stub is chosen, one needs to decide if the stub gets an input or produces an output. For instance, in Figure 9.15(a), it corresponds to the choice between the transition labeled by $i/$ and the transitions with labels τ_1/x, τ_1/y, τ_2/z, provided that the stub A is activated. The choice between inputs and outputs is not completely free, because stubs should produce only productive runs.

3. Finally, the production of outputs of guarantee clauses (e.g., the choice between τ_1/x, τ_1/y and τ_2/z) is non-deterministic. This non-determinism can be completely free unless one cares about the coverage of the specification.

Summary

In this chapter, we have introduced the encoding of stubs and systems made of stubs into Mealy machines. This encoding is the implementation of our generalized stubbing mechanics, which is independent of any programming language or a scheduler. In the next chapter, we show the realization of this implementation in SystemC/TLM, as well as the experiments.

Chapter 10

Execution Mechanics: Implementation with the SystemC Scheduler

Contents

10.1 Implementation Principles **155**
10.1.1 Enabling Communication 156
10.1.2 Implementing a Stub 156
10.2 Simulation with the SystemC Scheduler **157**
10.2.1 Recall: the Semantics of a Stub-Based System 157
10.2.2 A Scheduling Algorithm 157
10.3 Experiments **160**
10.3.1 Experimental Settings 160
10.3.2 Observations and Results 160

This chapter provides the SystemC implementation of stubs and systems made of stubs. This implementation serves a proof-of-concept of the generalized stubbing mechanics. There are many ways to implement stubbing in SystemC; here we provide one of the possible implementations, which follows the traditional style of defining SystemC/TLM components and virtual prototypes. Thus, it should be immediately usable by people used to SystemC/TLM. The implementation is convenient in both cases: if one develops a SystemC/TLM virtual prototype in a top-down manner, or wants to make the existing SystemC/TLM model more abstract by adding non-determinism and removing some implementation details. Our goal here is to show that the simulation with stubs works, and can be used to find synchronization bugs early in the model of the system. To accomplish the goal, we choose an arbitrary scheduling policy to activate stubs and produce their outputs; the scheduling is performed by the SystemC scheduler. The SystemC implementation is the translation of the operational semantics of stub-based systems defined in Chapter 9.

In this chapter we also show the experiments with the intercom system. The experiments demonstrate that (i) the stubbing mechanics can save effort spent to write the model, (ii) it provides the capability to expose bugs and to blame the faulty components, (iii) it can reduce simulation time.

The chapter is organized as follows: In Section 10.1, we present the SystemC implementation of the generalized stubbing mechanics. Section 10.2 explains the simulation of a stub-based system with the SystemC scheduler. Finally, in Section 10.3 we discuss the experiments with the running example.

10.1 Implementation Principles

We consider a system made of stubs. Each stub has assume $A_1, \ldots, A_n$ and guarantee $\pi_1 \Rightarrow \sigma_i, \ldots, \pi_m \Rightarrow \sigma_m$ clauses; their property languages satisfy the requirements defined in Chapter 8. The SystemC implementation of the stub-based system defines:

- the communication mechanism,
- the implementation of stubs,
- the scheduling policy (this is discussed in Section 10.2).

10.1.1 Enabling Communication

Stubs communicate by sending/receiving blocking transactions and interrupt requests. Transactions and interrupt requests are inputs and outputs of the stubs. In our implementation stubs do not have explicit input fifos; the order of inputs is preserved because transactions and interrupts are executed by means of *functional calls* on target stubs, and they are sent by asynchronous components. Such communication influences the subset of runs observed at simulation (this is discussed in Section 10.2.2.1). The communication principle by means of transactions is as described in Section 3.2.3.3. Interrupt requests are sent as follows: if stub A sends an interrupt request to a stub B, it calls a function `send_irq()` of B. Notice that we do not use here SystemC signal channels because they do not guarantee the preservation of the interrupts' order.

10.1.2 Implementing a Stub

The general idea of a stub's implementation in SystemC is the following. A stub is a *SystemC module*, which has a *checking part* for assume clauses $A_1, \ldots, A_n$, and a *generation part* for guarantee clauses $\pi_1 \Rightarrow \sigma_1, \ldots, \pi_m \Rightarrow \sigma_m$. There are many ways to implement the checking part; for instance, in Figures 10.6 and 10.7 it is implemented as defensive code. The main requirement for the implementation is that it can perform *steps* when inputs and/or outputs occur, and can deliver verdicts, if any of the assume clauses $A_1, \ldots, A_n$ are violated.

The generation part for guarantee clauses $\pi_1 \Rightarrow \sigma_1, \ldots, \pi_m \Rightarrow \sigma_m$ is made of:

- the continuous recognizers of π_is implemented, for instance, as defensive code (see Fig. 10.7). The recognizers should perform *steps* when inputs and/or outputs of the stub occur, and deliver Boolean values whenever occurrences of π_is are detected;
- the generators of σ_is implemented by means of SystemC processes, one process per σ_i. Each process has a `while(true)` loop; one iteration of the loop corresponds to the generation of one sequence defined by σ_i (e.g., see Fig. 10.8). The generation starts, when an occurrence of a corresponding π_i is detected. Thus, when π_i occurs, a SystemC immediate event is notified (`start_event.notify()`); this event can resume the generation process. When the generation of a sequence of σ_i is finished, the process calls the `wait(TIME)` function.

In our implementation we use two types of SystemC events[1]; these events are notified in corresponding states (phases) of the SystemC scheduling algorithm[2]:

- *immediate events* (e.g., `start_event.notify()`) are used to start generation processes of σ_is, when corresponding π_is are detected. The generation process of σ_i, waiting for the event `start_event`, is resumed by the SystemC scheduler at the same evaluation when the event is notified, i.e., π_i has been detected;

- *timed events* (timeouts of `wait(TIME)`) are used by the generation processes to yield the control to the scheduler. The generation process of σ_i, waiting for a timeout, is resumed by the scheduler after advancement of the simulation time.

We need to take into account the scheduling algorithm of the SystemC kernel, in order to implement the semantics of a guarantee clause $\pi \Rightarrow \sigma$. In particular, we need to check that overlapping of occurrences of π and σ are treated as defined in Section 8.5.1. We also need to ensure bounds for guarantee clauses and productiveness of stubs (this is discussed in Section 10.2.2.3).

Implementing the Semantics of a Guarantee Clause Immediate SystemC events are not persistent; this can be a reason of the violation of the semantics of $\pi \Rightarrow \sigma$ defined in Section 8.5.1. Recall, the semantics of $\pi \Rightarrow \sigma$ is such that whenever π occurs, σ should occur after within a bound. If at the same evaluation phase the generation of σ is finished first (i.e., the generation process of σ calls `wait(TIME)`), and then π occurs (the event `start_event` is notified), the detected occurrence of π will be "lost", because the generation process misses the notified event.

To avoid such situation, we use flags as follows: When π occurs, a flag is set. When the generation process of σ is resumed after calling `wait(TIME)`, it checks the flag: if the flag is set, the process directly start generation of another occurrence of σ, otherwise it calls `wait(start_event)`.

[1] The types of SystemC events are described in the background chapter (see Sec. 2.1.4.4).
[2] The scheduling algorithm of the SystemC kernel is explained in Section 2.1.4.8 of the background.

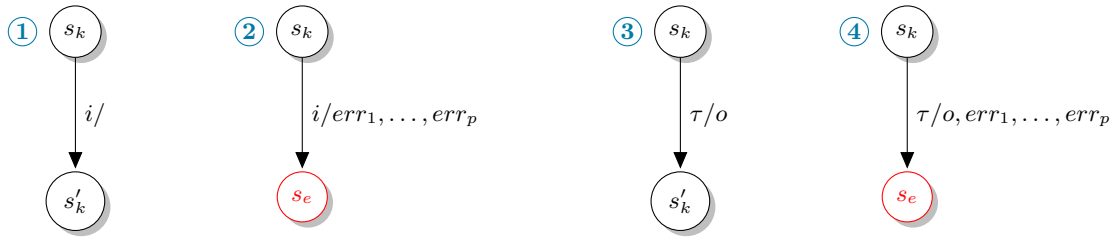


Figure 10.1 The types of transitions of the Mealy machine encoding a stub-based system. s_e is the global error state of the system.

10.2 Simulation with the SystemC Scheduler

A system made of stubs is executed by the SystemC scheduler. The scheduler produces a subset of runs of the system. To describe this subset, we first briefly recall the encoding of a stub-based system into Mealy machine proposed in Chapter 9; one path of this machine corresponds to one run of the system. Afterwards, we analyze the influence of the communication mechanism and the implementation of stubs on the system's scheduling, i.e., the set of investigated paths of the system's Mealy machine.

10.2.1 Recall: the Semantics of a Stub-Based System

By construction, the Mealy machine encoding a stub-based system is such that each its state can have several outgoing transitions corresponding to steps of the system's stubs; one transition per step. The transitions can be of four types shown in Figure 10.1; these types correspond to two types of stubs' steps:

- if a stub gets an input from its fifo and the input does not violate (resp. violates) the assume clause(s) of the stub, a transition of type 1 (resp. 2) is taken; here, i is the input, and $err_1, \ldots, err_p$ are error outputs,
- if the stub produces an output and the assume clause(s) are not violated (resp. are violated), a transition of type 3 (resp. 4) is taken; here, o is the output, τ enables production of o, and as before $err_1, \ldots, err_p$ are error outputs.

The system has a global error state, where it enters if transitions of the type 2 or 4 are triggered. This is state s_e in Figure 10.1.

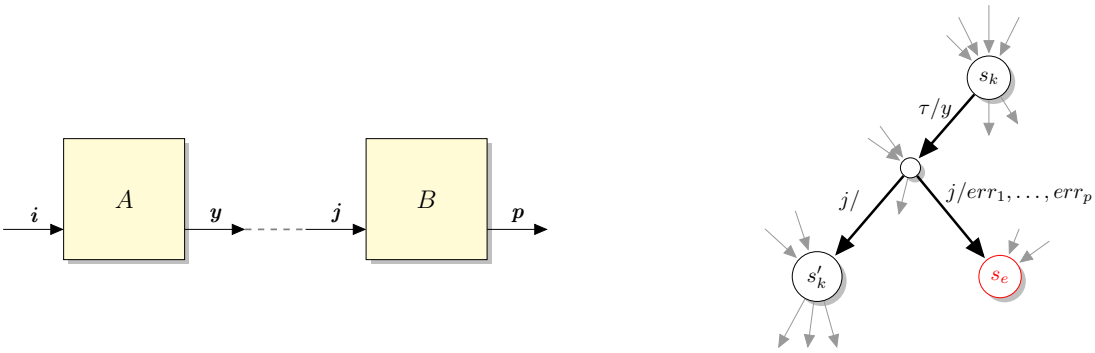
10.2.2 A Scheduling Algorithm

Our SystemC implementation and the non-preemptive nature of the SystemC scheduler define a subset of the system's runs which can be observed at simulation time.

10.2.2.1 Impact of the Communication Mechanism

The communication between stubs is implemented by means of functional calls. It means that stubs have the *same relative speed*, and the non-deterministic choice between getting inputs and producing outputs of stubs (see Sec. 9.4) is solved by choosing to get inputs as soon as they are provided (i.e., appear in the stubs' input fifos). This is done without any intervention of the SystemC scheduler.

For instance, consider two stubs A and B connected as shown in Figure 10.2(a). The implementation is such that when A produces output y (e.g., a transaction), the output is immediately executed on the target stub B (i.e., B gets input j), and the control returns to A. Figure 10.2(b) shows the transitions of the Mealy machine encoding the system with A and B, which are triggered when A communicates with B. The transition labeled with τ/y is triggered when A produces y; according to the semantics of the system (see Sec. 9.3.2), the produced output is added to the input fifo of B. Then, the transition labeled with $j/$ is triggered; the input j is removed from the B's fifo and B makes a step. Notice that input j can violate assume clauses of B, in this case the system enters the error state s_e and the simulation stops. Also notice that our communication mechanism ensures that the input fifos always have at most size 1, since inputs are immediately consumed by target stubs.



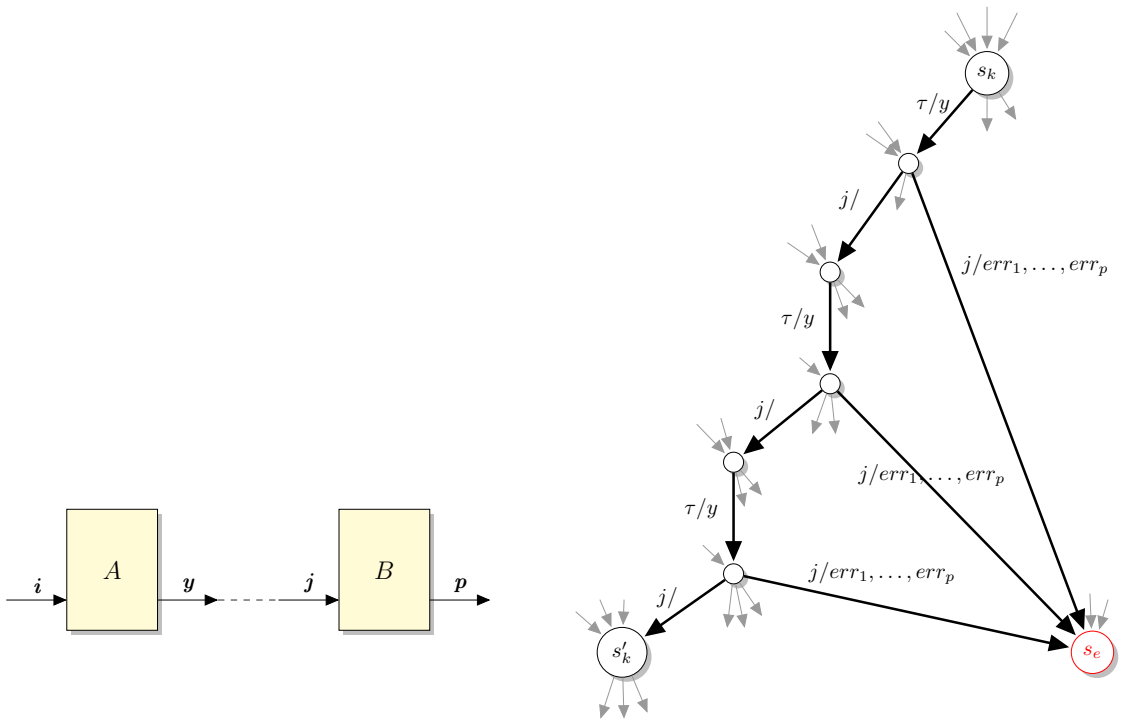
(a) Connected stubs

(b) The system's transitions triggered when A sends output y to B.

Figure 10.2 Impact of the communication mechanism on the execution of the stub-based system: output y of A is connected to input j of B; τ enables production of y.

10.2.2.2 Impact of the Atomicity of SystemC Processes

SystemC process producing outputs of stubs has to yield by calling the `wait()` function, because the SystemC scheduler is non-preemptive. An atomic step of the SystemC process corresponds to a sequence of statements between two consecutive calls to `wait()`. Some of those statements can actually produce outputs; therefore, several transitions of the system can be triggered as shown in Figure 10.3. Here, the path from state s_k to state s'_k (or s_e) corresponds to one atomic step of the SystemC process generating an output sequence of stub A. In s_k, the process is resumed by the SystemC scheduler; in s'_k, the process yields by calling `wait()`. Notice, at any intermediate state the checking part of stub B, which consumes the produced outputs, can (potentially) detect an error(s); in this state the system enters the error state s_e and the simulation stops.



(a) Connected stubs

(b) The system's transitions triggered when A produces output y.

Figure 10.3 Impact of the atomicity of the SystemC processes on the execution of the stub-based system: output y of A is connected to input j of B; τ enables production of output y of A.

```
// when π1 is detected:
   flag1 = true;
   start1.notify();

// generation of σ1:
void gen1(){
   while(true){
     if(!flag1)
       wait(start1);
     //produce outputs
     wait(t'1);
     //produce outputs
     flag1 = false;
     wait(t''1);
 }}
```

```
// when π2 is detected:
   flag2 = true;
   start2.notify();

// generation of σ2:
void gen2(){
   while(true){
     if(!flag2)
       wait(start2);
     //produce outputs
     flag2 = false;
     wait(t2);
 }}
```

```
// when π3 is detected:
   flag3 = true;
   start3.notify();

// generation of σ3:
void gen3(){
     if(!flag3)
       wait(start3);
     //produce outputs
     wait(t'3);
     //produce outputs
     flag3 = false;
     wait(t''3);
 }}
```

Figure 10.4 The SystemC implementation of three guarantee clauses $\pi_i \Rightarrow \sigma_i$ for $i \in [1, 3]$.

10.2.2.3 Ensuring Bounds of Guarantee Clauses and Productiveness of Stubs' Runs

In the proposed SystemC implementation bounds of guarantee clauses ($\pi_i \Rightarrow \sigma_i$)s are mapped on simulation time. The bound Δ of a guarantee clause $\pi \Rightarrow \sigma$ is equal to $\Delta = \sum T_i$, where T_is are time arguments of all calls to the `wait()` function in the generation process of σ. The advancement of simulation time in the generation processes after each iteration of corresponding `white(true)` loops is mandatory to ensure bounds. This also guarantees that runs of the stubs are productive.

10.2.2.4 Wrap-up: A Scheduling Principle

Consider a system made of stubs such that $\pi_1 \Rightarrow \sigma_1, \ldots, \pi_k \Rightarrow \sigma_k$ are guarantee clauses of the stubs, and $gen_1, \ldots, gen_k$ are the corresponding SystemC processes producing output sequences defined by $\sigma_1, \ldots, \sigma_k$ respectively. Figure 10.4 schematically shows the implementation of three guarantee clauses. Here, when π_i is detected, event $start_i$ is notified and flag $flag_i$ is set; gen_i is the generation process of σ_i, and t_is are the process's timeouts. The production of outputs is scheduled by the SystemC kernel as follows:

1. To start the simulation, at least one of the generation process of σ_is has to be able to produce outputs. That is, one of the π_is should be considered detected and the respective flag should be set (e.g., $flag_1$ of gen_1 in Fig. 10.4).

2. At the *evaluation phase*, the SystemC scheduler resumes some generation process of σ_is that are ready to produce outputs (e.g., gen_1). Produced outputs can make other generation processes of σ_is runnable, if those outputs constitute occurrences of corresponding π_is (e.g., gen_3 waiting for $start_3$ is resumed, if outputs produced by gen_1 make an occurrence of π_3). When the evaluation phase finishes, some generation processes of σ_is are waiting for timeouts (e.g., gen_1 and gen_3 in Fig. 10.4 calling respectively `wait(`t'_1`);` or `wait(`t''_1`);` and `wait(`t'_3`);` or `wait(`t''_3`);`), other generation processes are waiting for occurrences of corresponding π_is (e.g., gen_2 calling `wait(`$start_2$`);`.

3. When there are no runnable proccesses, the SystemC *advances simulation time*; some generation processes waiting for timeouts (e.g., gen_1 and gen_3) can become runnable and the scheduler proceeds to the evaluation phase.

4. The simulation stops either when any of the checking parts of the stubs is violated, or when none of the generating processes can produce outputs. The latter situation happens if after the advancement of simulation time there are no runnable processes, i.e., all the processes are waiting for occurrences of corresponding π_i.

10.2.2.5 Summary: the Sources of Non-Determinism and SystemC Scheduling

As stated in Section 9.4, a stub-based system has several sources of non-determinism:

- the activation of stubs,
- the choice of the type of a step (get an input or produce an output) performed by the activated stub,

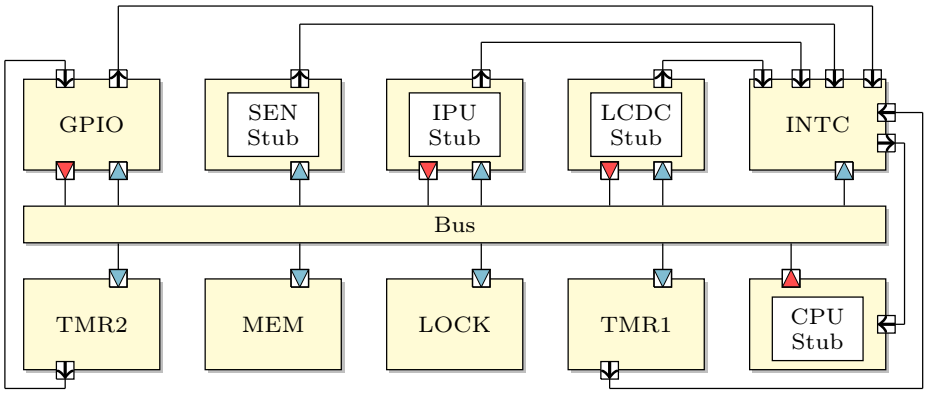


Figure 10.5 The TLM platform of the Smart Intercom device with stubs.

- the choice of an output produced by the activated stub.

In the proposed SystemC implementation stubs are activated to get inputs as soon as those inputs are received. The activation of stubs to produce outputs is controlled by means of yielding points of the generation processes, which are implemented with the `wait(TIME)` function.

10.3 Experiments

This section shows the results of the experiments with the intercom system (see Chapter 3). The SystemC/TLM model of the intercom is *loosely-timed*; the components interact by means of blocking transactions.

10.3.1 Experimental Settings

For each component of the intercom system we define an interface and a contract; some contracts are defined by means of the loose-ordering language presented in the first part of this document (e.g., see Sec. 3.4). We compare three versions of the intercom system:

1. Full is the virtual prototype of the intercom system as it is defined in Chapter 3.
2. Def: some components are coupled with the SystemC monitors for loose-ordering properties (see Chapter 6).
3. Stubs: some components of the intercom are coupled with the SystemC monitors for loose-ordering properties, and others are replaced with stubs (Fig. 10.5); the stubs and components send/receive interrupt requests by means of functional calls.

The criteria of the comparison are:

- effort spent to write the model of the system,
- capability to expose bugs and blame the faulty components,
- simulation time.

10.3.2 Observations and Results

Three types of simulations corresponding to three considered versions of the intercom have been performed:

1. The simulation of Full alone allows to see what *types of bugs* we can find in a detailed TLM model, and how those bugs are *detected*. This is described in details in Chapter 3. The bugs BS1 – BS15 were detected with this type of simulation.

2. The simulation of Def version of the intercom helps not only to detect all the bugs of Full (BS1 – BS15), but also to find their sources. In particular, it has been established that some of the bugs have several causes. The results on this simulation are provided in Section 7.2 (see Tables 7.3 and 7.4).

3. When STUBS is simulated, since stubs implement contracts of the components, some of the intercom's synchronization bugs have been removed simply by construction. For instance, the simulation of DEF has established that the bug BS2 "*the display of the intercom is always black*" occurs either because the CPU fails to respect its promises and "forgets" to start the LCDC, or because the SEN "forgets" to send an interrupt request when it captures an image. This bug is removed from STUBS, because the SEN and the CPU are replaced by their stubs which ensure that the components respect their guarantees. The bugs BS3 "*the display shows nonsense*", BS4 "*the system gets stuck at face recognition:* BUTTON-START *does not respond, the same (currently analyzed) image is shown on the display*", BS12 "*the user does not see any notification image on the screen when the face recognition is finished*" are removed by construction when the simulation with STUBS is performed. The rest of the synchronization bugs of the intercom (e.g., BS7 "*the registered used gets the access denied notification*") have been detected and localized either by the checking parts of the stubs (e.g., BS7 has been detected by the checking part of the IPU's stub), or by the SystemC monitors of the fully implemented SystemC/TLM components.

The comparison of the simulations with three versions of the intercom system allows us to make the following conclusions:

- **Effort spent to write the model:** the contract-only versions reduces the effort. For instance, if we focus on the IPU, instead of describing the full behavior of the component (in our case the procedure comparing images, see Sec. 3.2, Fig. 3.7), the contract-only version simply produces the corresponding outputs of the component, ignoring data (see Fig. 10.8). Thus, it replaces a timing- and function-precise sequences of memory reads by a random number of reads where data is ignored. Moreover, contracts are more relaxed than detailed models, keeping only the essential interaction effects. The simulation with contracts has the potential to cover more cases, and the software validated with this new model is therefore more robust.

- **Capability to expose bugs and blame the faulty components:** this is where the gain is the most significant. The contract-like specification style helps blaming the component in which something has to be corrected (if the assume clause of the IPU complains, then the embedded software that provides inputs to the IPU is to be blamed).

- **Simulation time:** if a component contains heavy function computations (e.g., the IPU implementing actual face recognition), the simulation time for the contract-only version will be decreased considerably.

Summary

In this section, we have presented one of the possible SystemC implementations of the generalized stubbing mechanism. We have shown how to implement a stub as a SystemC module, how to define the architecture of a system made of stubs, and how to integrate a stub into a SystemC/TLM virtual prototype. We also have presented the experiments with the intercom system. The results showed (i) the proof of concept for the simulation with stubs and its advantages, (ii) the application of the loose-ordering language to the definition of stubs.

```
#include "ipu.h"
#include "offsets/ipu.h"
#include "stdlib.h"
#include <assert.h>

IPU::IPU(sc_core::sc_module_name name):
      sc_core::sc_module(name){
    SC_THREAD(face_recognition_analysis);
    sensitive << start_event;
    irq_out.initialize(false);
    /* initialize the registers */
    image_size = 0;
    image_addr = 0;
    gallery_size = 0;
    gallery_addr = 0;
    confidence_value = 0;
}

/* 'read' access to registers */
tlm::tlm_response_status IPU::read(addr_t addr, data_t &d){
  switch(addr){
  case IPU_GET_CNF_VAL_OFFSET:
    d = confidence_value; break;
  case IPU_CFG_IMAGE_OFFSET:
    d = image_addr; break;
  case IPU_CFG_GAL_OFFSET:
    d = gallery_addr; break;
  case IPU_CFG_IMAGE_SIZE_OFFSET:
    d = image_size; break;
  case IPU_CFG_GAL_SIZE_OFFSET:
    d = gallery_size; break;
  default:
    SC_REPORT_ERROR(name(), "register is not implemented,
                    or it is of 'write-only' type");
    return tlm::TLM_ADDRESS_ERROR_RESPONSE; }
  return tlm::TLM_OK_RESPONSE; }
```

```
#include "ipu.h"
#include "offsets/ipu.h"
#include "stdlib.h"
#include <assert.h>
#include "dispatcher.h"

IPU::IPU(sc_core::sc_module_name name):
      sc_core::sc_module(name){
    SC_THREAD(face_recognition_analysis);
    sensitive << start_event;
    /* initialize the registers */
    image_size = 0;
    image_addr = 0;
    gallery_size = 0;
    gallery_addr = 0;
    confidence_value = 0;
    /* initialize variable for defensive code */
    fl_img_addr = false; fl_img_size = false;
    fl_gl_addr = false;  fl_gl_size = false;
    fl_irq = false;      flag_start = false;
    /* seed for random generation */
    srand(time(NULL)); }

/* 'read' access to registers */
tlm::tlm_response_status IPU::read(addr_t addr, data_t &d){
  switch(addr){
  case IPU_GET_CNF_VAL_OFFSET:
    assert(fl_irq); fl_irq = false;
    d = confidence_value; break;
  case IPU_CFG_IMAGE_OFFSET:
    d = image_addr; break;
  case IPU_CFG_GAL_OFFSET:
    d = gallery_addr; break;
  case IPU_CFG_IMAGE_SIZE_OFFSET:
    d = image_size; break;
  case IPU_CFG_GAL_SIZE_OFFSET:
    d = gallery_size; break;
  default:
    SC_REPORT_ERROR(name(), "register is not implemented,
                    or it is of 'write-only' type");
    return tlm::TLM_ADDRESS_ERROR_RESPONSE; }
  return tlm::TLM_OK_RESPONSE; }
```

Figure 10.6 Implementation of the checking part of the IPU component (form the right-hand side). Highlighted lines are the added defensive code. Part I.

```
/* 'write' access to registers */
tlm::tlm_response_status IPU::write(addr_t addr, data_t d){
 switch(addr){
 case IPU_CFG_IMAGE_OFFSET:
  image_addr = d;
  break;
 case IPU_CFG_GAL_OFFSET:
  gallery_addr = d;
  break;
 case IPU_CFG_IMAGE_SIZE_OFFSET:
  image_size = d;
  break;
 case IPU_CFG_GAL_SIZE_OFFSET:
  gallery_size = d;
  break;
 case IPU_START_OFFSET:
  start_event.notify();
  break;
 default:
  SC_REPORT_ERROR(name(), "register is not implemented");
  return tlm::TLM_ADDRESS_ERROR_RESPONSE;}
 return tlm::TLM_OK_RESPONSE; }
```

```
/* 'write' access to registers */
tlm::tlm_response_status IPU::write(addr_t addr, data_t d){
 switch(addr){
 case IPU_CFG_IMAGE_OFFSET:
  image_addr = d;
  fl_img_addr = true;
  break;
 case IPU_CFG_GAL_OFFSET:
  gallery_addr = d;
  fl_gl_addr = true;
  break;
 case IPU_CFG_IMAGE_SIZE_OFFSET:
  image_size = d;
  fl_img_size = true;
  break;
 case IPU_CFG_GAL_SIZE_OFFSET:
  gallery_size = d;
  fl_gl_size = true;
  break;
 case IPU_START_OFFSET:
  assert(fl_img_addr && fl_gl_addr && fl_img_size && fl_gl_size);
  fl_img_addr = false; fl_img_size = false;
  fl_gl_addr = false;  fl_gl_size = false;
  flag_start = true;
  start_event.notify();
  break;
 default:
  SC_REPORT_ERROR(name(), "register is not implemented");
  return tlm::TLM_ADDRESS_ERROR_RESPONSE;}
 return tlm::TLM_OK_RESPONSE; }
```

Figure 10.7 Implementation of the checking part of the IPU component (from the right-hand side). Highlighted lines are the added defensive code. Part II.

```
/* the face recognition procedure */
void IPU::face_recognition_analysis(){
 while(true){
   wait(start_event);

   /* 1. configure registers */
   confidence_value = 0;
   /* 2. configure buffers */
   captured_image_buffer=new data_t [image_size/sizeof(data_t)];
   gallery_image_buffer =new data_t [image_size/sizeof(data_t)];
   /* 3. read the image under analysis */
   get_image(image_addr, captured_image_buffer);
   /* 4. compare with images from the gallery */
   data_t temp_conf_value = 0;
   data_t result = 0;
   addr_t gallery_image_addr;
   int i = 0;
   while ( i < gallery_size ){
     gallery_image_addr = gallery_addr + i * image_size;
     get_image(gallery_image_addr, gallery_image_buffer);
     temp_conf_value = compare_facial_features();
     if(temp_conf_value > result){
       result = temp_conf_value;
       recogn_img_addr = gallery_image_addr;}
     i += 1;}
   /* 5. compute the confidence value */
   std::srand(time(0));
   if(result == 1){
     std::cout<<name()<< ": User was identified"<<std::endl;
     confidence_value = rand()%30 + 70;
   } else
     confidence_value = rand()%70;
   /* 6. mimic the computation time */
   wait(100, sc_core::SC_NS);
   /* 7. discard configuration of registers */
   image_addr = 0;
   gallery_addr = 0;
   gallery_size = 0;
   image_size   = 0;
   /* 8. discard configuration of buffers */
   delete [] captured_image_buffer;
   delete [] gallery_image_buffer;
   /* 9. send an interrupt */
   irq_out.write(true);
   wait(5, sc_core::SC_NS);
   irq_out.write(false); }}
```

```
/* the face recognition procedure */
void IPU::face_recognition_analysis(){
 while(true){
   if (!flag_start)
     wait(start_event);

   /* 1. configure registers */
   confidence_value = 0;
   /* 2. configure buffers */
   /* 3. read the image under analysis */
   /* 4. compare with images from the gallery */
   int repeat_count = 20 + (std::rand() % 19000);
   data_t dummy_read_data;
   for ( int i = 0; i < repeat_count; i++ ){
     initiator_socket.read(addr, dummy_read_data);}
   /* 5. compute the confidence value */
   confidence_value = 1 + std::rand() % 100;
   /* 7. discard configuration of registers */
   image_addr = 0;
   gallery_addr = 0;
   gallery_size = 0;
   image_size   = 0;
   /* 8. discard configuration of buffers */
   /* 9. send an interrupt */
   dispatcher->send_irq();
   flag_start=false;
   wait(500, sc_core::SC_NS); }}
```

Figure 10.8 Implementation of the generation part of the IPU (from the right-hand side). Highlighted lines correspond to the removed and simplified parts of the implementation of the IPU.

Part V

Related Work and Conclusions

Chapter 11

Related Work

Contents

11.1 Functional Verification of SystemC/TLM . . . 167
11.1.1 Specification Languages and TLM assertions . . . 167
11.1.2 Monitoring SystemC/TLM . . . 168
11.1.3 Verification Methodologies and Libraries . . . 172
11.2 Functional Verification Through the Design Flow of SoCs . . . 173
11.2.1 Transactor-Based Verification: Properties Reuse . . . 173
11.2.2 Properties Refinement . . . 173
11.3 Contracts for SystemC/TLM . . . 174
11.3.1 Contracts for Hardware Designs . . . 174
11.3.2 Other Uses of Formally-Defined Contracts . . . 174
11.4 Virtual Prototyping of SoCs . . . 174
11.4.1 Early Detection of Bugs . . . 174
11.4.2 Raising Abstraction Level Beyond TLM . . . 175

The specification/verification method we propose in this work is compatible with most of work proposed nowadays in the domain of electronic system verification. In this chapter, we provide high-level overview of the related work with respect to both our loose-orderings and our generalized stubbing with contracts.

11.1 Functional Verification of SystemC/TLM

This work is targeting functional verification of SystemC/TLM. Thus, it is not surprising that the big part of related work comes from the respective field. When checking functional correctness of the SystemC/TLM design one may ask the following questions:

1. How to defined properties? Specifically, which specification language to choose and what aspects of the design to capture?
2. How to interpret the defined properties?
3. How to ensure at simulation that the design satisfies its specification?
4. Finally, how to check quality of the used testbench? In other words, when one can be sure that there are no behaviors of the design which would violate any of the properties?

In the following sections, we examine how different works answer these questions and compare those works with our proposals.

11.1.1 Specification Languages and TLM assertions

The largest amount of related work is in the set of *specification languages* for hardware/software systems (e.g., *e* [IJ04], Sugar/PSL [CVK04]). There are a lot of proposals for extensions of temporal logics, with past or future operators, allowing the conjoint use of regular expressions, etc.

Ecker et al. A *conceptual* language for specifying TL properties is presented in [Eck+06b]. The proposed constructs of the language are interpreted on sequences of *events* of a TL model. The language is (very) similar to PSL with semantics being adapted to asynchronous TL models. In [Eck+06a] a specialized language for transaction level assertions is defined. The language is a SystemC extension of SVA which allows expression of properties on SystemC events. A possible implementation of the language is described more precisely in [Eck+07]. This approach is appealing, but can be seen unnecessarily complicated because the temporal fragments of PSL and SVA languages are rich enough to express a large range of properties, even at the TLM level. To a certain extent, syntactic sugar could be considered, but new semantic constructs are not necessary since various modalities (such as *until, before*) are present in the language and are given a clean trace semantics.

Tabakov et al. [Tab+08] presents a temporal logic for SystemC, but focusing on what happens in the simulation kernel.

Tomasena et al. In [Tom+09] the TLM assertions are defined as *sequences of propositions* about transaction events occurrence in a defined order. Those propositions are combined to describe a transaction events pattern by means of binary logical connectives (conjunction, disjunction, implication). Proposition definition includes (i) a *trigger part* which specifies the sequential relationship between the time instants of the transaction events involved in the evaluation of the property, (ii) *time-out* condition, and (iii) *expression* which is evaluated only when the trigger is true. As the simulation of the DUV produces events, the propositions of the property change their state. The authors claim that the proposed TL assertions can be used to express system properties expressible in PSL. The *time-out* attribute of the proposed TL assertions considers simulation time of the executable SystemC/TLM model.

As we see in Chapter 8 all of the existing specification languages can be used to define stubs in our methodology, provided that they satisfy our requirements on property languages. Moreover, (i) automatic translation into efficient monitors (observers) should be available, preferably in C or C++, and (ii) to generate outputs, efficient Boolean/numerical constraint solvers should exist, at least for a subset of the language, and offer an API in C or C++. In Chapter 4, it is shown that our loose-ordering properties can be encoded in most of available specification languages supporting regular expressions and linear temporal operators; however such encoding can lead to very inefficient monitoring (see Chapter 7). Our loose-orderings can extend existing industrial specification languages with new constructs. Time-outs defined in [Tom+09] are particularly interesting for us because they are similar to our concept of a bound of a guarantee clause (Chapters 4 and 8).

11.1.2 Monitoring SystemC/TLM

When properties of a SystemC/TLM design are defined and encoded in some specification language, we want to check their satisfaction at simulation. For that it is needed (i) to *capture* relevant events occurring at the design when it is up and running, (ii) to have monitors of the properties which are updated on occurrences of (some of) those events. Therefore, the related work on this matter includes monitoring models of SystemC/TLM and construction of monitors for properties.

11.1.2.1 Monitoring and Property Evaluation Models

Despite difference in the vocabularies, most of the recently proposed monitoring models for SystemC/TLM are similar. They all enable online checking of TLM assertions which specify *communication protocols* of TL models. The properties are evaluated on *sequence of events*. The models rely on dedicated entities which allow capturing of communication events. Those entities are either redefined classed of communication channels or special functions which are called when events occur. Implementation of the models require some instrumentation of the original SystemC code with the defined entities. Most of the monitoring models implement the *observer patter* [Gam+95] to notify monitors about occurrences of events. When events occur, monitors change their state. Below we examine some of the monitoring models available for SystemC/TLM. Our monitoring model for online checking of loose-ordering properties, discussed in details in Chapter 6, is inspired by the model proposed in [PF08].

Pierre et al. In [PF08] the authors propose a method for monitoring SystemC/TLM specifications which define properties regarding *communication of the design*. Properties are defined in the Foundation Language (FL) class of PSL which represents the linear temporal logic, i.e., the properties are PSL formulas made of the operators `until!`, `always`, etc. The semantics of the specification is defined on the execution traces of a system at the TLM level. A trace is made of *observation points*, i.e., points where the assertions needs to be re-evaluated. Assertions are re-evaluated each time a variable of the assertion has possibly been modified, i.e., an appropriate event occurs in the corresponding communication channel. Hence, the traces are made of all the events that enable the observation of updated values for the variables involved in the assertions.

Monitoring of the design's properties relies on the model defined in [Pie07] (see Fig. 11.1) and it is inspired by the observer pattern. The approach is relatively non-intrusive, it allows monitoring of properties that involve several channels possibly of different types. To monitor SystemC/TLM design at simulation, one needs to defined *observable* subclasses for the channels under observation (*Subjects*) and to use them in place of their parent classes. The observable channels and/or signals involved in a property are observed by monitors enclosed in a wrapper associated with that property.

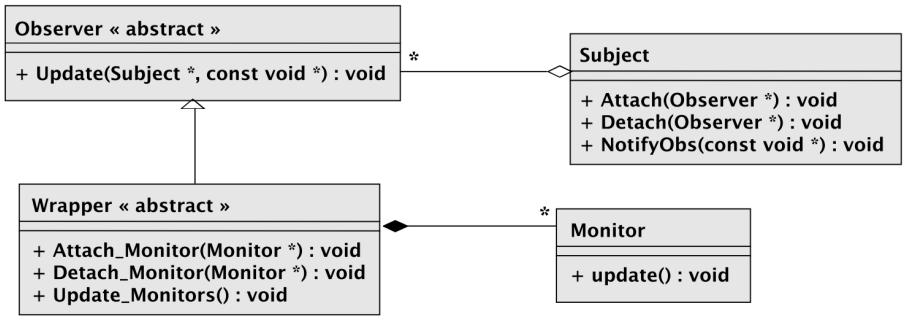


Figure 11.1 A model for observing transactions. (Source [Pie07])

In [PF10] the authors provide monitoring support for the PSL Modeling Layer which enables the use of (global) auxiliary variables in assertions. It allows to monitor components with pipelined behavior enabling the simultaneous processing of several data by considering *reentrant assertions*. Those assertions are evaluated simultaneously for different data through the use of multiple checker instances with local variables. For that the PSL syntax is extended with an appropriate syntactical construct, and the semantics are accordingly adapted.

Xiong et al. In [XBZ10] properties defining the communication protocol of a TL model are formalized in PSL. They can depend on fields of transactions, calculation results of several transactions, etc. PSL properties are evaluated on the sequences of transactions. The order of those transactions is determined by sorting respective timing information about their occurrence. If a TL model is untimed, a global counter is used. The counter is increasing by one whenever a transaction event occurs.

Values of transactions parameters are extracted at simulation by dedicated functions associated with respective TLM ports. To collect properties, the SystemC/TLM design is instrumented with dedicated entities, called here monitors, according to the observer pattern. Those entities report transaction information with current simulation time information to the global data collector whenever the data arrives, i.e., every state change of a transaction. The notification mechanism is implemented outside the SystemC simulation context.

Tomasena et al. In [Tom+09] TLM assertions are defined by means of XML files. Assertions are communication properties of the TL model, they are defined as sequences of propositions about transaction events occurring in a defined order (see Sec. 11.1.1). At simulation when the design produces communication events, the state of propositions changes. The time order is defined by the simulation time of the SystemC scheduler. The assertion checking framework is supported by a library built on top of SystemC. Figure 11.2 shows the proposed monitoring model. To integrate the framework within the SystemC simulation and to perform evaluation, one needs to replace transactional ports with *Monitorable Transactional Ports* (MTPs) provided by the framework, and register them in the assertions manager which is responsible for the coordination of the monitoring process. The assertion manager parses the XML files where assertions have been specified, and generates the list of assertions. One monitor is created per MTP. All

monitors are registered inside the assertions manager which finds assertions with appropriate monitors. The assertion manager supports the *overlapping* and *non-overlapping* modes. The overlapping mode may contain several instances of the property; evaluation of a new instance of the property is (potentially) started with each occurrence of the DUV's event. Once the simulation of the DUV is finished, the assertions manager collects the statistical information of the monitoring process and generates an XML report file.

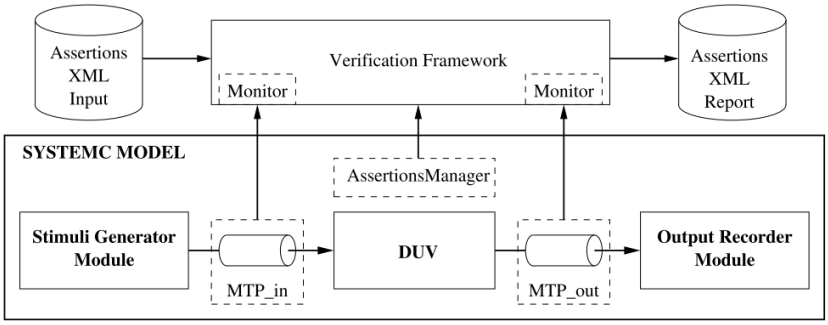


Figure 11.2 Verification framework with XML assertions. MTP_in, MTP_out are Monitorirable Transactional Ports (Source [Tom+09])

Tabakov et al. In [TRV12] and [TV10], the authors propose monitors for checking properties for SystemC. The monitors are built based on the temporal logic introduced in [Tab+08].

11.1.2.2 Compositional Building of Monitors

In this work we define direct translation of loose-orderings into efficient SystemC monitors (Chapter 5). Our monitors are built in a compositional way, and can be automatically generated from a loose-ordering property (Chapter 6). Similar approaches for producing monitors are discussed below. They all use *primitive monitors* to build more complex monitors. Such composition building helps to ensure that the obtained monitors are correct.

Dahan et al. [Dah+05] defines the translation of PSL to SystemC/C++. The translation consists of three steps. First, each PSL property is translated into a nondeterministic finite automaton (NFA). The NFA has a set of error states, entering an error state means that the design does not adhere to the specification under the test conditions. Secondly, the NFA is converted into a deterministic automaton (DFA). Finally, the DFA is translated to the target C++ language as defined in [Aba+00]. The generated C++ checker consists of the *checker logic* (state machine transition function and the assertion condition) generated from the PSL assertions, and the *checker control code* which calls the checker's reset and transition functions. It is the user's responsibility (i.e., a part of the monitoring model definition), to define (i) when the checker's reset and transition functions should be invoked (i.e., when the generated monitor is updated), and (ii) when to sample the design signal values to assign them to the corresponding checker ports (i.e., when to capture the design's events). The C++ checkers are wrapped into SystemC modules that are later on connected as read-only monitors to the SystemC platform. Although the reported results show that the checkers can be beneficial in finding, localizing and fixing bugs, the checkers are used only for SystemC Bus Cycle Accurate model of the SoC and no evidence of their potential application to SystemC/TLM models at higher abstraction levels is provided.

Pnueli et al. In [PZ06] monitors are referred to as *testers*. Temporal testers are non-deterministic transducers which at any point (step) output a Boolean value if the corresponding temporal formula holds starting at the current position, i.e., if the suffix of the input sequence is in the language of the formula. A tester is compositional, it is constructed out of the testers for its sub-formulas. Testers for both the LTL and SERE operators of PSL are defined. The authors state that for the SERE subset of PSL the complexity of testers is linear in the size of the respective SERE formulas. To the best of our knowledge, this work is purely theoretical and was not implemented in SystemC/C++.

Pierre et al. The monitor in [PF08] is generated from the property thanks to a method that simply interconnects the elementary monitors associated with the primitive operators of a PSL property (Fig. 11.3). The method adapts the proved correct compositional construction of monitors at the RT level [MAB06]. In [MAB06] a monitor of a property takes as inputs the reset, the synchronization clock, a signal *Start* that triggers the evaluation, and the signals of the DUV that are operands of the PSL operators in the property (e.g., see Fig. 11.3). The primary outputs *Checking* and *Valid* are those of the rightmost PSL operator. The TRUE value of the former (resp. the latter) indicates that output *Valid* is effective at the next synchronization time (resp. absence of error). The combined values of *Checking* and *Valid* indicate whether the property is inactive, currently being computed, or known and passed or failed. Primitive single operator monitors are interconnected to build complex monitors. The method is based on the syntax tree of the property: the output *Checking* of a monitor of a parent operator is connected to input *Start* of a monitor of its child operator (Fig. 11.3).

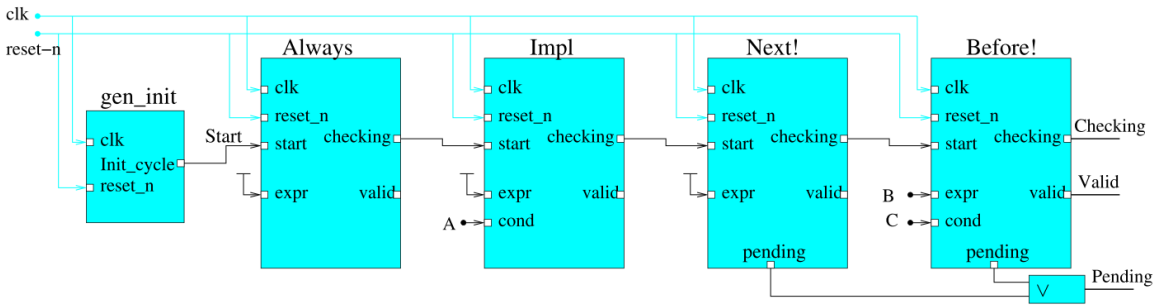


Figure 11.3 A composition monitor for the PSL property `always`(A`->next!`[2](B `before!` C)). (Source [MAB06])

The extension of the construction method to TLM consists in transformation of each primitive primitive monitor into a SystemC class. The main method of the SystemC classes is the *update* function. The function corresponds to the evaluation of the property and it is called each time a notification by an observable subject occurs (Fig. 11.4). An observer wraps the monitor, to activate this method when necessary. [FP10; PF10; Pie+12] define the *ISIS* tool which allows generation of SystemC monitors from PSL specification and enables automatic instrumentation of the verified model with those monitors. The complexity of the generation is proportional to the size of the PSL expression.

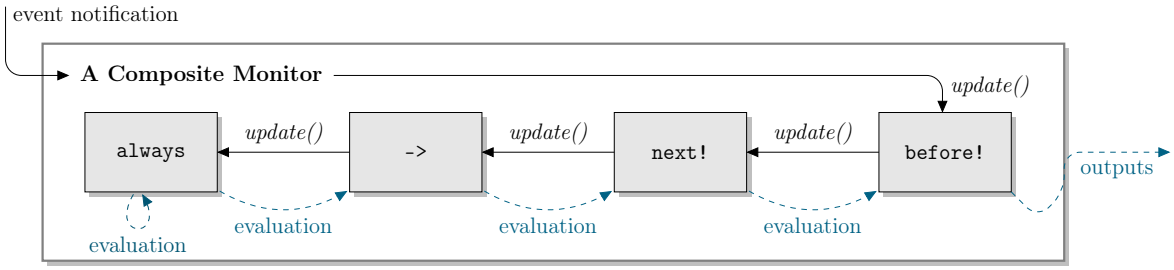


Figure 11.4 Evaluation principle of SystemC primitive monitors of a PSL property `always`(A`->next!`(B `before!` C)). (Source [Fer11])

Belhadj Amor et al. [BA14] extends the work of [FP10; PF10; Pie+12] with monitors for Sequential Extended Regular Expressions (SERE) of PSL. The work is inspired by [MGB07; MAB07b]. The idea is to define a primitive monitor per SERE operator (e.g., the concatenation operator ";", the Kleene star operator "$*$", etc., see Sec. 2.3.2.4) and to connect those primitive monitors according to the syntax tree of the PSL property. Implementation of a monitor of a suffix implication operator $r \mapsto \phi$ is particularly interesting. Here, r is a SERE formula and ϕ can be either LTL or SERE formula of PSL. *Each* detection of r starts evaluation of the *new instance* of the formula ϕ. For ϕ each evaluation instance aims at finding violations of ϕ. Several evaluation instances can be evaluated simultaneously; each evaluation instance is represented by a token propogated from the left-most monitors to the right-most monitors. To implement this mechanism, each primitive monitor of SERE and LTL operators exists in two versions: *monochrome* is used for the left part r, *polychrome* is used for the right part ϕ. In the former case only one token is

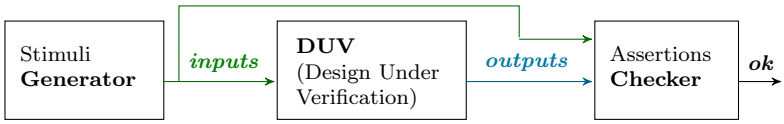


Figure 11.5 The verification framework for hardware designs

propagated at a time. In the latter case several instances of tokens can be processed at the same time, one token per evaluated instance.

11.1.3 Verification Methodologies and Libraries

Various verification methodologies and libraries are actively used for modeling and verification of mixed hardware and software systems. All of them follow the generation/checking testing strategy illustrated in Section 2.1.3 (Fig. 11.5). For system-level testing we complement these approaches with a generalized use of the same ingredients (stimuli generation and assertion checking). As a consequence, the system-level view requires less effort before starting the first testing experiments.

Verification Libraries Verification methodologies are supported with sets of *libraries* which enable implementation of testbenches in different hardware verification languages (HDL) like *e* [IJ04], SystemVerilog [ST12], SystemC [Sys]. In Figure 11.6 colors show in which HDL one can implement testbenches in corresponding verification methodology.

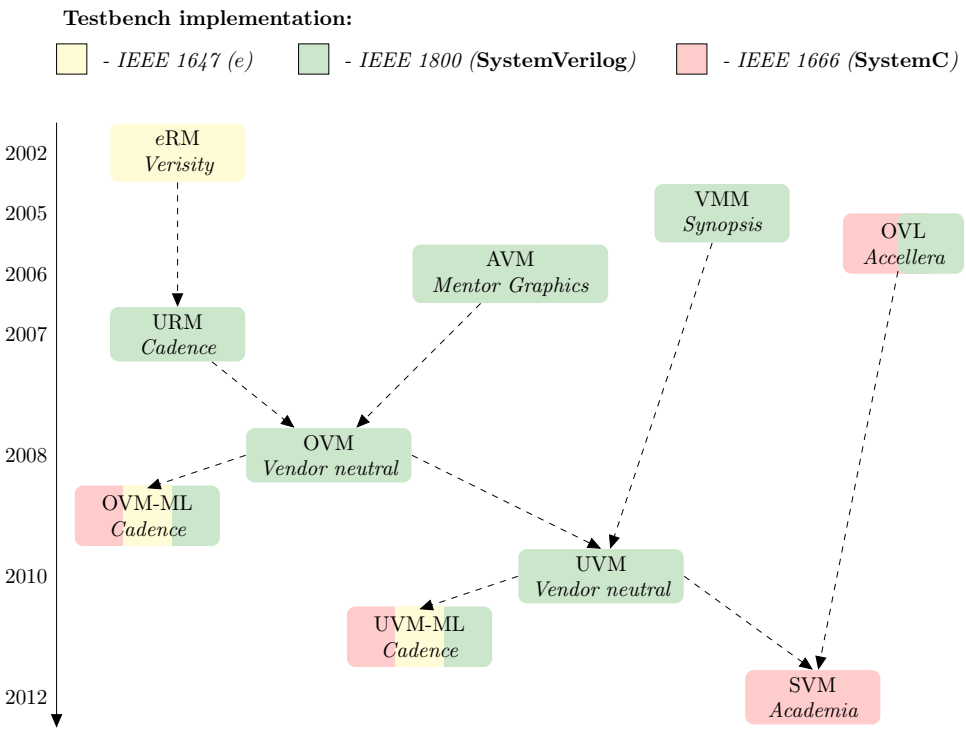


Figure 11.6 Taxonomy of verification methodologies for SoCs design. Arrows show ancestor-descendant relation.

Verification Methodologies Many verification methodologies for complex hardware/software systems like SoCs have been proposed during the past decades (Fig. 11.6). They differ in specific features they use for testbench construction (e.g., description of generated stimuli sequences, handling of DUT outputs, etc.), the way they interact with and sample outputs of DUT, etc.

The *e Reuse Methodology* (*e*RM) [Erm] has been proposed by Vericity Design with the implementation provided in *e* HDL. The *e*RM gave birth to the *Universal Reuse Methodology* (URM) [CDSb] introduced by Cadence with the implementation in SystemVerilog. Independently Synopsis and Mentor Graphics defined the *Verification Methodology Manual* (VMM) [Syn05; Ber+06; Inc16b] and the *Advanced Verification*

Methodology (AVM) [Inc16a] respectively, both in SystemVerilog. AVM was the first open-source verification solution. Cadence and Mentor Graphics joined their efforts to the *Open Verification Methodology* (OVM) [Gla09; Inc05] with a later unification to the *Universal Verification Methodology* (UVM) [RM13; HB15; Vas16] with Synopsys. OVM and UVM are the first vendor neutral verification methodologies tested against many simulators. Originally they have implementation in SystemVerilog. Cadence developed multi-language support for OVM and UVM known as OVM-ML [CDSa] and UVM-ML [CDSc] respectively. Under the multi-language umbrella OVM-ML and UVM-ML enable implementation of subset of respective methodologies in *e*, SystemVerilog and SystemC.

Addionally, the *Open Verification Library* (OVL) [Fos06; AO14a] was developed by Accellera standard. It enables assertion-based verification (ABV) of SoCs and provides the implementation in SystemVerilog and SystemC. The library serves as repository of checking modules which can be instantiated as necessary in test-benches and facilitates specific checking goals. For example, the OVL has a module for checking that a specified expression must not change its value for a specified period of time. The OVM, the UVM and the OVL inspired creation of the *System Verification Methodology* (SVM) [Oli+12] with the set of SystemC libraries [FSO+12] for system level verification at TL level.

Some of the enumerated methodologies are obsolete and not used anymore (the AVM, the VMM). Nowadays the UVM is the widely accepted Accellera standard. To the best of our knowledge the SVM remains to be purely academic work and it is not adopted by the industry yet.

11.2 Functional Verification Through the Design Flow of SoCs

Our stubs of TL components (see Chapter 8) could be integrated in mixed platforms consisting of components defined at different abstraction levels (e.g., TLM, RTL). Our loose-ordering properties could be reused in the design flow.

11.2.1 Transactor-Based Verification: Properties Reuse

Our stubs could be integrated into mixed TL-RTL platforms by means of *transactors*. Transactor-Based Verification[1] (TBV) allows a mixed TL-RTL co-verification [JJ03]. A transactor works as a translator from a TL functional call to an RTL sequence of statements (Fig. 11.7), i.e., it provides the mapping between transaction-level requests, made by TL components, and detailed signal-level protocols on the interface of RTL IPs. A transactor is associated with a pair of interfaces, one at RTL and one at transaction level. Using transactors one can reuse loose-ordering properties for testing hardware design at RTL level.

In [BFF05] transactors are exploited to reuse PSL properties defined for TLM, at the RTL. The effectiveness of reusing testbenches and assertions by TBV, with respect to their manual conversion, has been theoretically proven in [BFP06].

In [KT07] a native assertion mechanism for TBV is defined. The mechanism is inspired by SVA and PSL. It enables construction of assertions which are synchronized with events reported by callbacks.

The effectiveness of TBV can be improved if transactors specific for TLM-RTL communication are automatically generated [BP06; BFD08; BP07; BF06].

11.2.2 Properties Refinement

Bombieri et al. [BFP07] defines incremental assertion-based verification methodology to check the correctness of the TL-to-RTL design refinement. The methodology relies on reusing assertions and already checked code; it is guided by an assertion coverage metrics. The methodology reuses assertions defined at TL to check the functional correctness of the RTL implementation of the DUV by means of transactor (see Sec. 11.2.1).

Pierre et al. In [PA13] PSL assertions expressing temporal requirements on the interactions and communications in the SoC are refined automatically into their RTL counterparts. The work addresses the issue of the modification of *temporal granularity* due to the introduction of actual communication channels in place of abstract components. It proposes the set of transformation rules to refine a communication (atomic transaction) into a *sequence of events* specific to RTL. Some of the defined rules allow definition of specific delays for a communication. They allow to obtain definition of RTL properties stating that some communication action *is expected after a certain number of clock ticks*. The rules translate PSL properties

[1] In some sources Transactor-Based Verification is also referred to as Transaction-Based Verification [Bra+00; KT07].

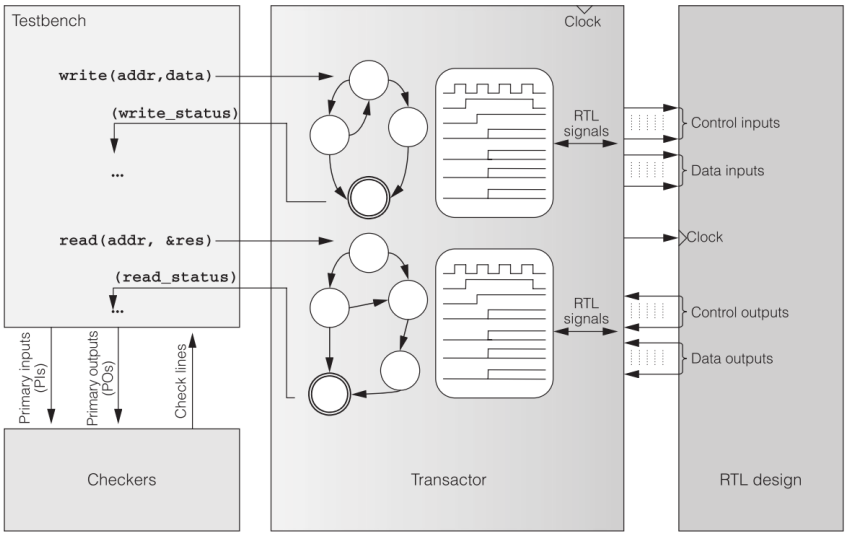


Figure 11.7 The principle of Transactor-Based Verification. (Source [Bom+07])

of TLM with operators like `until!` into sequences of nested `next![x]` operators and implications defining RTL properties. A formula `next![x]`ϕ is a formula which means that ϕ should hold after x instances (e.g., clock ticks). The set of defined rules is independent from any ABV tools.

11.3 Contracts for SystemC/TLM

11.3.1 Contracts for Hardware Designs

The assume/guarantee reasoning have been formalized for various types of software or hardware systems. For instance, [Ben+08] defines a generic composition theory of contracts providing relations of contract satisfaction and refinement; [Bau+12] builds a contract framework on top of any specification theory (e.g., modal specifications [Rac+11], timed specifications [Ber+09]) which supports composition and specification refinement.

Applications of (early) contracts include modular verification [McM99], executable specifications for synchronous languages [MM04], hardware optimization by exploiting *(sequential) don't care* sets [Dev91], etc. A sequential don't care [Dev91] corresponds to input *sequences* of vectors (e.g., 1011, 1011, 1000) that does not appear at finite state machine modeling the chip. Such don't cares are used to reduce the number of states of state machines leading to area and performance optimization of the chip.

11.3.2 Other Uses of Formally-Defined Contracts

When contracts are formally defined for a set of components, a lot of manipulations can be done, apart from solving constraints and producing one or several concrete random executions. For instance Interface Automata [AH01] allow to compute the most liberal environment compatible with a component. More related work can be found in [Bau+12] and [Rac+11] which unifies interface automata and modal specifications.

11.4 Virtual Prototyping of SoCs

11.4.1 Early Detection of Bugs

In [Eng15] the author investigates the application of continuous integration processes (from agile methods) to embedded systems testing. It relies on the intensive usage of virtual prototypes and simulation. The work shows that the approach can significantly hasten development of a system due to bugs detected earlier in the design flow. The traditional testing approach is meant to be used (see Section 11.1.3, Fig. 11.5). In our work we go further. We turn the virtual platform into a test generation/test management system. It allows faster continuous integration at subsystem-, system-, or large system-level testing.

11.4.2 Raising Abstraction Level Beyond TLM

In [Eck+10] the authors propose the modeling styles which allows abstraction beyond the currently applied TLM methodology. It enabled a higher modeling abstraction through merging hardware dependent low level driver software with the HW interface. Thus, sequences of HW transactions can be merged to single HW/SW transactions while preserving both the HW architecture and the low-level to high-level SW interfaces. It allows to reduce the overall number of communication activities. A central resource model is used to ensure that despite the increased level of abstraction it is possible to maintian a high degree of timing accuracy.

Our generalized stubbing can be seen as an attempt to raise the abstraction level of the design beyond TLM. In our work we focus on the definition of hardware components rather than software/hardware interactions.

Chapter 12

Conclusions and Prospects

Contents

12.1 Summary and Contributions **177**
12.1.1 Context of the Work 177
12.1.2 Contributions 178
12.2 Prospects **179**
12.2.1 Integrating Loose-Orderings into Existing Standards 179
12.2.2 Direct Prospects for the Academic World 180
12.2.3 Long Term Prospects 180

12.1 Summary and Contributions

Systems-on-a-Chip (SoCs) are very complex. The complexity comes from the heterogeneous nature of these systems, which include the hardware and software parts. To tackle this complexity, SystemC-based Transaction Level Modeling (TLM) is used. SystemC/TLM allows defining high-level executable component-based models for SoCs, called virtual prototypes. These models are used for the early development of the embedded software, and the validation of the hardware. There are Assertion-Based Verification (ABV) methods, which allow property checking of TL models early in the design flow. Nevertheless, TL models can be *over-constrained*, which means that they do not represent all the malfunctions of the design. Our contributions consist of two orthogonal and complementary parts: On one hand, we identify sources of over-constraints in TL models appearing due to the order of interactions between components, and propose a notion of *loose-ordering* which allows to remove these over-constraints. On the other hand, we propose a *generalized stubbing mechanics* which allows the very early simulation with SystemC/TLM virtual prototypes. Globally, this work contributes to the very early detection and correction of functional faults of SystemC/TLM virtual prototypes.

12.1.1 Context of the Work

A part of this work was about examining and experimenting with SystemC/TLM virtual prototypes of SoCs. The experiments were done on the family of SoCs which have one or several CPUs running the software, one or several hardware accelerators, a memory, inputs/outputs, one or several interconnects. I have designed the intercom system which performs face recognition, and implemented its SystemC/TLM virtual prototype (see Chapter 3). My running example was inspired by industrial case studies; it has non-trivial synchronization protocols between components. By experimenting with the intercom's virtual prototype, I made several interesting observations which were the basis for the contributions of this thesis:

- The first observation is that it can be hard to find (detect) bugs of a SystemC/TLM virtual prototype, when it is simulated without a clear specification of the system in mind. For instance, if the button of the intercom does not respond, is it a bug?
- Second, even if the specification exists, very often it is implicit and not clearly defined. Thus, there are no means that would help to find sources of potential bugs. For example, if the button of the

intercom does not respond, is it the fault of the GPIO, or the embedded software? Or the timer of the intercom? Or all of them?

- Third, when the specification is defined formally (i.e., by means of a specification language like PSL), there should be tools which would allow checking of the properties at simulation time. And again there are several difficulties: although there are many proposals in the academic world about specification languages, many of them are either only abstract ideas (i.e., cannot serve practical needs), or not very expressive, therefore cannot be really used to define properties of interest (see Sec. 11.1.1). There are works providing ABV support for PSL specifications of TL models (see Sec. 11.1.2.2). Nevertheless, the respective tools, even though claimed to be useful and efficient to a certain extent, are not publicly available (see Sec. 11.1.2.2).

- Finally, I examined the "implicit" types of properties kept in mind when defining TL models. I made an observation that most of these properties can be divided into two groups: the properties defining the data expected by components, and the properties defining what the components should do. Moreover, most of the properties of each type follow the same *patterns*, i.e., "*get the configuration data before a computation is started*". At the same time, I observed that the order in which components get data is usually irrelevant. Here I discovered several interesting and unexpected things: even if the components of TL models can get data in any order, sometimes they are defined in such a way that they always provide some outputs in the same order (e.g., when they configure or start other components). It means that at simulation time with such TL models only those behaviors of the hardware are exposed, which correspond to the defined order. I made experiments with my running example by changing the order of some interactions, and observed that some bugs, which were not visible before has appeared. The study of related work has shown that the problem of specification of the order non-determinism was not addressed, i.e., no ABV support was available.

Separately, I was searching for an answer of the fundamental question on abstraction levels: if one wants to start with models above TLM, what kind of information can one accept to be not available yet? And what kind of system properties can one expect to assess on such very abstract models? The answers to these questions are the contributions of this thesis.

12.1.2 Contributions

Identification of Over-Constraints in TL Models I have identified that TL models can be over-constraints due to the order of interactions between components. To capture such over-constraints, I have proposed the notion of *loose-ordering*. By reviewing industrial case studies I have also identified a set of primitive constructs (patterns) to define loose-ordering properties (Chapter 4).

A Tool for Generating Efficient Monitors for SystemC/TLM In this work, I have discovered that with traditional specification languages based on temporal logic it can be hard to define order non-determinism. Therefore, ABV methods provided for such specification are not appropriate for checking loose-ordering properties; these methods either do not reflect the semantics of loose-orderings, or they are not very efficient. I have extended ABV for SystemC/TLM by proposing different types of recognizers for loose-orderings. My recognizers can *report errors* when loose-orderings are violated, or they *continuously* try to detect occurrences of loose-orderings. The recognizers have been implemented in Lustre and extensively tested with a formal testing tool. To integrate the loose-ordering properties into the ABV framework and make them interoperable with the existing industrial standards, I have implemented a library of efficient SystemC monitors for loose-orderings.

I have performed experiments with the intercom system. First, I defined loose-ordering properties for the system's components. Then, I built the SystemC monitors for those properties, and run a simulation with the virtual prototype of the running example coupled with the monitors. With such simulation I have not only detected all the synchronization bugs of the design, but also was able to find the bugs' sources, which was problematic otherwise, specially if a bug had several causes.

A Generalized Stubbing Mechanics with Contracts for Early Simulation with Virtual Prototypes of SoCs My proposal allows specifying SoC components which are not entirely determined on the values of the exchanged data, the order of the interactions and/or the timing. It is built by gathering ideas from testing approaches for hardware designs, testing approaches for real-time reactive systems, and design-by-contract for hardware or software designs. I propose to define a stub of a component as a

contract: an *assume* clause specifies what the component expects from its environment, and a *guarantee* clause defines the obligation of the component, if it is placed in a proper environment. I propose to define the guarantee clause in a form of an implication $\pi \Rightarrow \sigma$: when π occurs, σ should occur within a bound. Both assume and guarantee clauses specify *sequences* of inputs and/or outputs which are supposed to be respectively received and produced by the component. The main ideas behind the generalized stubbing mechanics is that we can start with very abstract description of components, provided we have the information on how they interact and create conflicts on shared resources, possibly with cyclic dependencies (a component has some impact on the rest of the system, which itself impacts the component). The *contract-like* specification style is very adequate to capture this type of dependencies. I want the stubbing framework to be as general as possible, and at the same time implementable and applicable to real, industrial case studies. To achieve these two objectives, I make a compromise between the expressivity of contracts and the implementability of stubs, and accept to put constraints on property languages of assume and guarantee clauses. This leads us to several contributions:

- *Identification of limitations on property languages and syntactic constraints to make stubbing implementable.* To ensure the efficiency of the implementation, I require that the property language of the left-hand part π of a guarantee clause $\pi \Rightarrow \sigma$ has a deterministic continuous recognizer. When there are several guarantee clauses $(\pi_i \Rightarrow \sigma_i)$s, I also require that the left-hand parts σ_is do not share names, and that the guarantee clauses do not form cycles (i.e., do not depend on the names of each other).
- *Identification of the type of scheduling required to simulate stub-based systems.* A part from that, I have identified the types of scheduling problems that should be addressed in order to run simulations with stubs. In particular, we need to ensure that bounds of all guarantee clauses $(\pi_i \Rightarrow \sigma_i)$s are preserved (this can be done by counting events), and that the stubs are activated fairly.
- *Operational semantics of a stub and a system made of stubs.* The operational semantics shows that the system made of stubs have several sources of non-determinism (e.g., the non-deterministic production of outputs). This is where scheduling should be applied.

Additionally, when working on stubbing, I have discovered that the implication operator $\alpha \mapsto \beta$, which inspired our guarantee clause and which is provided by specification languages like PSL, does not have the well-defined semantics. The semantics of the operator can depend on the vendors implementing the property checking tools, and it is often hidden from the users of those tools. Here I have contributed to the specification of digital circuits proposing the following:

- *Clarification of the semantics of properties of the implication kind.* I have made several semantic choices for the implication operator $\alpha \mapsto \beta$. Thus, I propose the semantics of the overlapping of several occurrences of α, and the overlapping of occurrences of α and β. Moreover, my limitations on the property languages and syntactic constraints on guarantee clauses (see above) enable online exploitation of $\alpha \mapsto \beta$ in both cases; as a part of ABV, and as a part of stubbing.

A Proof of Concept: A SystemC Implementation of the Generalized Stubbing Mechanics As a proof of concept I have implemented the generalized stubbing mechanics with the SystemC scheduler. The implementation allows to build a virtual prototype for systems made of stubs only, and to integrate stubs with already existing SystemC/TLM components. This has been demonstrated on my running example. The experiments showed that (i) less effort and time can be required to build virtual prototypes with integrated stubs, (ii) stubs have the capability to detect bugs and blame faulty components. Moreover, the simulation with stubs can be faster, since the implementation details (e.g., the implementation of computation algorithms) are omitted.

12.2 Prospects

This work opens several future prospects, which can be classified depending on the required amount of further research, and the applicability in the industry or in the academic world.

12.2.1 Integrating Loose-Orderings into Existing Standards

For comparison purposes, I have encoded a subset of loose-orderings into PSL, and observed that the encoding of order non-determinism into PSL is either *non-trivial* or *explosive*. Thus, the encoding into

LTL fragment of PSL is very hard, because the semantics of the conjunction and nesting of LTL operators is not intuitive. Without special tools (e.g., SPOT) which would visualize the semantics of LTL formulas (e.g., in the form of automata), it is very hard to assess the correctness of the obtained encodings. At the same time the most intuitive and naive encoding of loose-ordering into Extended Sequential Regular Expression (a fragment of PSL), consisting in the explicit enumeration of all total orders, provokes the combinatorial explosion of the length of the obtained formula. I did not prove that my proposal is the best possible way to encode the order non-determinism. Nevertheless, I think that having dedicated constructs in specification languages like PSL is useful, and loose-orderings can be good candidates for that. It would be also interesting to compare the expressive powers of PSL and the loose-ordering language.

12.2.2 Direct Prospects for the Academic World

Specifying Stubs This work provides recipes which enable early simulation with virtual prototypes for SoCs. One of the recipes is the set of constraints on the property languages defining stubs. One of the constraints is that the languages should have deterministic continuous recognizers. Although this work presents continuous recognizers for regular languages (see Sec. 2.4), systematic work should be done to identify types of languages which have this property, and therefore can be plugged in our stubbing framework.

Solving the Coverage Problem The generalized stubbing mechanics has several sources of non-determinism: at the system level there is a non-deterministic choice between stubs to be activated; at the stub level, there are choices between inputs and outputs, as well as between different outputs of the stub. All these choices can be controlled by a scheduler. Different scheduling policies allow observing subsets of runs of stub-based systems. This leads to a coverage problem which is not directly addressed in this work. It would be interesting to examine the influence of different scheduling algorithms on the subset of observed runs, and to find a method which allows to improve the coverage of the system.

12.2.3 Long Term Prospects

The long term prospect of this work is the consideration of both functional and non-functional (e.g., energy consumption) properties of systems-on-a-chip. The theoretical results and implementations of this thesis are ready to integrate the results obtained by STMicroelectronics and DOCEA Power (now Intel) in the framework of HELP project[1] on the modeling of energy consumption and temperature in high-level models of systems-on-a-chip. Such integration will make my results compatible with the recent advances in TLM power modeling. Moreover, the work on loose-ordering and relaxing constraints in models has helped define a new promising research direction which aims to answer the question: how to define and simulate models with *loose-power* information. This problem is already being addressed at Verimag.

[1] an ANR Arpege project http://www-verimag.imag.fr/PROJECTS/SYNCHRONE/HELP/

Appendix A

LUSTRE Implementation of Loose-Orderings

```
node range_error_non_shuffled(u,v: int; s, start, n, C, Ac, Af, B: bool)
returns (err, ok, nok: bool);
var cpt: int; s0, s1, s2, s3, s4, s5: bool;
let
-- ensuring that names do not occur simultaneously
assert(#(B, Ac));  assert(#(B, Af)); assert(#(B, C));  assert(#(B, n));
assert(#(Ac, Af)); assert(#(Ac, C)); assert(#(Ac, n)); assert(#(Af, C));
assert(#(Af, n));  assert(#(C,  n));

-- encoding transitions
s0=true  -> pre(s0 and not(start))    or pre(s2 and Ac and not(s)) or
            pre(s3 and Ac and cpt>=u) or pre(s4 and Ac);
s1=false -> pre(s0 and start and not(n or C)) or
            pre(s1 and not(n or C or Ac or Af or B));

s2=false -> pre(s0 and start and C) or pre(s1 and C) or
            pre(s2 and not (n or Ac or Af or B));
s3=false -> pre(s0 and start and n) or pre((s1 or s2) and n) or
            pre(s3 and not(C or Ac or Af or B or (n and cpt=v)));

s4=false -> pre(s3 and C and cpt>=u) or
            pre(s4 and not(n or Ac or Af or B));
s5=false -> pre((s1 or s2 or s3 or s4) and (Af or B)) or
            pre((s1 or (s2 and s)) and Ac) or
            pre(s3 and (Ac or C) and cpt<u) or pre(s3 and n and cpt=v) or
            pre(s4 and n) or pre(s5);

-- the counter
cpt=0 -> if pre(s0 and start and not(n)) then 0 else
         if pre((s0 or s1 or s2) and n) then 1 else
         if pre(s3 and n and cpt<v) then pre(cpt)+1 else pre(cpt);

-- the outputs
ok  = (s3 and Ac and (cpt>=u and cpt<=v)) or (s4 and Ac);
nok = (s2 and Ac and not(s));
err = ((s1 or s2 or s3 or s4) and (Af or B)) or
      ((s1 or (s2 and s)) and Ac) or (s3 and (Ac or C) and cpt<u) or
      (s3 and n and cpt=v) or (s4 and n) or s5;
tel
```

Figure A.1 The LUSTRE implementation of the error reporting elementary recognizer $\mathbb{R}^{error}_{non\text{-}shuffled}$ of ranges appearing in a fragment with the non-shuffled semantics ($\sqcup\!\sqcup$ = *Non-Shuffled*).

```
node range_error_shuffled(u,v: int; s, start, n, C, Ac, Af, B: bool)
returns (err, ok, nok: bool);
var cpt: int; s0, s1, s2, s3, s4: bool;
let
-- ensuring that names do not occur simultaneously
assert(#(B, Ac));  assert(#(B, Af)); assert(#(B, C));  assert(#(B, n));
assert(#(Ac, Af)); assert(#(Ac, C)); assert(#(Ac, n)); assert(#(Af, C));
assert(#(Af, n));  assert(#(C,  n));

-- encoding transitions
s0=true -> pre(s0 and not(start)) or
             pre(s2 and Ac and not(s)) or
             pre(s3 and Ac and cpt>=u);

s1=false -> pre(s0 and start and not(n or C)) or
            pre(s1 and not(n or C or Ac or Af or B));

s2=false -> pre(s0 and start and C) or
            pre(s1 and C) or
            pre(s2 and not (n or Ac or Af or B));

s3=false -> pre(s0 and start and n) or
            pre((s1 or s2) and n) or
            pre(s3 and not(Ac or Af or B or (n and cpt=v)));

s4=false -> pre((s1 or s2 or s3) and (Af or B)) or
            pre((s1 or (s2 and s)) and Ac) or
            pre(s3 and Ac and cpt<u) or
            pre(s3 and n and cpt=v) or
            pre(s4);

-- the counter
cpt=0 -> if pre(s0 and start and not(n)) then 0 else
         if pre((s0 or s1 or s2) and n) then 1 else
         if pre(s3 and n and cpt<v) then pre(cpt)+1 else
           pre(cpt);

-- the outputs
ok  = (s3 and Ac and (cpt>=u and cpt<=v));
nok = (s2 and Ac and not(s));
err = ((s1 or s2 or s3) and (Af or B)) or
      ((s1 or (s2 and s)) and Ac) or
      (s3 and Ac and cpt<u) or (s3 and n and cpt=v) or s4;
tel
```

Figure A.2 The LUSTRE implementation of the error reporting elementary recognizer $\mathbb{R}^{error}_{shuffled}$ of ranges appearing in a fragment with the shuffled semantics ($\sqcup\!\sqcup$ = *Shuffled*).

```
node range_stop_non_shuffled(u, v: int;
                             s, start, n, C, Ac, Af, B, stopC: bool)
returns (nok, ok, stop: bool);
var cpt: int; s0, s1, s2, s3, s4: bool;
let
-- ensuring that names do not occur simultaneously
assert(#(B, Ac));  assert(#(B, Af)); assert(#(B, C));  assert(#(B, n));
assert(#(Ac, Af)); assert(#(Ac, C)); assert(#(Ac, n)); assert(#(Af, C));
assert(#(Af, n));  assert(#(C,  n));

-- encoding transitions
s0=true -> pre(s0 and not(start)) or
           pre((s1 or s2 or s3 or s4) and (Ac or Af or B)) or
           pre((s2 or s4) and C and stopC) or
           pre(s3 and (cpt<u and C or cpt>=u and C and stopC)) or
           pre(cpt=v and n) or pre(s4 and n);

s1=false-> pre(s0 and start and not(n or C)) or
           pre(s1 and not(n or Ac or Af or B or C));

s2=false-> pre(s0 and start and C) or pre(s1 and C) or
           pre(s2 and not(n or Ac or Af or B or C and stopC));

s3=false-> pre(s0 and start and n) or pre((s1 or s2) and n) or
           pre(s3 and not(C or Ac or Af or B or n and cpt=v));

s4=false-> pre(s3 and cpt >=u and C and not(stopC)) or
           pre(s4 and not(n or Ac or Af or B or C and stopC));

-- the counter
cpt=0-> if pre(s0 and start and not(n)) then 0 else
        if pre((s0 or s1 or s2) and n) then 1 else
        if pre(s3 and n and cpt<v) then pre(cpt)+1 else
          pre(cpt);

-- the outputs
ok=(s3 and cpt>=u and (Ac or Af or B or C and stopC)) or
   (s3 and cpt=v and n) or (s4 and (Af or Ac or B or n or C and stopC));
nok =(s2 and not(s) and (Ac or Af or B or C and stopC));
stop=(s3 and cpt=v and n) or (s3 and cpt<u and C);
tel
```

Figure A.3 The LUSTRE implementation of the stopping elementary recognizer $\mathbb{R}^{stop}_{non\text{-}shuffled}$ of ranges appearing in a fragment with the non-shuffled semantics ($\sqcup\!\sqcup$ = *Non-Shuffled*).

```
node range_stop_shuffled(u,v: int; s, start, n, C, Ac, Af, B, stopC:bool)
returns (nok, ok, stop: bool);
var cpt: int; s0, s1, s2, s3: bool;
let
-- ensuring that names do not occur simultaneously
assert(#(B, Ac));  assert(#(B, Af)); assert(#(B, C));  assert(#(B, n));
assert(#(Ac, Af)); assert(#(Ac, C)); assert(#(Ac, n)); assert(#(Af, C));
assert(#(Af, n));  assert(#(C,  n));

-- encoding transitions
s0 = true -> pre(s0 and not(start)) or
             pre((s1 or s2 or s3) and (Ac or Af or B)) or
             pre(s2 and C and stopC) or
             pre(s3 and (cpt=v and n or C and stopC));

s1 = false-> pre(s0 and start and not(n or C)) or
             pre(s1 and not(n or Ac or Af or B or C));

s2 = false-> pre(s0 and start and C) or pre(s1 and C) or
             pre(s2 and not(n or Ac or Af or B or C and stopC));

s3 = false-> pre(s0 and start and n) or pre((s1 or s2) and n) or
             pre(s3 and not(C and stopC or Ac or Af or B or n and cpt=v));

-- the counter
cpt= 0-> if pre(s0 and start and not(n)) then 0 else
         if pre((s0 or s1 or s2) and n) then 1 else
         if pre(s3 and n and cpt<v) then pre(cpt)+1 else
           pre(cpt);

-- the outputs
ok  = (s3 and cpt>=u and (Ac or Af or B or C and stopC)) or
      (s3 and cpt=v and n);
nok = (s2 and not(s) and (Ac or Af or B or C and stopC));
stop = (s3 and cpt=v and n);
tel
```

Figure A.4 The LUSTRE implementation of the stopping elementary recognizer $\mathbb{R}^{stop}_{shuffled}$ of ranges appearing in a fragment with the shuffled semantics ($\sqcup\!\sqcup = \mathit{Shuffled}$).

```
node range_reinit_non_shuffled(u, v: int;
                               s, start, n, C, Ac, Af, B, rinitC: bool)
returns (nok, ok, rinit: bool);
var cpt: int; s0, s1, s2, s3, s4: bool;
let
-- ensuring that names do not occur simultaneously
assert(#(B, Ac));  assert(#(B, Af)); assert(#(B, C));  assert(#(B, n));
assert(#(Ac, Af)); assert(#(Ac, C)); assert(#(Ac, n)); assert(#(Af, C));
assert(#(Af, n));  assert(#(C,  n));

-- encoding transitions
s0 = true -> pre(s0 and not(start)) or
             pre((s1 or s2 or s3 or s4) and (Ac or Af or B)) or
             pre((s2 or s4) and C and rinitC) or
             pre(s3 and (cpt<u and C or cpt=v and n)) or
             pre(s3 and cpt>=u and C and rinitC) or
             pre(s4 and n);

s1 = false-> pre(s0 and start and not(n or C)) or
             pre(s1 and not(n or C or Ac or Af or B));

s2 = false-> pre(s0 and start and C) or
             pre(s1 and C) or
             pre(s2 and not(n or Ac or Af or B or C and rinitC));

s3 = false-> pre(s0 and start and n) or
             pre((s1 or s2) and n) or
             pre(s3 and not(C or Ac or Af or B or n and cpt=v));

s4 = false-> pre(s3 and cpt >=u and C and not(rinitC)) or
             pre(s4 and not(n or Ac or Af or B or C and rinitC));

-- the counter
cpt= 0-> if pre(s0 and start and not(n)) then 0 else
         if pre((s0 or s1 or s2) and n) then 1 else
         if pre(s3 and n and cpt<v) then pre(cpt)+1 else
           pre(cpt);

-- the outputs
ok  = (s3 and Ac and (cpt>=u and cpt<=v)) or (s4 and Ac);
nok = (s2 and Ac and not(s));
rinit = (s3 and cpt=v and n) or (s3 and cpt<u and C);
tel
```

Figure A.5 The LUSTRE implementation of the re-initializing elementary recognizer $\mathbb{R}^{re\text{-}init}_{non\text{-}shuffled}$ of ranges appearing in a fragment with the non-shuffled semantics ($\sqcup\!\sqcup$ = *Non-Shuffled*).

```
node range_reinit_shuffled(u,v:int; s, start, n, C, Ac, Af, B, rinitC:bool)
returns (nok, ok, rinit: bool);
var cpt: int; s0, s1, s2, s3: bool;
let
-- ensuring that names do not occur simultaneously
assert(#(B, Ac));  assert(#(B, Af)); assert(#(B, C));  assert(#(B, n));
assert(#(Ac, Af)); assert(#(Ac, C)); assert(#(Ac, n)); assert(#(Af, C));
assert(#(Af, n));  assert(#(C,  n));

-- encoding transitions
s0 = true -> pre(s0 and not(start)) or
             pre((s1 or s2 or s3) and (Ac or Af or B)) or
             pre((s2 or s3) and C and rinitC) or
             pre(s3 and n and cpt=v);

s1 = false-> pre(s0 and start and not(n or C)) or
             pre(s1 and not(n or C or Ac or Af or B));

s2 = false-> pre(s0 and start and C) or
             pre(s1 and C) or
             pre(s2 and not(n or Ac or Af or B or C and rinitC));

s3 = false-> pre(s0 and start and n) or
             pre((s1 or s2) and n) or
             pre(s3 and not(C and rinitC or Ac or Af or B or n and cpt=v));

-- the counter
cpt= 0-> if pre(s0 and start and not(n)) then 0 else
         if pre((s0 or s1 or s2) and n) then 1 else
         if pre(s3 and n and cpt<v) then pre(cpt)+1 else
           pre(cpt);

-- the outputs
ok  = (s3 and Ac and (cpt>=u and cpt<=v));
nok = (s2 and Ac and not(s));
rinit = (s3 and cpt=v and n);
tel
```

Figure A.6 The LUSTRE implementation of the re-initializing elementary recognizer $\mathbb{R}^{re\text{-}init}_{shuffled}$ of ranges appearing in a fragment with the shuffled semantics ($\sqcup\!\sqcup$ = *Shuffled*).

Appendix B

Encoding of Loose-Orderings into PSL

B.1 Encoding of an Antecedent Requirement into LTL

B.1.1 Encoding of $\big(a < b < c \ll i \mid \nabla\big)$ into LTL

Consider the configuration of an antecedent requirement $\big(a < b < c \ll i \mid \nabla\big)$. Its LTL encoding consists of the following components:

$$\Box\big(\neg(a \wedge b)\big) \wedge \Box\big(\neg(a \wedge c)\big) \wedge \Box\big(\neg(a \wedge i)\big) \wedge \Box\big(\neg(b \wedge c)\big) \wedge \Box\big(\neg(b \wedge i)\big) \wedge \Box\big(\neg(c \wedge i)\big) \tag{$\mathcal{A}1$-Exclusiveness}$$

$$\Box\big(a \rightarrow \bigcirc(\neg a \,\mathcal{U}\, i)\big) \wedge \Box\big(b \rightarrow \bigcirc(\neg b \,\mathcal{U}\, i)\big) \wedge \Box\big(c \rightarrow \bigcirc(\neg c \,\mathcal{U}\, i)\big) \tag{$\mathcal{A}1$-MaxOne}$$

$$\Big(\big(a \rightarrow \bigcirc(\neg a \,\mathcal{U}\, i)\big) \,\mathcal{U}\, i\Big) \wedge \Big(\big(b \rightarrow \bigcirc(\neg b \,\mathcal{U}\, i)\big) \,\mathcal{U}\, i\Big) \wedge \Big(\big(c \rightarrow \bigcirc(\neg c \,\mathcal{U}\, i)\big) \,\mathcal{U}\, i\Big) \tag{$\mathcal{A}1$-MaxOneNR}$$

$$\Box\big(b \rightarrow (\neg a \,\mathcal{U}\, i)\big) \wedge \Box\big(c \rightarrow (\neg b \,\mathcal{U}\, i)\big) \tag{$\mathcal{A}1$-Order}$$

$$\Big(\big(b \rightarrow \bigcirc(\neg a \,\mathcal{U}\, i)\big) \,\mathcal{U}\, i\Big) \wedge \Big(\big(c \rightarrow \bigcirc(\neg b \,\mathcal{U}\, i)\big) \,\mathcal{U}\, i\Big) \tag{$\mathcal{A}1$-OrderNR}$$

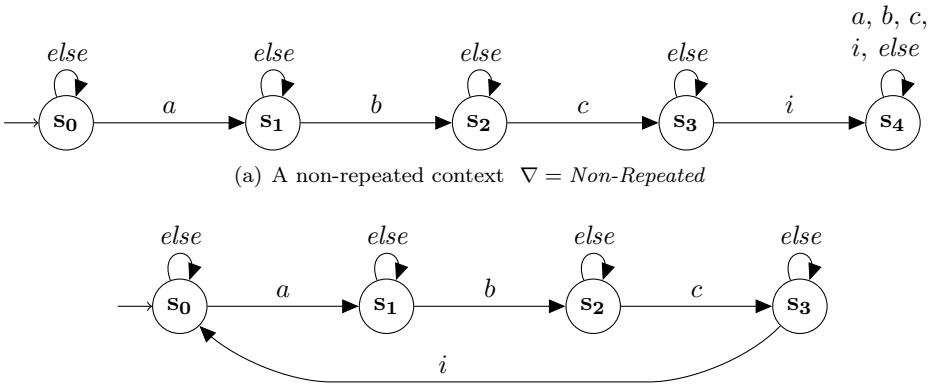
$$\big(\neg i \,\mathcal{U}\, a\big) \wedge \big(\neg i \,\mathcal{U}\, b\big) \wedge \big(\neg i \,\mathcal{U}\, c\big) \tag{$\mathcal{A}1$-First$\mathcal{F}'$}$$

$$\Box\big(i \rightarrow \bigcirc(\neg i \,\mathcal{U}\, a)\big) \wedge \Box\big(i \rightarrow \bigcirc(\neg i \,\mathcal{U}\, b)\big) \wedge \Box\big(i \rightarrow \bigcirc(\neg i \,\mathcal{U}\, c)\big) \tag{$\mathcal{A}1$-After$\mathcal{F}'$}$$

The LTL encoding of $\big(a < b < c \ll i \mid \textit{Non-Repeated}\big)$ with a non-repeated context is Conjunction B.1. The LTL encoding of the antecedent requirement $\big(a < b < c \ll i \mid \textit{Repeated}\big)$ with a repeated context is Conjunction B.2. Figures B.1(a) and B.1(b) show the monitors provided by the SPOT tool for Conjunctions B.1 and B.2 respectively. Figure B.2 provides the LTL encoding of the property with a repeated context in SPOT syntax.

$$\mathcal{A}1\text{-Exclusiveness} \wedge \mathcal{A}1\text{-MaxOneNR} \wedge \mathcal{A}1\text{-OrderNR} \wedge \mathcal{A}1\text{-First}\mathcal{F}' \tag{B.1}$$

$$\mathcal{A}1\text{-Exclusiveness} \wedge \mathcal{A}1\text{-MaxOne} \wedge \mathcal{A}1\text{-Order} \wedge \mathcal{A}1\text{-First}\mathcal{F}' \wedge \mathcal{A}1\text{-After}\mathcal{F}' \tag{B.2}$$



(a) A non-repeated context $\nabla = \textit{Non-Repeated}$

(b) A repeated context $\nabla = \textit{Repeated}$

Figure B.1 The SPOT monitor of the LTL encoding of $(a < b < c \ll i \mid \nabla)$. *else* stands for any name which is not of $\{a, b, c, i\}$. Transitions which are not defined are forbidden.

```
~ A1-Exclusiveness
G(!(a && b)) && G(!(a && c))&& G(!(a && i)) &&
G(!(b && c)) && G(!(b && i)) && G(!(c && i)) &&

~ A1-MaxOne
G(a-> X(!a U i)) && G(b-> X(!b U i)) && G(c-> X(!c U i)) &&

~ A1-Order
G(b-> (!a U i)) && G(c->(!b U i)) &&

~ A1-FirstF'
(!i U a) && (!i U b) && (!i U c) &&

~ A1-AfterF'
G(i -> X(!i U a)) && G(i -> X(!i U b)) && G(i -> X(!i U c))
```

Figure B.2 The LTL encoding of $(a < b < c \ll i \mid \textit{Repeated})$ in SPOT syntax.

B.1.2 Encoding of $\Big(\big(\{a,b,c,d\},\wedge\big) \ll i \mid \nabla\Big)$ into LTL

Consider the configuration $\Big(\big(\{a,b,c,d\},\wedge\big) \ll i \mid \nabla\Big)$ of an antecedent requirement. Its LTL encoding consists of the following components:

$$\Box\big(\neg(a\wedge b)\big) \wedge \Box\big(\neg(a\wedge c)\big) \wedge \Box\big(\neg(a\wedge d)\big) \wedge \Box\big(\neg(a\wedge i)\big) \wedge \Box\big(\neg(b\wedge c)\big) \wedge \Box\big(\neg(b\wedge d)\big) \wedge \Box\big(\neg(b\wedge i)\big) \wedge \Box\big(\neg(c\wedge d)\big) \wedge \Box\big(\neg(c\wedge i)\big) \wedge \Box\big(\neg(d\wedge i)\big) \tag{$\mathcal{A}2$-Exclusiveness}$$

$$\Box\big(a \rightarrow \bigcirc(\neg a\ \mathcal{U}\ i)\big) \wedge \Box\big(b \rightarrow \bigcirc(\neg b\ \mathcal{U}\ i)\big) \wedge \Box\big(c \rightarrow \bigcirc(\neg c\ \mathcal{U}\ i)\big) \wedge \Box\big(d \rightarrow \bigcirc(\neg c\ \mathcal{U}\ i)\big) \tag{$\mathcal{A}2$-MaxOne}$$

$$\Big(\big(a \rightarrow \bigcirc(\neg a\ \mathcal{U}\ i)\big)\ \mathcal{U}\ i\Big) \wedge \Big(\big(b \rightarrow \bigcirc(\neg b\ \mathcal{U}\ i)\big)\ \mathcal{U}\ i\Big) \wedge \Big(\big(c \rightarrow \bigcirc(\neg c\ \mathcal{U}\ i)\big)\ \mathcal{U}\ i\Big) \wedge \Big(\big(d \rightarrow \bigcirc(\neg d\ \mathcal{U}\ i)\big)\ \mathcal{U}\ i\Big) \tag{$\mathcal{A}2$-MaxOneNR}$$

$$(\neg i\ \mathcal{U}\ a) \wedge (\neg i\ \mathcal{U}\ b) \wedge (\neg i\ \mathcal{U}\ c) \wedge (\neg i\ \mathcal{U}\ d) \tag{$\mathcal{A}2$-First$\mathcal{F}'$}$$

$$\Box\big(i \rightarrow \bigcirc(\neg i\ \mathcal{U}\ a)\big) \wedge \Box\big(i \rightarrow \bigcirc(\neg i\ \mathcal{U}\ b)\big) \wedge \Box\big(i \rightarrow \bigcirc(\neg i\ \mathcal{U}\ c)\big) \wedge \Box\big(i \rightarrow \bigcirc(\neg i\ \mathcal{U}\ d)\big) \tag{$\mathcal{A}2$-After$\mathcal{F}'$}$$

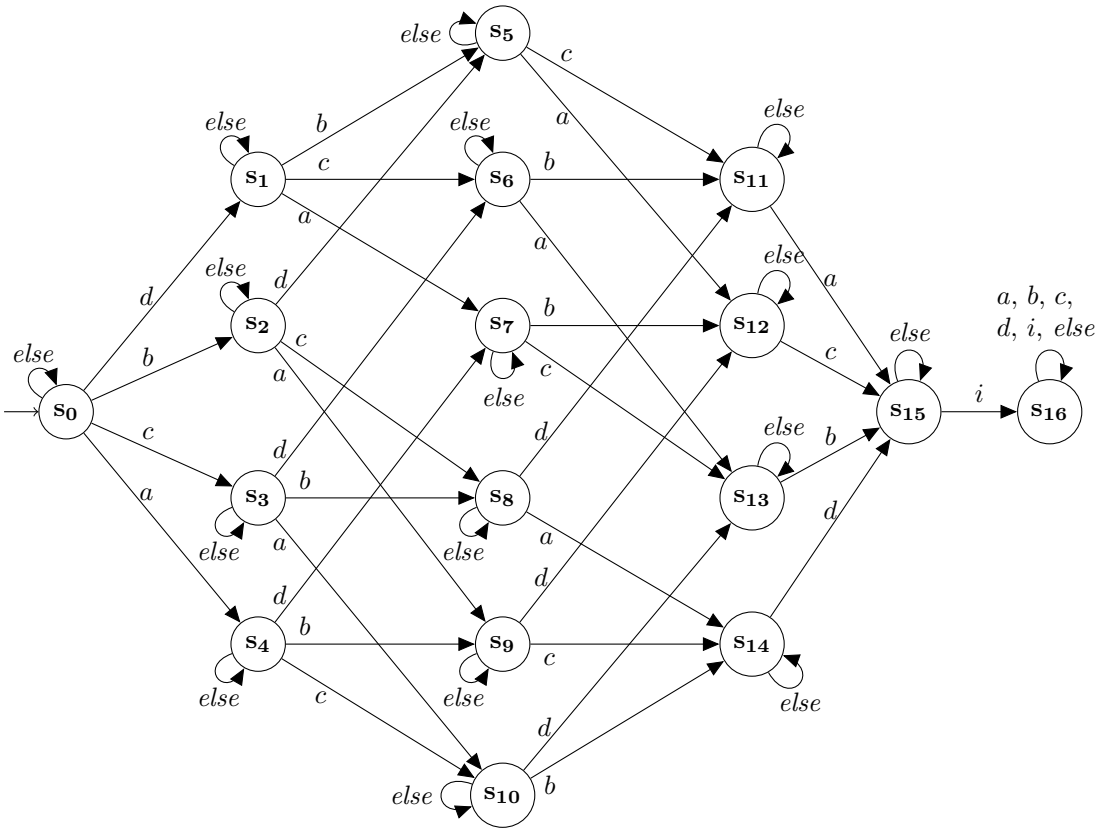
The encoding of $\Big(\big(\{a,b,c,d\},\wedge\big) \ll i \mid \textit{Non-Repeated}\Big)$ with a non-repeated context by means of temporal operators is defined by Conjunction B.3. The LTL encoding of the antecedent requirement $\Big(\big(\{a,b,c,d\},\wedge\big) \ll i \mid \textit{Repeated}\Big)$ with a repeated context into LTL is Conjunction B.4. Figures B.4(a) and B.4(b) illustrate the SPOT monitors corresponding to Conjunctions B.3 and B.4 respectively. Figure B.3 provides the LTL encoding of the property with a repeated context in SPOT syntax.

$$\mathcal{A}2\text{-Exclusiveness} \wedge \mathcal{A}2\text{-MaxOneNR} \wedge \mathcal{A}2\text{-First}\mathcal{F}' \tag{B.3}$$

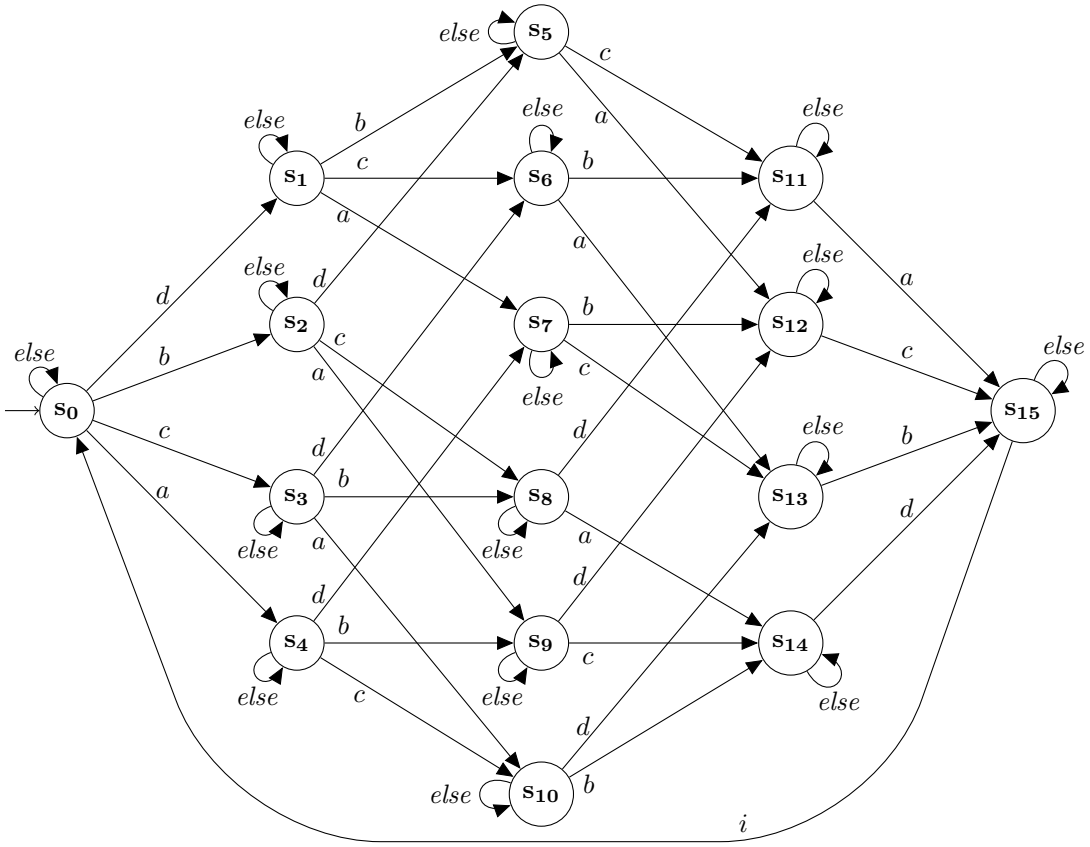
$$\mathcal{A}2\text{-Exclusiveness} \wedge \mathcal{A}2\text{-MaxOne} \wedge \mathcal{A}2\text{-First}\mathcal{F}' \wedge \mathcal{A}2\text{-After}\mathcal{F}' \tag{B.4}$$

```
~ A2-Exclusiveness
G(!(a && b)) && G(!(a && c)) && G(!(a && d)) && G(!(a && i)) &&
G(!(b && c)) && G(!(b && d)) && G(!(b && i)) &&
G(!(c && d)) && G(!(c && i)) && G(!(d && i)) &&

~ A2-MaxOne
G(a-> X(!a U i)) && G(b-> X(!b U i)) && G(c-> X(!c U i)) &&
G(d-> X(!c U i)) &&

~ A2-FirstF'
(!i U a) && (!i U b) && (!i U c) && (!i U d) &&

~ A2-AfterF'
G(i -> X(!i U a)) && G(i -> X(!i U b)) && G(i -> X(!i U c)) &&
G(i -> X(!i U d))
```

Figure B.3 The LTL encoding of $\big(\big(\{a,b,c,d\},\wedge\big) \ll i \mid \textit{Repeated}\big)$ in SPOT syntax.



(a) A non-repeated context $\nabla = \textit{Non-Repeated}$



(b) A repeated context $\nabla = \textit{Repeated}$

Figure B.4 The SPOT monitor of the LTL encoding of $\big((\{a,b,c,d\},\wedge) \ll i \mid \nabla\big)$. *else* stands for any name which is not of $\{a,b,c,d,i\}$. Transitions which are not defined are forbidden.

B.1.3 Encoding of $\Big(\big(\{a,b\},\wedge\big) < \big(\{c,d^{[1,2]}\},\vee\big) \ll i \mid \nabla\Big)$ into LTL

Consider the antecedent requirement $\Big(\big(\{a,b\},\wedge\big) < \big(\{c,d^{[1,2]}\},\vee\big) \ll i \mid \nabla\Big)$. After the removal of the range $d^{[1,2]}$, the new vocabulary of the property is the set $\{a,b,c,d1,d2,i\}$ (see Chapter 4, Sec. 4.5.3). The LTL encoding of the property consists of the following components:

$$\begin{aligned}&\Box\big(\neg(a\wedge d1)\big)\wedge\Box\big(\neg(a\wedge d2)\big)\wedge\Box\big(\neg(a\wedge c)\big)\wedge\Box\big(\neg(a\wedge b)\big)\\&\wedge\Box\big(\neg(a\wedge i)\big)\wedge\Box\big(\neg(d1\wedge d2)\big)\wedge\Box\big(\neg(d1\wedge c)\big)\wedge\Box\big(\neg(d1\wedge b)\big)\\&\wedge\Box\big(\neg(d1\wedge i)\big)\wedge\Box\big(\neg(d2\wedge c)\big)\wedge\Box\big(\neg(d2\wedge b)\big)\wedge\Box\big(\neg(d2\wedge i)\big)\\&\wedge\Box\big(\neg(c\wedge b)\big)\wedge\Box\big(\neg(c\wedge i)\big)\wedge\Box\big(\neg(b\wedge i)\big)\end{aligned}\tag{$\mathcal{A}4$-Exclusiveness}$$

$$\begin{aligned}&\Box\big(a\rightarrow\bigcirc(\neg a\,\mathcal{U}\,i)\big)\wedge\Box\big(b\rightarrow\bigcirc(\neg b\,\mathcal{U}\,i)\big)\\&\wedge\Box\big(c\rightarrow\bigcirc(\neg c\,\mathcal{U}\,i)\big)\wedge\Box\big(d1\rightarrow\bigcirc(\neg d1\,\mathcal{U}\,i)\big)\wedge\Box\big(d2\rightarrow\bigcirc(\neg d2\,\mathcal{U}\,i)\big)\end{aligned}\tag{$\mathcal{A}4$-MaxOne}$$

$$\begin{aligned}&\Big(\big(a\rightarrow\bigcirc(\neg a\,\mathcal{U}\,i)\big)\,\mathcal{U}\,i\Big)\wedge\Big(\big(b\rightarrow\bigcirc(\neg b\,\mathcal{U}\,i)\big)\,\mathcal{U}\,i\Big)\\&\wedge\Big(\big(c\rightarrow\bigcirc(\neg c\,\mathcal{U}\,i)\big)\,\mathcal{U}\,i\Big)\wedge\Big(\big(d1\rightarrow\bigcirc(\neg d1\,\mathcal{U}\,i)\big)\,\mathcal{U}\,i\Big)\wedge\Big(\big(d2\rightarrow\bigcirc(\neg d2\,\mathcal{U}\,i)\big)\,\mathcal{U}\,i\Big)\end{aligned}\tag{$\mathcal{A}4$-MaxOneNR}$$

$$\Box\big(d1\rightarrow(\neg d2\,\mathcal{U}\,i)\big)\wedge\Box\big(d2\rightarrow(\neg d1\,\mathcal{U}\,i)\big)\tag{$\mathcal{A}4$-Range}$$

$$\Big(\big(d1\rightarrow(\neg d2\,\mathcal{U}\,i)\big)\,\mathcal{U}\,i\Big)\wedge\Big(\big(d2\rightarrow(\neg d1\,\mathcal{U}\,i)\big)\,\mathcal{U}\,i\Big)\tag{$\mathcal{A}4$-RangeNR}$$

$$\begin{aligned}&\Box\big(c\rightarrow(\neg a\,\mathcal{U}\,i)\big)\wedge\Box\big(c\rightarrow(\neg b\,\mathcal{U}\,i)\big)\\&\wedge\Box\big(d1\rightarrow(\neg a\,\mathcal{U}\,i)\big)\wedge\Box\big(d1\rightarrow(\neg b\,\mathcal{U}\,i)\big)\wedge\Box\big(d2\rightarrow(\neg a\,\mathcal{U}\,i)\big)\wedge\Box\big(d2\rightarrow(\neg b\,\mathcal{U}\,i)\big)\end{aligned}\tag{$\mathcal{A}4$-Order}$$

$$\begin{aligned}&\Big(\big(c\rightarrow(\neg a\,\mathcal{U}\,i)\big)\,\mathcal{U}\,i\Big)\wedge\Big(\big(c\rightarrow(\neg b\,\mathcal{U}\,i)\big)\,\mathcal{U}\,i\Big)\wedge\Big(\big(d1\rightarrow(\neg a\,\mathcal{U}\,i)\big)\,\mathcal{U}\,i\Big)\\&\wedge\Big(\big(d1\rightarrow(\neg b\,\mathcal{U}\,i)\big)\,\mathcal{U}\,i\Big)\wedge\Big(\big(d2\rightarrow(\neg a\,\mathcal{U}\,i)\big)\,\mathcal{U}\,i\Big)\wedge\Big(\big(d2\rightarrow(\neg b\,\mathcal{U}\,i)\big)\,\mathcal{U}\,i\Big)\end{aligned}\tag{$\mathcal{A}4$-OrderNR}$$

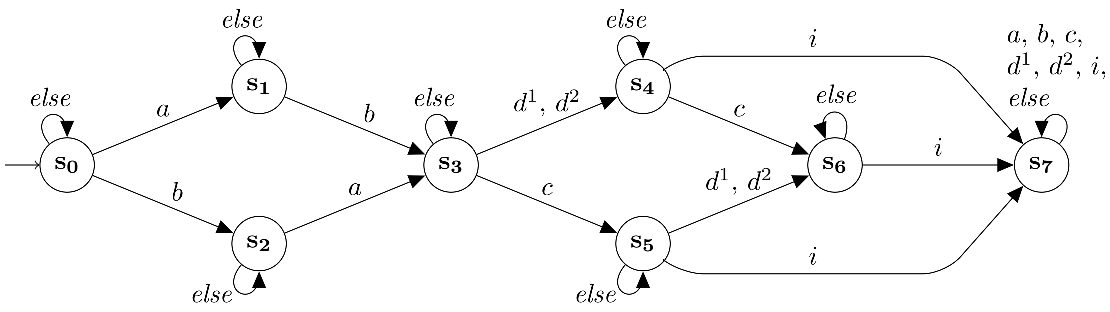
$$(\neg i\,\mathcal{U}\,a)\wedge(\neg i\,\mathcal{U}\,b)\wedge\big((\neg i\,\mathcal{U}\,c)\vee(\neg i\,\mathcal{U}\,d1)\vee(\neg i\,\mathcal{U}\,d2)\big)\tag{$\mathcal{A}4$-First$\mathcal{F}'$}$$

$$\begin{aligned}&\Box\Big(i\rightarrow\bigcirc(\neg i\,\mathcal{U}\,a)\Big)\wedge\Box\Big(i\rightarrow\bigcirc(\neg i\,\mathcal{U}\,b)\Big)\\&\wedge\Box\Big(i\rightarrow\bigcirc\big((\neg i\,\mathcal{U}\,c)\vee(\neg i\,\mathcal{U}\,d1)\vee(\neg i\,\mathcal{U}\,d2)\big)\Big)\end{aligned}\tag{$\mathcal{A}4$-After$\mathcal{F}'$}$$

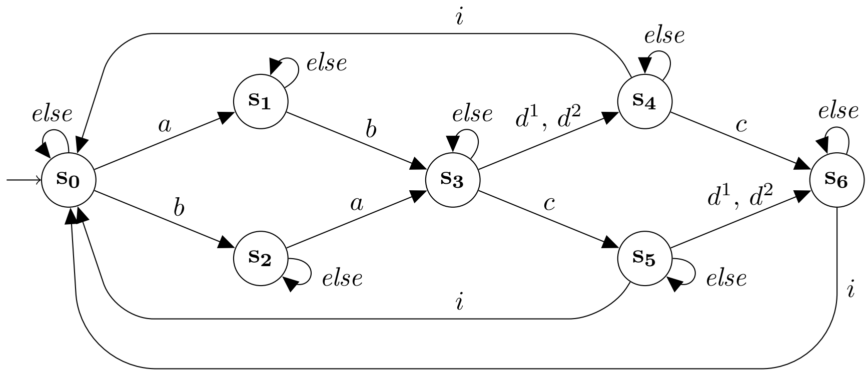
The encoding of $\Big(\big(\{a,b\},\wedge\big) < \big(\{c,d^{[1,2]}\},\vee\big) \ll i \mid \textit{Non-Repeated}\Big)$ with a non-repeated context into LTL is Conjunction B.5. The corresponding SPOT monitor is shown in Figure B.5(a). The encoding of the antecedent requirement $\Big(\big(\{a,b\},\wedge\big) < \big(\{c,d^{[1,2]}\},\vee\big) \ll i \mid \textit{Repeated}\Big)$ with a repeated context into LTL is Conjunction B.6. The corresponding SPOT monitor is in Figure B.5(b). Figure B.6 provides the LTL encoding of the property with the repeated context in SPOT syntax.

$$\mathcal{A}4\text{-Exclusiveness}\wedge\mathcal{A}4\text{-MaxOneNR}\wedge\mathcal{A}4\text{-RangeNR}\wedge\mathcal{A}4\text{-OrderNR}\wedge\mathcal{A}4\text{-First}\mathcal{F}'\tag{B.5}$$

$$\mathcal{A}4\text{-Exclusiveness}\wedge\mathcal{A}4\text{-MaxOne}\wedge\mathcal{A}4\text{-Range}\wedge\mathcal{A}4\text{-Order}\wedge\mathcal{A}4\text{-First}\mathcal{F}'\wedge\mathcal{A}4\text{-After}\mathcal{F}'\tag{B.6}$$



(a) A non-repeated context $\nabla = \textit{Non-Repeated}$



(b) A repeated context $\nabla = \textit{Repeated}$

Figure B.5 The SPOT monitor for the LTL encoding of $\big((\{a,b\},\wedge) < (\{c,d^{[1,2]}\},\vee) \ll i \mid \nabla\big)$. *else* stands for any name which is not of $\{a,b,c,d1,d2,i\}$. "$d1$, $d2$" means that $d1$ and $d2$ cannot occur simultaneously. Transitions which are not defined are forbidden.

```
~ A4-Exclusiveness
G(!(a  && d1)) && G(!(a  && d2))&& G(!(a  && c)) &&
G(!(a  && b))  && G(!(a  && i)) &&
G(!(d1 && d2)) && G(!(d1 && c)) && G(!(d1 && b)) &&
G(!(d1 && i))  &&
G(!(d2 && c))  && G(!(d2 && b)) && G(!(d2 && i)) &&
G(!(c  && b))  && G(!(c  && i)) && G(!(b  && i)) &&

~ A4-MaxOne
G(a -> X(!a U i)) && G(b -> X(!b U i)) && G(c -> X(!c U i)) &&
G(d1-> X(!d1 U i)) && G(d2-> X(!d2 U i)) &&

~ A4-Range
G(d1 -> (!d2 U i)) && G(d2 -> (!d1 U i)) &&

~ A4-Order
G(c-> (!a U i))  && G(c-> (!b U i))  &&
G(d1-> (!a U i)) && G(d1-> (!b U i)) &&
G(d2-> (!a U i)) && G(d2-> (!b U i)) &&

~ A4-FirstF'
(!i U a) && (!i U b) &&
((!i U c) || (!i U d1) || (!i U d2))   &&

~ A4-AfterF'
G(i -> X(!i U a)) && G(i -> X(!i U b)) &&
G(i -> X((!i U c) || (!i U d1) || (!i U d2)))
```

Figure B.6 The LTL encoding of $\big((\{a,b\},\wedge) < (\{c,d^{[1,2]}\},\vee) \ll i \mid \textit{Repeated}\big)$ in SPOT syntax.

B.1.4 Encoding of $\Big(\big(\{a, b^{[1,2]}\}, \vee\big) < \big(\{c, d\}, \wedge\big) \ll i \mid \nabla\Big)$ into LTL

Consider the antecedent requirement $\Big(\big(\{a, b^{[1,2]}\}, \vee\big) < \big(\{c, d\}, \wedge\big) \ll i \mid \nabla\Big)$. After the removal of the range $b^{[1,2]}$, the new vocabulary of the property is the set $\{a, b1, b2, c, d, i\}$ (see Chapter 4, Sec. 4.5.3). The LTL encoding of the property consists of the following components:

$$\begin{aligned}&\Box\big(\neg(a \wedge b1)\big) \wedge \Box\big(\neg(a \wedge b2)\big) \wedge \Box\big(\neg(a \wedge c)\big) \wedge \Box\big(\neg(a \wedge d)\big)\\ &\wedge \Box\big(\neg(a \wedge i)\big) \wedge \Box\big(\neg(b1 \wedge b2)\big) \wedge \Box\big(\neg(b1 \wedge c)\big) \wedge \Box\big(\neg(b1 \wedge d)\big)\\ &\wedge \Box\big(\neg(b1 \wedge i)\big) \wedge \Box\big(\neg(b2 \wedge c)\big) \wedge \Box\big(\neg(b2 \wedge d)\big) \wedge \Box\big(\neg(b2 \wedge i)\big)\\ &\wedge \Box\big(\neg(c \wedge d)\big) \wedge \Box\big(\neg(c \wedge i)\big) \wedge \Box\big(\neg(d \wedge i)\big)\end{aligned} \tag{$\mathcal{A}$5-Exclusiveness}$$

$$\begin{aligned}&\Box\big(a \rightarrow \bigcirc(\neg a\ \mathcal{U}\ i)\big) \wedge \Box\big(b1 \rightarrow \bigcirc(\neg b1\ \mathcal{U}\ i)\big) \wedge \Box\big(b2 \rightarrow \bigcirc(\neg b2\ \mathcal{U}\ i)\big)\\ &\wedge \Box\big(c \rightarrow \bigcirc(\neg c\ \mathcal{U}\ i)\big) \wedge \Box\big(d \rightarrow \bigcirc(\neg d\ \mathcal{U}\ i)\big)\end{aligned} \tag{$\mathcal{A}$5-MaxOne}$$

$$\begin{aligned}&\Big(\big(a \rightarrow \bigcirc(\neg a\ \mathcal{U}\ i)\big)\ \mathcal{U}\ i\Big) \wedge \Big(\big(b1 \rightarrow \bigcirc(\neg b1\ \mathcal{U}\ i)\big)\ \mathcal{U}\ i\Big)\\ &\wedge \Big(\big(b2 \rightarrow \bigcirc(\neg b2\ \mathcal{U}\ i)\big)\ \mathcal{U}\ i\Big) \wedge \Big(\big(c \rightarrow \bigcirc(\neg c\ \mathcal{U}\ i)\big)\ \mathcal{U}\ i\Big) \wedge \Big(\big(d \rightarrow \bigcirc(\neg d\ \mathcal{U}\ i)\big)\ \mathcal{U}\ i\Big)\end{aligned} \tag{$\mathcal{A}$5-MaxOneNR}$$

$$\Box\big(b1 \rightarrow (\neg b2\ \mathcal{U}\ i)\big) \wedge \Box\big(b2 \rightarrow (\neg b1\ \mathcal{U}\ i)\big) \tag{$\mathcal{A}$5-Range}$$

$$\Big(\big(b1 \rightarrow (\neg b2\ \mathcal{U}\ i)\big)\ \mathcal{U}\ i\Big) \wedge \Big(\big(b2 \rightarrow (\neg b1\ \mathcal{U}\ i)\big)\ \mathcal{U}\ i\Big) \tag{$\mathcal{A}$5-RangeNR}$$

$$\begin{aligned}&\Box\big(c \rightarrow (\neg a\ \mathcal{U}\ i)\big) \wedge \Box\big(c \rightarrow (\neg b1\ \mathcal{U}\ i)\big) \wedge \Box\big(c \rightarrow (\neg b2\ \mathcal{U}\ i)\big)\\ &\wedge \Box\big(d \rightarrow (\neg a\ \mathcal{U}\ i)\big) \wedge \Box\big(d \rightarrow (\neg b1\ \mathcal{U}\ i)\big) \wedge \Box\big(d \rightarrow (\neg b2\ \mathcal{U}\ i)\big)\end{aligned} \tag{$\mathcal{A}$5-Order}$$

$$\begin{aligned}&\Big(\big(c \rightarrow (\neg a\ \mathcal{U}\ i)\big)\ \mathcal{U}\ i\Big) \wedge \Big(\big(c \rightarrow (\neg b1\ \mathcal{U}\ i)\big)\ \mathcal{U}\ i\Big) \wedge \Big(\big(c \rightarrow (\neg b2\ \mathcal{U}\ i)\big)\ \mathcal{U}\ i\Big)\\ &\wedge \Big(\big(d \rightarrow (\neg a\ \mathcal{U}\ i)\big)\ \mathcal{U}\ i\Big) \wedge \Big(\big(d \rightarrow (\neg b1\ \mathcal{U}\ i)\big)\ \mathcal{U}\ i\Big) \wedge \Big(\big(d \rightarrow (\neg b2\ \mathcal{U}\ i)\big)\ \mathcal{U}\ i\Big)\end{aligned} \tag{$\mathcal{A}$5-OrderNR}$$

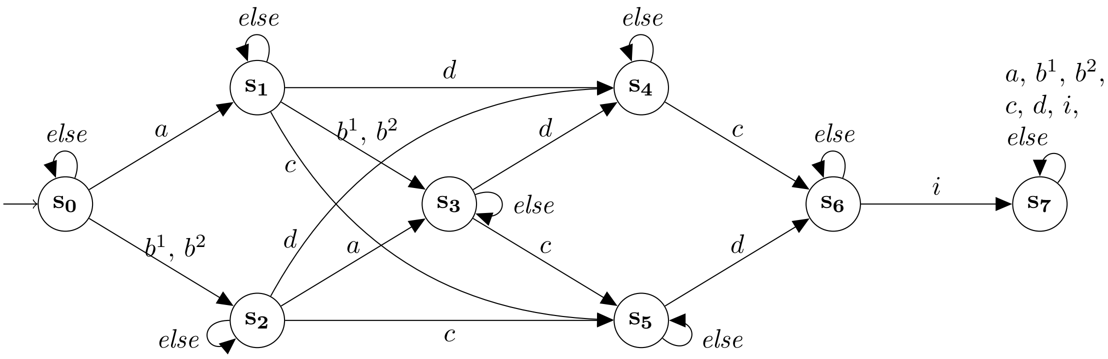
$$\big((\neg i\ \mathcal{U}\ b1) \vee (\neg i\ \mathcal{U}\ b2) \vee (\neg i\ \mathcal{U}\ a)\big) \wedge \big(\neg i\ \mathcal{U}\ c\big) \wedge \big(\neg i\ \mathcal{U}\ d\big) \tag{$\mathcal{A}$5-First$\mathcal{F}'$}$$

$$\Box\Big(i \rightarrow \bigcirc\big((\neg i\ \mathcal{U}\ b1) \vee (\neg i\ \mathcal{U}\ b2) \vee (\neg i\ \mathcal{U}\ a)\big)\Big) \wedge \Box\Big(i \rightarrow \bigcirc(\neg i\ \mathcal{U}\ c)\Big) \wedge \Box\Big(i \rightarrow \bigcirc(\neg i\ \mathcal{U}\ d)\Big) \tag{$\mathcal{A}$5-After$\mathcal{F}'$}$$

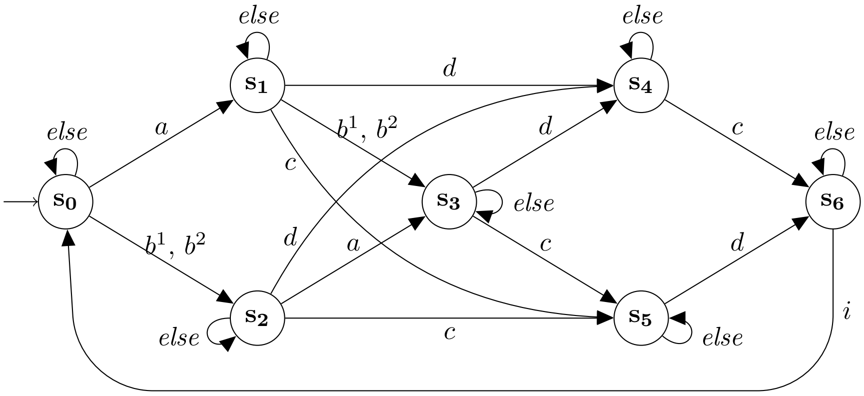
The encoding of $\Big(\big(\{a, b^{[1,2]}\}, \vee\big) < \big(\{c, d\}, \wedge\big) \ll i \mid \textit{Non-Repeated}\Big)$ with a non-repeated context into LTL is Conjunction B.7. The corresponding SPOT monitor is shown in Figure B.7(a). The encoding of $\Big(\big(\{a, b^{[1,2]}\}, \vee\big) < \big(\{c, d\}, \wedge\big) \ll i \mid \textit{Repeated}\Big)$ with a repeated context into LTL is Conjunction B.8. The corresponding SPOT monitor is in Figure B.7(b). Figure B.8 provides the LTL encoding of the property with the repeated context in SPOT syntax.

$$\mathcal{A}\text{5-Exclusiveness} \wedge \mathcal{A}\text{5-MaxOneNR} \wedge \mathcal{A}\text{5-RangeNR} \wedge \mathcal{A}\text{5-OrderNR} \wedge \mathcal{A}\text{5-First}\mathcal{F}' \tag{B.7}$$

$$\mathcal{A}\text{5-Exclusiveness} \wedge \mathcal{A}\text{5-MaxOne} \wedge \mathcal{A}\text{5-Range} \wedge \mathcal{A}\text{5-Order} \wedge \mathcal{A}\text{5-First}\mathcal{F}' \wedge \mathcal{A}\text{5-After}\mathcal{F}' \tag{B.8}$$



(a) A non-repeated context $\nabla = \textit{Non-Repeated}$



(b) A repeated context $\nabla = \textit{Repeated}$

Figure B.7 The SPOT monitor of the LTL encoding of $\big((\{a, b^{[1,2]}\}, \vee) < (\{c, d\}, \wedge) \ll i \mid \nabla\big)$. *else* stands for any name which is not of $\{a, b1, b2, c, d, i\}$. "$b1$, $b2$" means that $b1$ and $b2$ cannot occur simultaneously. Transitions which are not defined are forbidden.

```
~ A5-Exclusiveness
G(!(a  && b1)) && G(!(a  && b2))&& G(!(a  && c)) &&
G(!(a  && d))  && G(!(a  && i)) &&
G(!(b1 && b2)) && G(!(b1 && c)) && G(!(b1 && d)) &&
G(!(b1 && i))  &&
G(!(b2 && c))  && G(!(b2 && d)) && G(!(b2 && i)) &&
G(!(c  && d))  && G(!(c  && i)) && G(!(d  && i)) &&

~ A5-MaxOne
G(a -> X(!a U i))  &&
G(b1-> X(!b1 U i)) && G(b2-> X(!b2 U i)) &&
G(c -> X(!c U i))  && G(d -> X(!d U i))  &&

~ A5-Range
G(b1 -> (!b2 U i)) && G(b2 -> (!b1 U i)) &&

~ A5-Order
G(c-> (!a U i)) && G(c-> (!b1 U i)) && G(c-> (!b2 U i)) &&
G(d-> (!a U i)) && G(d-> (!b1 U i)) && G(d-> (!b2 U i)) &&

~ A5-FirstF'
((!i U b1) || (!i U b2) || (!i U a ))) &&
(!i U c) && (!i U d)   &&

~ A5-AfterF'
G(i -> X((!i U b1) || (!i U b2) || (!i U a)) &&
G(i -> X(!i U c)) && G(i -> X(!i U d))
```

Figure B.8 The LTL encoding of $\big((\{a, b^{[1,2]}\}, \vee) < (\{c, d\}, \wedge) \ll i \mid \textit{Repeated}\big)$ in SPOT syntax.

B.1.5 Encoding of $\Big(\big(\{a,b\},\vee\big) < \big(\{c,d^{[1,2]}\},\wedge\big) \ll i \mid \nabla\Big)$ into LTLT

Consider the antecedent requirement $\Big(\big(\{a,b\},\vee\big) < \big(\{c,d^{[1,2]}\},\wedge\big) \ll i \mid \nabla\Big)$. After the removal of the range $d^{[1,2]}$, the new vocabulary of the property is the set $\{a,b,c,d1,d2,i\}$ (see Chapter 4, Sec. 4.5.3). The LTL encoding of the property consists of the following components:

$$\begin{aligned}&\Box\big(\neg(a\wedge d1)\big)\wedge\Box\big(\neg(a\wedge d2)\big)\wedge\Box\big(\neg(a\wedge c)\big)\wedge\Box\big(\neg(a\wedge b)\big)\\&\wedge\Box\big(\neg(a\wedge i)\big)\wedge\Box\big(\neg(d1\wedge d2)\big)\wedge\Box\big(\neg(d1\wedge c)\big)\wedge\Box\big(\neg(d1\wedge b)\big)\\&\wedge\Box\big(\neg(d1\wedge i)\big)\wedge\Box\big(\neg(d2\wedge c)\big)\wedge\Box\big(\neg(d2\wedge b)\big)\wedge\Box\big(\neg(d2\wedge i)\big)\\&\wedge\Box\big(\neg(c\wedge b)\big)\wedge\Box\big(\neg(c\wedge i)\big)\wedge\Box\big(\neg(b\wedge i)\big)\end{aligned}\tag{$\mathcal{A}6$-Exclusiveness}$$

$$\begin{aligned}&\Box\big(a\rightarrow\bigcirc(\neg a\,\mathcal{U}\,i)\big)\wedge\Box\big(b\rightarrow\bigcirc(\neg b\,\mathcal{U}\,i)\big)\\&\wedge\Box\big(c\rightarrow\bigcirc(\neg c\,\mathcal{U}\,i)\big)\wedge\Box\big(d1\rightarrow\bigcirc(\neg d1\,\mathcal{U}\,i)\big)\wedge\Box\big(d2\rightarrow\bigcirc(\neg d2\,\mathcal{U}\,i)\big)\end{aligned}\tag{$\mathcal{A}6$-MaxOne}$$

$$\begin{aligned}&\Big(\big(a\rightarrow\bigcirc(\neg a\,\mathcal{U}\,i)\big)\,\mathcal{U}\,i\Big)\wedge\Big(\big(b\rightarrow\bigcirc(\neg b\,\mathcal{U}\,i)\big)\,\mathcal{U}\,i\Big)\\&\wedge\Big(\big(c\rightarrow\bigcirc(\neg c\,\mathcal{U}\,i)\big)\,\mathcal{U}\,i\Big)\wedge\Big(\big(d1\rightarrow\bigcirc(\neg d1\,\mathcal{U}\,i)\big)\,\mathcal{U}\,i\Big)\wedge\Big(\big(d2\rightarrow\bigcirc(\neg d2\,\mathcal{U}\,i)\big)\,\mathcal{U}\,i\Big)\end{aligned}\tag{$\mathcal{A}6$-MaxOneNR}$$

$$\Box\big(d1\rightarrow(\neg d2\,\mathcal{U}\,i)\big)\wedge\Box\big(d2\rightarrow(\neg d1\,\mathcal{U}\,i)\big)\tag{$\mathcal{A}6$-Range}$$

$$\Big(\big(d1\rightarrow(\neg d2\,\mathcal{U}\,i)\big)\,\mathcal{U}\,i\Big)\wedge\Big(\big(d2\rightarrow(\neg d1\,\mathcal{U}\,i)\big)\,\mathcal{U}\,i\Big)\tag{$\mathcal{A}6$-RangeNR}$$

$$\begin{aligned}&\Box\big(c\rightarrow(\neg a\,\mathcal{U}\,i)\big)\wedge\Box\big(c\rightarrow(\neg b\,\mathcal{U}\,i)\big)\\&\wedge\Box\big(d1\rightarrow(\neg a\,\mathcal{U}\,i)\big)\wedge\Box\big(d1\rightarrow(\neg b\,\mathcal{U}\,i)\big)\wedge\Box\big(d2\rightarrow(\neg a\,\mathcal{U}\,i)\big)\wedge\Box\big(d2\rightarrow(\neg b\,\mathcal{U}\,i)\big)\end{aligned}\tag{$\mathcal{A}6$-Order}$$

$$\begin{aligned}&\Big(\big(c\rightarrow(\neg a\,\mathcal{U}\,i)\big)\,\mathcal{U}\,i\Big)\wedge\Big(\big(c\rightarrow(\neg b\,\mathcal{U}\,i)\big)\,\mathcal{U}\,i\Big)\wedge\Big(\big(d1\rightarrow(\neg a\,\mathcal{U}\,i)\big)\,\mathcal{U}\,i\Big)\\&\wedge\Big(\big(d1\rightarrow(\neg b\,\mathcal{U}\,i)\big)\,\mathcal{U}\,i\Big)\wedge\Big(\big(d2\rightarrow(\neg a\,\mathcal{U}\,i)\big)\,\mathcal{U}\,i\Big)\wedge\Big(\big(d2\rightarrow(\neg b\,\mathcal{U}\,i)\big)\,\mathcal{U}\,i\Big)\end{aligned}\tag{$\mathcal{A}6$-OrderNR}$$

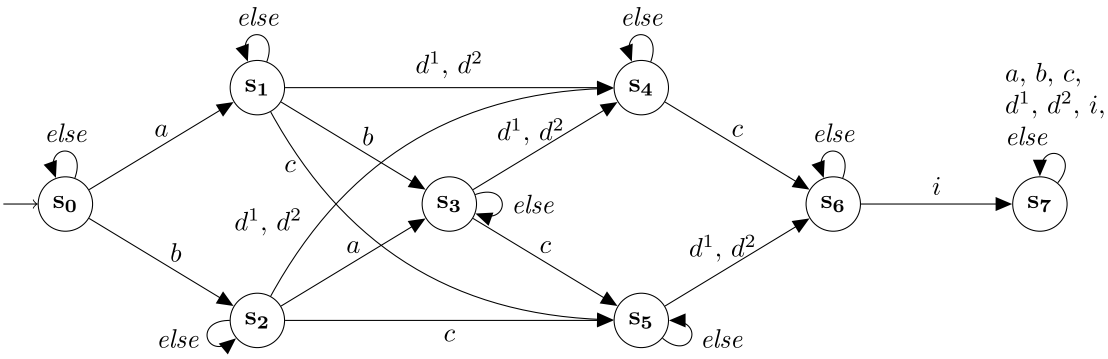
$$\big((\neg i\,\mathcal{U}\,a)\vee(\neg i\,\mathcal{U}\,b)\big)\wedge\big(\neg i\,\mathcal{U}\,c\big)\wedge\big((\neg i\,\mathcal{U}\,d1)\vee(\neg i\,\mathcal{U}\,d2)\big)\tag{$\mathcal{A}6$-First$\mathcal{F}'$}$$

$$\begin{aligned}&\Box\Big(i\rightarrow\bigcirc\big((\neg i\,\mathcal{U}\,a)\vee(\neg i\,\mathcal{U}\,b)\big)\Big)\wedge\Box\Big(i\rightarrow\bigcirc(\neg i\,\mathcal{U}\,c)\Big)\\&\wedge\Box\Big(i\rightarrow\bigcirc\big((\neg i\,\mathcal{U}\,d1)\vee(\neg i\,\mathcal{U}\,d2)\big)\Big)\end{aligned}\tag{$\mathcal{A}6$-After$\mathcal{F}'$}$$

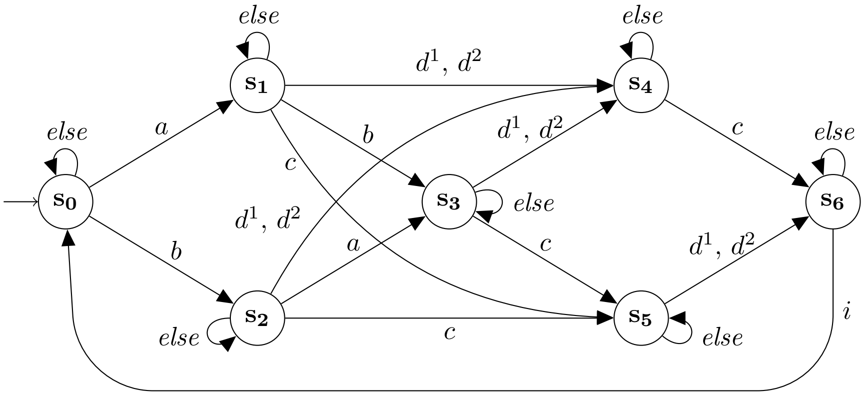
The encoding of $\Big(\big(\{a,b\},\vee\big) < \big(\{c,d^{[1,2]}\},\wedge\big) \ll i \mid \textit{Non-Repeated}\Big)$ with a non-repeated context into LTL is Conjunction B.9. The corresponding SPOT monitor is shown in Figure B.9(a). The encoding of $\Big(\big(\{a,b\},\vee\big) < \big(\{c,d^{[1,2]}\},\wedge\big) \ll i \mid \textit{Repeated}\Big)$ with a repeated context into LTL is Conjunction B.10. The corresponding SPOT monitor is in Figure B.9(b). Figure B.10 provides the LTL encoding of the property with the repeated context in SPOT syntax.

$$\mathcal{A}6\text{-Exclusiveness}\wedge\mathcal{A}6\text{-MaxOneNR}\wedge\mathcal{A}6\text{-RangeNR}\wedge\mathcal{A}6\text{-OrderNR}\wedge\mathcal{A}6\text{-First}\mathcal{F}'\tag{B.9}$$

$$\mathcal{A}6\text{-Exclusiveness}\wedge\mathcal{A}6\text{-MaxOne}\wedge\mathcal{A}6\text{-Range}\wedge\mathcal{A}6\text{-Order}\wedge\mathcal{A}6\text{-First}\mathcal{F}'\wedge\mathcal{A}6\text{-After}\mathcal{F}'\tag{B.10}$$



(a) A non-repeated context $\nabla = \textit{Non-Repeated}$



(b) A repeated context $\nabla = \textit{Repeated}$

Figure B.9 The SPOT monitor of the LTL encoding of $\big((\{a,b\},\vee) < (\{c,d^{[1,2]}\},\wedge) \ll i \mid \nabla\big)$. *else* stands for any name which is not of $\{a,b,c,d^1,d^2,i\}$. "$d1$, $d2$" means that $d1$ and $d2$ cannot occur simultaneously. Transitions which are not defined are forbidden.

```
~ A6-Exclusiveness
G(!(a  && d1)) && G(!(a  && d2))&& G(!(a  && c)) &&
G(!(a  && b))  && G(!(a  && i)) &&
G(!(d1 && d2)) && G(!(d1 && c)) && G(!(d1 && b)) &&
G(!(d1 && i))  &&
G(!(d2 && c))  && G(!(d2 && b)) && G(!(d2 && i)) &&
G(!(c  && b))  && G(!(c  && i)) && G(!(b  && i)) &&

~ A6-MaxOne
G(a -> X(!a U i)) && G(b -> X(!b U i)) && G(c -> X(!c U i)) &&
G(d1-> X(!d1 U i)) && G(d2-> X(!d2 U i)) &&

~ A6-Range
G(d1 -> (!d2 U i)) && G(d2 -> (!d1 U i)) &&

~ A6-Order
G(c-> (!a U i))  && G(c-> (!b U i))  &&
G(d1-> (!a U i)) && G(d1-> (!b U i)) &&
G(d2-> (!a U i)) && G(d2-> (!b U i)) &&

~ A6-FirstF'
((!i U a) || (!i U b)) &&
(!i U c) && ((!i U d1) || (!i U d2))   &&

~ A6-AfterF'
G(i -> X((!i U a) || (!i U b))) &&
G(i -> X(!i U c)) && G(i->X((!i U d1) || (!i U d2)))
```

Figure B.10 The LTL encoding of $\big((\{a,b\},\vee) < (\{c,d^{[1,2]}\},\wedge) \ll i \mid \textit{Repeated}\big)$ in SPOT syntax.

B.1.6 Encoding of $\Big(\big(\{a,b\},\wedge\big) < \big(\{c,d\},\vee\big) < \big(\{e,f\},\wedge\big) \ll i \mid \nabla\Big)$ into LTL

Consider the antecedent requirement $\Big(\big(\{a,b\},\wedge\big) < \big(\{c,d\},\vee\big) < \big(\{e,f\},\wedge\big) \ll i \mid \nabla\Big)$. The vocabulary of the property is the set $\{a,b,c,d,e,f,i\}$. The LTL encoding of the property consists of the following components:

$$\begin{aligned}&\Box\big(\neg(a\wedge b)\big)\wedge\Box\big(\neg(a\wedge c)\big)\wedge\Box\big(\neg(a\wedge d)\big)\wedge\Box\big(\neg(a\wedge e)\big)\wedge\Box\big(\neg(a\wedge f)\big)\\&\wedge\Box\big(\neg(a\wedge i)\big)\wedge\Box\big(\neg(b\wedge c)\big)\wedge\Box\big(\neg(b\wedge d)\big)\wedge\Box\big(\neg(b\wedge e)\big)\wedge\Box\big(\neg(b\wedge f)\big)\\&\wedge\Box\big(\neg(b\wedge i)\big)\wedge\Box\big(\neg(c\wedge d)\big)\wedge\Box\big(\neg(c\wedge e)\big)\wedge\Box\big(\neg(c\wedge f)\big)\wedge\Box\big(\neg(c\wedge i)\big)\\&\wedge\Box\big(\neg(d\wedge e)\big)\wedge\Box\big(\neg(d\wedge f)\big)\wedge\Box\big(\neg(d\wedge i)\big)\wedge\Box\big(\neg(e\wedge f)\big)\wedge\Box\big(\neg(e\wedge i)\big)\wedge\Box\big(\neg(f\wedge i)\big)\end{aligned} \tag{$\mathcal{A}7$-Exclusiveness}$$

$$\begin{aligned}&\Box\big(a\rightarrow\bigcirc(\neg a\,\mathcal{U}\,i)\big)\wedge\Box\big(b\rightarrow\bigcirc(\neg b\,\mathcal{U}\,i)\big)\wedge\Box\big(c\rightarrow\bigcirc(\neg c\,\mathcal{U}\,i)\big)\\&\wedge\Box\big(d\rightarrow\bigcirc(\neg d\,\mathcal{U}\,i)\big)\wedge\Box\big(e\rightarrow\bigcirc(\neg e\,\mathcal{U}\,i)\big)\wedge\Box\big(f\rightarrow\bigcirc(\neg f\,\mathcal{U}\,i)\big)\end{aligned} \tag{$\mathcal{A}7$-MaxOne}$$

$$\begin{aligned}&\Big(\big(a\rightarrow\bigcirc(\neg a\,\mathcal{U}\,i)\big)\,\mathcal{U}\,i\Big)\wedge\Big(\big(b\rightarrow\bigcirc(\neg b\,\mathcal{U}\,i)\big)\,\mathcal{U}\,i\Big)\\&\wedge\Big(\big(c\rightarrow\bigcirc(\neg c\,\mathcal{U}\,i)\big)\,\mathcal{U}\,i\Big)\wedge\Big(\big(d\rightarrow\bigcirc(\neg d\,\mathcal{U}\,i)\big)\,\mathcal{U}\,i\Big)\\&\wedge\Big(\big(e\rightarrow\bigcirc(\neg e\,\mathcal{U}\,i)\big)\,\mathcal{U}\,i\Big)\wedge\Big(\big(f\rightarrow\bigcirc(\neg f\,\mathcal{U}\,i)\big)\,\mathcal{U}\,i\Big)\end{aligned} \tag{$\mathcal{A}7$-MaxOneNR}$$

$$\begin{aligned}&\Box\big(c\rightarrow(\neg a\,\mathcal{U}\,i)\big)\wedge\Box\big(c\rightarrow(\neg b\,\mathcal{U}\,i)\big)\wedge\Box\big(d\rightarrow(\neg a\,\mathcal{U}\,i)\big)\wedge\Box\big(d\rightarrow(\neg b\,\mathcal{U}\,i)\big)\\&\wedge\Box\big(e\rightarrow(\neg c\,\mathcal{U}\,i)\big)\wedge\Box\big(e\rightarrow(\neg d\,\mathcal{U}\,i)\big)\wedge\Box\big(f\rightarrow(\neg c\,\mathcal{U}\,i)\big)\wedge\Box\big(f\rightarrow(\neg d\,\mathcal{U}\,i)\big)\end{aligned} \tag{$\mathcal{A}7$-Order}$$

$$\begin{aligned}&\Big(\big(c\rightarrow(\neg a\,\mathcal{U}\,i)\big)\,\mathcal{U}\,i\Big)\wedge\Big(\big(c\rightarrow(\neg b\,\mathcal{U}\,i)\big)\,\mathcal{U}\,i\Big)\wedge\Big(\big(d\rightarrow(\neg a\,\mathcal{U}\,i)\big)\,\mathcal{U}\,i\Big)\\&\wedge\Big(\big(d\rightarrow(\neg b\,\mathcal{U}\,i)\big)\,\mathcal{U}\,i\Big)\wedge\Big(\big(e\rightarrow(\neg c\,\mathcal{U}\,i)\big)\,\mathcal{U}\,i\Big)\wedge\Big(\big(e\rightarrow(\neg d\,\mathcal{U}\,i)\big)\,\mathcal{U}\,i\Big)\\&\wedge\Big(\big(f\rightarrow(\neg c\,\mathcal{U}\,i)\big)\,\mathcal{U}\,i\Big)\wedge\Big(\big(f\rightarrow(\neg d\,\mathcal{U}\,i)\big)\,\mathcal{U}\,i\Big)\end{aligned} \tag{$\mathcal{A}7$-OrderNR}$$

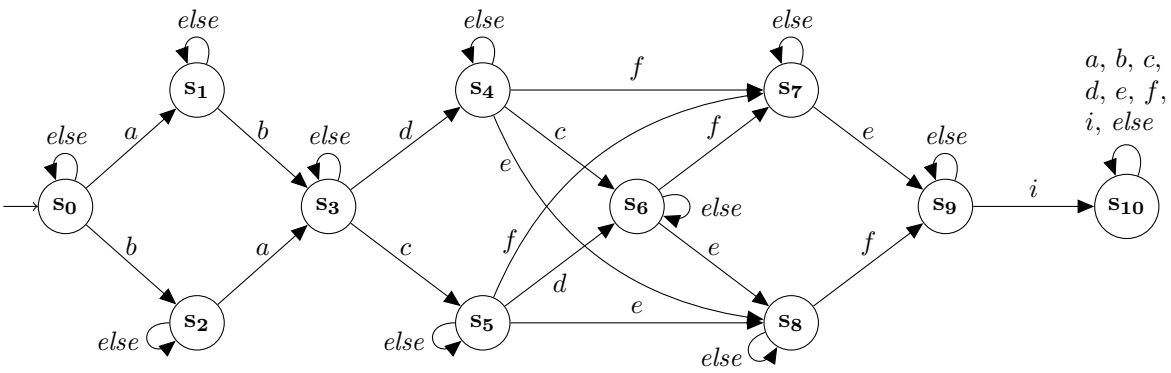
$$\big(\neg i\,\mathcal{U}\,a\big)\wedge\big(\neg i\,\mathcal{U}\,b\big)\wedge\big((\neg i\,\mathcal{U}\,c)\vee(\neg i\,\mathcal{U}\,d)\big)\wedge\big(\neg i\,\mathcal{U}\,e\big)\wedge\big(\neg i\,\mathcal{U}\,f\big) \tag{$\mathcal{A}7$-First$\mathcal{F}'$}$$

$$\begin{aligned}&\Box\Big(i\rightarrow\bigcirc(\neg i\,\mathcal{U}\,a)\Big)\wedge\Box\Big(i\rightarrow\bigcirc(\neg i\,\mathcal{U}\,b)\Big)\wedge\Box\Big(i\rightarrow\bigcirc\big((\neg i\,\mathcal{U}\,c)\vee(\neg i\,\mathcal{U}\,d)\big)\Big)\\&\wedge\Box\Big(i\rightarrow\bigcirc(\neg i\,\mathcal{U}\,e)\Big)\wedge\Box\Big(i\rightarrow\bigcirc(\neg i\,\mathcal{U}\,f)\Big)\end{aligned} \tag{$\mathcal{A}7$-After$\mathcal{F}'$}$$

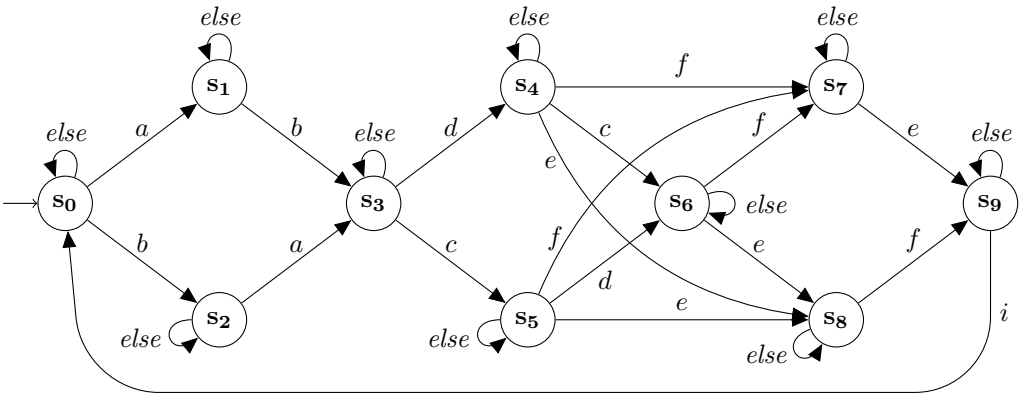
The encoding of $\Big(\big(\{a,b\},\wedge\big) < \big(\{c,d\},\vee\big) < \big(\{e,f\},\wedge\big) \ll i \mid \textit{Non-Repeated}\Big)$ with a non-repeated context into LTL is Conjunction B.11. The corresponding SPOT monitor is shown in Figure B.11(a). The encoding of the antecedent requirement $\Big(\big(\{a,b\},\wedge\big) < \big(\{c,d\},\vee\big) < \big(\{e,f\},\wedge\big) \ll i \mid \textit{Repeated}\Big)$ with a repeated context into LTL is Conjunction B.12. The corresponding SPOT monitor is in Figure B.11(b). Figure B.12 provides the LTL encoding of the property with the repeated context in SPOT syntax.

$$\mathcal{A}7\text{-Exclusiveness} \wedge \mathcal{A}7\text{-MaxOneNR} \wedge \mathcal{A}7\text{-OrderNR} \wedge \mathcal{A}7\text{-First}\mathcal{F}' \tag{B.11}$$

$$\mathcal{A}7\text{-Exclusiveness} \wedge \mathcal{A}7\text{-MaxOne} \wedge \mathcal{A}7\text{-Order} \wedge \mathcal{A}7\text{-First}\mathcal{F}' \wedge \mathcal{A}7\text{-After}\mathcal{F}' \tag{B.12}$$



(a) A non-repeated context $\nabla = \textit{Non-Repeated}$



(b) A repeated context $\nabla = \textit{Repeated}$

Figure B.11 The SPOT monitor for the LTL encoding of $\big((\{a,b\},\wedge) < (\{c,d\},\vee) < (\{e,f\},\wedge) \ll i \mid \nabla\big)$. *else* stands for any name which is not of $\{a,b,c,d,e,f,i\}$. Transitions which are not defined are forbidden.

```
~ A7-Exclusiveness
G(!(a && b)) && G(!(a && c))&& G(!(a && d)) && G(!(a && e)) &&
G(!(a && f)) && G(!(a && i)) &&
G(!(b && c))&& G(!(b && d)) && G(!(b && e)) && G(!(b && f)) &&
G(!(b && i)) &&
G(!(c && d)) && G(!(c && e)) && G(!(c && f)) && G(!(c && i)) &&
G(!(d && e)) && G(!(d && f)) && G(!(d && i)) &&
G(!(e && f)) && G(!(e && i)) &&
G(!(f && i)) &&

~ A7-MaxOne
G(a-> X(!a U i)) && G(b-> X(!b U i)) && G(c-> X(!c U i)) &&
G(d-> X(!d U i)) && G(e-> X(!e U i)) && G(f-> X(!f U i)) &&

~ A7-Order
G(c-> (!a U i)) && G(c-> (!b U i)) && G(d-> (!a U i)) && G(d-> (!b U i)) &&
G(e-> (!c U i)) && G(e-> (!d U i)) && G(f-> (!c U i)) && G(f-> (!d U i)) &&

~ A7-FirstF'
(!i U a) && (!i U b)&&((!i U c) || (!i U d))&&(!i U e) && (!i U f) &&

~ A7-AfterF'
G(i -> X(!i U a)) && G(i -> X(!i U b)) && G(i -> X((!i U c) || (!i U d))) &&
G(i -> X(!i U e)) && G(i -> X(!i U f))
```

Figure B.12 The LTL encoding of $\big((\{a,b\},\wedge) < (\{c,d\},\vee) < (\{e,f\},\wedge) \ll i \mid \textit{Repeated}\big)$ in SPOT syntax.

B.2 Encoding of a Timed Implication Constraint into LTL

B.2.1 Encoding of $\big(a \Longrightarrow (\{b,c\},\wedge) < (\{x,y\},\wedge) \;\mid t\big)$ into LTL

Consider the timed implication constraint $\big(a \Longrightarrow (\{b,c\},\wedge) < (\{x,y\},\wedge) \;\mid t\big)$. The LTL encoding of the property consists of the following components:

$$\Box(\neg(a \wedge b)) \wedge \Box(\neg(a \wedge c)) \wedge \Box(\neg(a \wedge x)) \wedge \Box(\neg(a \wedge y)) \wedge \Box(\neg(b \wedge c)) \wedge \Box(\neg(b \wedge x)) \wedge \Box(\neg(b \wedge y)) \wedge \Box(\neg(c \wedge x)) \wedge \Box(\neg(c \wedge y)) \wedge \Box(\neg(x \wedge y)) \tag{$\mathcal{T}$1-Exclusiveness}$$

$$\Box(a \to \bigcirc(\neg a\, \mathcal{U}\, x)) \wedge \Box(a \to \bigcirc(\neg a\, \mathcal{U}\, y)) \wedge \Box(b \to \bigcirc(\neg b\, \mathcal{U}\, x)) \wedge \Box(b \to \bigcirc(\neg b\, \mathcal{U}\, y)) \wedge \Box(c \to \bigcirc(\neg c\, \mathcal{U}\, x)) \wedge \Box(c \to \bigcirc(\neg c\, \mathcal{U}\, y)) \tag{$\mathcal{T}$1-MaxOne}$$

$$\Box(b \to (\neg a\, \mathcal{U}\, x)) \wedge \Box(b \to (\neg a\, \mathcal{U}\, y)) \wedge \Box(c \to (\neg a\, \mathcal{U}\, x)) \wedge \Box(c \to (\neg a\, \mathcal{U}\, y)) \tag{$\mathcal{T}$1-Order}$$

$$\Box(x \to (\neg b\, \mathcal{U}\, a)) \wedge \Box(x \to (\neg c\, \mathcal{U}\, a)) \wedge \Box(y \to (\neg b\, \mathcal{U}\, a)) \wedge \Box(y \to (\neg c\, \mathcal{U}\, a)) \tag{$\mathcal{T}$1-Order$\mathcal{F}'$}$$

$$(\neg x\, \mathcal{U}\, a) \wedge (\neg x\, \mathcal{U}\, b) \wedge (\neg x\, \mathcal{U}\, c) \wedge (\neg y\, \mathcal{U}\, a) \wedge (\neg y\, \mathcal{U}\, b) \wedge (\neg y\, \mathcal{U}\, c) \tag{$\mathcal{T}$1-First$\mathcal{F}'$}$$

$$\Box(x \to \bigcirc(\neg x\, \mathcal{U}\, a)) \wedge \Box(x \to \bigcirc(\neg x\, \mathcal{U}\, b)) \wedge \Box(x \to \bigcirc(\neg x\, \mathcal{U}\, c)) \wedge \Box(y \to \bigcirc(\neg y\, \mathcal{U}\, a)) \wedge \Box(y \to \bigcirc(\neg y\, \mathcal{U}\, b)) \wedge \Box(y \to \bigcirc(\neg y\, \mathcal{U}\, c)) \tag{$\mathcal{T}$1-After$\mathcal{F}'$}$$

The encoding of $\big(a \Longrightarrow (\{b,c\},\wedge) < (\{x,y\},\wedge) \;\mid t\big)$ is defined by Conjunction B.13. Figure B.14 provides the LTL encoding of the property in SPOT syntax. The respective SPOT monitor is provided in Figure B.13.

$$\mathcal{T}\text{1-Exclusiveness} \wedge \mathcal{T}\text{1-MaxOne} \wedge \mathcal{T}\text{1-Order} \wedge \mathcal{T}\text{1-Order}\mathcal{F}' \wedge \mathcal{T}\text{1-First}\mathcal{F}' \wedge \mathcal{T}\text{1-After}\mathcal{F}' \tag{B.13}$$

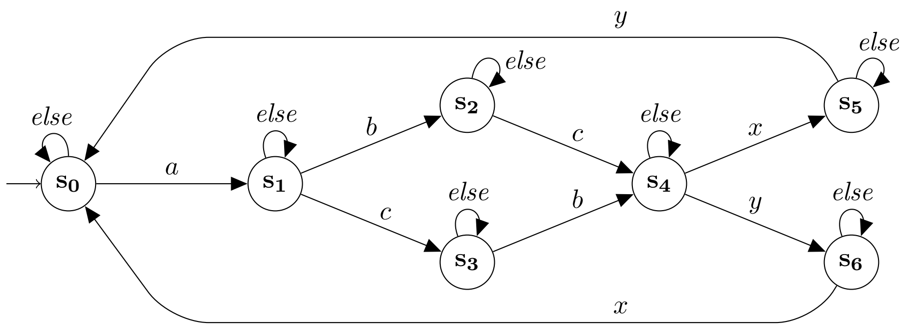


Figure B.13 The SPOT monitor of the LTL encoding of $\big(a \Longrightarrow (\{b,c\},\wedge) < (\{x,y\},\wedge) \;\mid t\big)$. *else* stands for any name which is not of $\{a,b,c,x,y\}$. Transitions which are not defined are forbidden.

```
~ T1-Exclusiveness
G(!(a && b)) && G(!(a && c)) && G(!(a && x)) && G(!(a && y)) &&
G(!(b && c)) && G(!(b && x)) && G(!(b && y)) &&
G(!(c && x)) && G(!(c && y)) &&
G(!(x && y)) &&

~ T1-MaxOne
G(a-> X(!a U x)) && G(a-> X(!a U y)) &&
G(b-> X(!b U x)) && G(b-> X(!b U y)) &&
G(c-> X(!c U x)) && G(c-> X(!c U y)) &&

~ T1-Order
G(b-> (!a U x)) && G(b-> (!a U y)) &&
G(c-> (!a U x)) && G(c-> (!a U y)) &&

~ T1-OrderF'
G(x-> (!b U a)) && G(x->(!c U a)) &&
G(y-> (!b U a)) && G(y->(!c U a)) &&

~ T1-FirstF'
(!x U a) && (!x U b) && (!x U c) &&
(!y U a) && (!y U b) && (!y U c) &&

~ T1-AfterF'
G(x -> X(!x U a)) && G(x -> X(!x U b)) && G(x -> X(!x U c)) &&
G(y -> X(!y U a)) && G(y -> X(!y U b)) && G(y -> X(!y U c))
```

Figure B.14 The LTL encoding of $\big(a \Longrightarrow (\{b,c\},\wedge) < (\{x,y\},\wedge) \ \mid t\big)$ in SPOT syntax.

B.2.2 Encoding of $\Big(a < (\{b, c^{[1,2]}\}, \wedge) \Longrightarrow (\{x, y\}, \wedge) \ \mid t\Big)$ into LTL

Consider the timed implication constraint $\Big(a < (\{b, c^{[1,2]}\}, \wedge) \Longrightarrow (\{x, y\}, \wedge) \ \mid t\Big)$. After the removal of the range $c^{[1,2]}$, the new vocabulary is the set $\{a, b, c1, c2, x, y\}$. The LTL encoding of the property consists of the following components:

$$\begin{aligned}&\Box(\neg(a \wedge b)) \wedge \Box(\neg(a \wedge c1)) \wedge \Box(\neg(a \wedge c2)) \wedge \Box(\neg(a \wedge x))\\ &\wedge \Box(\neg(a \wedge y)) \wedge \Box(\neg(b \wedge c1)) \wedge \Box(\neg(b \wedge c2)) \wedge \Box(\neg(b \wedge x))\\ &\wedge \Box(\neg(b \wedge y)) \wedge \Box(\neg(c1 \wedge c2)) \wedge \Box(\neg(c1 \wedge x)) \wedge \Box(\neg(c1 \wedge y))\\ &\wedge \Box(\neg(c2 \wedge x)) \wedge \Box(\neg(c2 \wedge y)) \wedge \Box(\neg(x \wedge y))\end{aligned} \tag{$\mathcal{T}$2-Exclusiveness}$$

$$\begin{aligned}&\Box(a \rightarrow \bigcirc(\neg a \ \mathcal{U} \ x)) \wedge \Box(a \rightarrow \bigcirc(\neg a \ \mathcal{U} \ y)) \wedge \Box(b \rightarrow \bigcirc(\neg b \ \mathcal{U} \ x))\\ &\wedge \Box(b \rightarrow \bigcirc(\neg b \ \mathcal{U} \ y)) \wedge \Box(c1 \rightarrow \bigcirc(\neg c1 \ \mathcal{U} \ x)) \wedge \Box(c1 \rightarrow \bigcirc(\neg c1 \ \mathcal{U} \ y))\\ &\wedge \Box(c2 \rightarrow \bigcirc(\neg c2 \ \mathcal{U} \ x)) \wedge \Box(c2 \rightarrow \bigcirc(\neg c2 \ \mathcal{U} \ y))\end{aligned} \tag{$\mathcal{T}$2-MaxOne}$$

$$\begin{aligned}&\Box(c1 \rightarrow (\neg c2 \ \mathcal{U} \ x)) \wedge \Box(c1 \rightarrow (\neg c2 \ \mathcal{U} \ y)) \wedge \Box(c2 \rightarrow (\neg c1 \ \mathcal{U} \ x))\\ &\wedge \Box(c2 \rightarrow (\neg c1 \ \mathcal{U} \ y))\end{aligned} \tag{$\mathcal{T}$2-Range}$$

$$\begin{aligned}&\Box(b \rightarrow (\neg a \ \mathcal{U} \ x)) \wedge \Box(b \rightarrow (\neg a \ \mathcal{U} \ y)) \wedge \Box(c1 \rightarrow (\neg a \ \mathcal{U} \ x))\\ &\wedge \Box(c1 \rightarrow (\neg a \ \mathcal{U} \ y)) \wedge \Box(c2 \rightarrow (\neg a \ \mathcal{U} \ x)) \wedge \Box(c2 \rightarrow (\neg a \ \mathcal{U} \ y))\end{aligned} \tag{$\mathcal{T}$2-Order}$$

$$\begin{aligned}&\Box(x \rightarrow (\neg b \ \mathcal{U} \ a)) \wedge \Box(x \rightarrow (\neg c1 \ \mathcal{U} \ a)) \wedge \Box(x \rightarrow (\neg c2 \ \mathcal{U} \ a))\\ &\wedge \Box(y \rightarrow (\neg b \ \mathcal{U} \ a)) \wedge \Box(y \rightarrow (\neg c1 \ \mathcal{U} \ a)) \wedge \Box(y \rightarrow (\neg c2 \ \mathcal{U} \ a))\end{aligned} \tag{$\mathcal{T}$2-Order$\mathcal{F}'$}$$

$$\begin{aligned}&(\neg x \ \mathcal{U} \ a) \wedge (\neg x \ \mathcal{U} \ b) \wedge ((\neg x \ \mathcal{U} \ c1) \vee (\neg x \ \mathcal{U} \ c2)) \wedge\\ &(\neg y \ \mathcal{U} \ a) \wedge (\neg y \ \mathcal{U} \ b) \wedge ((\neg y \ \mathcal{U} \ c1) \vee (\neg y \ \mathcal{U} \ c2))\end{aligned} \tag{$\mathcal{T}$2-First$\mathcal{F}'$}$$

$$\begin{aligned}&\Box\Big(x \rightarrow \bigcirc(\neg x \ \mathcal{U} \ a)\Big) \wedge \Box\Big(x \rightarrow \bigcirc(\neg x \ \mathcal{U} \ b)\Big)\\ &\wedge \Box\Big(x \rightarrow \bigcirc((\neg x \ \mathcal{U} \ c1) \vee (\neg x \ \mathcal{U} \ c2))\Big) \wedge \Box\Big(y \rightarrow \bigcirc(\neg y \ \mathcal{U} \ a)\Big)\\ &\wedge \Box\Big(y \rightarrow \bigcirc(\neg y \ \mathcal{U} \ b)\Big) \wedge \Box\Big(y \rightarrow \bigcirc((\neg y \ \mathcal{U} \ c1) \vee (\neg y \ \mathcal{U} \ c2))\Big)\end{aligned} \tag{$\mathcal{T}$2-After$\mathcal{F}'$}$$

The encoding of $\Big(a < (\{b, c^{[1,2]}\}, \wedge) \Longrightarrow (\{x, y\}, \wedge) \ \mid t\Big)$ into LTL is defined by Conjunction B.14. Figure B.16 provides the LTL encoding of the property in SPOT syntax. The respective SPOT monitor is shown in Figure B.15.

$$\begin{aligned}&\mathcal{T}\text{2-Exclusiveness} \ \wedge \ \mathcal{T}\text{2-MaxOne} \ \wedge \ \mathcal{T}\text{2-Range} \ \wedge \ \mathcal{T}\text{2-Order}\\ &\wedge \ \mathcal{T}\text{2-Order}\mathcal{F}' \ \wedge \ \mathcal{T}\text{2-First}\mathcal{F}' \ \wedge \ \mathcal{T}\text{2-After}\mathcal{F}'\end{aligned} \tag{B.14}$$

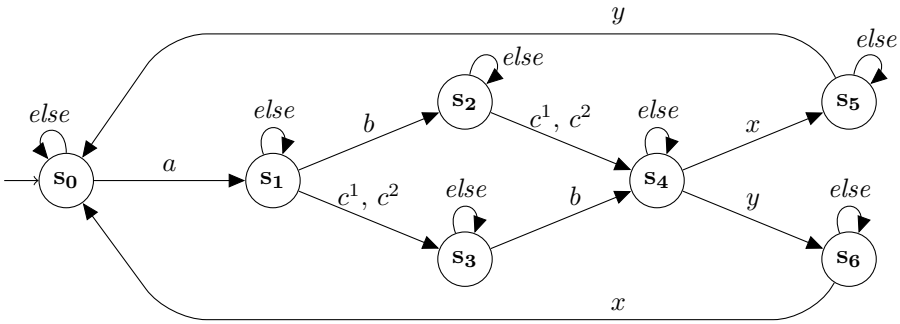


Figure B.15 The SPOT monitor of the LTL encoding of $\big(a < (\{b, c^{[1,2]}\}, \wedge) \implies (\{x, y\}, \wedge) \ \mid t\big)$. *else* stands for any name which is not of $\{a, b, c1, c2, x, y\}$. "$c1$, $c2$" means that $c1$ and $c2$ cannot occur simultaneously. Transitions which are not defined are forbidden.

```
~ T2-Exclusiveness
G(!(a && b)) && G(!(a && c1)) && G(!(a && c2)) && G(!(a && x)) &&
G(!(a && y)) &&
G(!(b && c1)) && G(!(b && c2)) && G(!(b && x)) && G(!(b && y)) &&
G(!(c1 && c2)) && G(!(c1 && x)) && G(!(c1 && y)) &&
G(!(c2 && x)) && G(!(c2 && y)) && G(!(x && y)) &&

~ T2-MaxOne
G(a-> X(!a U x)) && G(a-> X(!a U y)) && G(b-> X(!b U x)) &&
G(b-> X(!b U y)) && G(c1-> X(!c1 U x)) && G(c1-> X(!c1 U y)) &&
G(c2-> X(!c2 U x)) && G(c2-> X(!c2 U y)) &&

~ T2-Range
G(c1-> (!c2 U x)) && G(c1-> (!c2 U y)) &&
G(c2-> (!c1 U x)) && G(c2-> (!c1 U y)) &&

~ T2-Order
G(b-> (!a U x)) && G(b-> (!a U y)) && G(c1-> (!a U x)) &&
G(c1-> (!a U y)) && G(c2-> (!a U x)) && G(c2-> (!a U y)) &&

~ T2-OrderF'
G(x-> (!b U a)) && G(x->(!c1 U a)) && G(x->(!c2 U a)) &&
G(y-> (!b U a)) && G(y->(!c1 U a)) && G(y->(!c2 U a)) &&

~ T2-FirstF'
(!x U a) && (!x U b) && ((!x U c1) || (!x U c2))&&
(!y U a) && (!y U b) && ((!y U c1) || (!y U c2))&&

~ T2-AfterF'
G(x-> X(!x U a)) && G(x-> X(!x U b)) && G(x-> X((!x U c1) || (!x U c2)))&&
G(y-> X(!y U a)) && G(y-> X(!y U b)) && G(y-> X((!y U c1) || (!y U c2)))
```

Figure B.16 The LTL encoding of $\big(a < (\{b, c^{[1,2]}\}, \wedge) \implies (\{x, y\}, \wedge) \ \mid t\big)$ in SPOT syntax.

B.2.3 Encoding of $\Big(a \Longrightarrow \big(\{b,c\},\wedge\big) < \big(\{x, y^{[1,2]}\},\wedge\big) \ \mid t\Big)$ into LTL

Consider the timed implication constraint $\Big(a \Longrightarrow \big(\{b,c\},\wedge\big) < \big(\{x, y^{[1,2]}\},\wedge\big) \ \mid t\Big)$. After the removal of the range $y^{[1,2]}$, the new vocabulary is the set $\{a,b,c,x,y1,y2\}$. The LTL encoding is the conjunction of the following LTL formulas:

$$\begin{aligned}&\Box\big(\neg(a\wedge b)\big)\wedge\Box\big(\neg(a\wedge c)\big)\wedge\Box\big(\neg(a\wedge x)\big)\wedge\Box\big(\neg(a\wedge y1)\big)\\&\wedge\Box\big(\neg(a\wedge y2)\big)\wedge\Box\big(\neg(b\wedge c)\big)\wedge\Box\big(\neg(b\wedge x)\big)\wedge\Box\big(\neg(b\wedge y1)\big)\\&\wedge\Box\big(\neg(b\wedge y2)\big)\wedge\Box\big(\neg(c\wedge x)\big)\wedge\Box\big(\neg(c\wedge y1)\big)\wedge\Box\big(\neg(c\wedge y2)\big)\\&\wedge\Box\big(\neg(x\wedge y1)\big)\wedge\Box\big(\neg(x\wedge y2)\big)\wedge\Box\big(\neg(y1\wedge y2)\big)\end{aligned} \tag{$\mathcal{T}$3-Exclusiveness}$$

$$\begin{aligned}&\Box\Big(a\rightarrow\bigcirc(\neg a\,\mathcal{U}\,x)Big)\wedge\Box\Big(a\rightarrow\bigcirc\big((\neg a\,\mathcal{U}\,y1)\vee(\neg a\,\mathcal{U}\,y2)\big)\Big)\\&\wedge\Box\Big(b\rightarrow\bigcirc(\neg b\,\mathcal{U}\,x)\Big)\wedge\Box\Big(b\rightarrow\bigcirc\big((\neg b\,\mathcal{U}\,y1)\vee(\neg b\,\mathcal{U}\,y2)\big)\Big)\\&\wedge\Box\Big(c\rightarrow\bigcirc(\neg c\,\mathcal{U}\,x)\Big)\wedge\Box\Big(c\rightarrow\bigcirc\big((\neg c\,\mathcal{U}\,y1)\vee(\neg c\,\mathcal{U}\,y2)\big)\Big)\end{aligned} \tag{$\mathcal{T}$3-MaxOne}$$

$$\begin{aligned}&\Box\Big(b\rightarrow(\neg a\,\mathcal{U}\,x)\Big)\wedge\Box\Big(b\rightarrow\big((\neg a\,\mathcal{U}\,y1)\vee(\neg a\,\mathcal{U}\,y2)\big)\Big)\\&\wedge\Box\Big(c\rightarrow(\neg a\,\mathcal{U}\,x)\Big)\wedge\Box\Big(c\rightarrow\big((\neg a\,\mathcal{U}\,y1)\vee(\neg a\,\mathcal{U}\,y2)\big)\Big)\end{aligned} \tag{$\mathcal{T}$3-Order}$$

$$\begin{aligned}&\Box\big(y1\rightarrow(\neg y2\,\mathcal{U}\,a)\big)\wedge\Box\big(y1\rightarrow(\neg y2\,\mathcal{U}\,b)\big)\wedge\Box\big(y1\rightarrow\bigcirc(\neg y2\,\mathcal{U}\,c)\big)\\&\wedge\Box\big(y2\rightarrow(\neg y1\,\mathcal{U}\,a)\big)\wedge\Box\big(y2\rightarrow(\neg y1\,\mathcal{U}\,b)\big)\wedge\Box\big(y2\rightarrow\bigcirc(\neg y1\,\mathcal{U}\,c)\big)\end{aligned} \tag{$\mathcal{T}$3-Range$\mathcal{F}'$}$$

$$\begin{aligned}&\Box\big(x\rightarrow(\neg b\,\mathcal{U}\,a)\big)\wedge\Box\big(x\rightarrow(\neg c\,\mathcal{U}\,a)\big)\wedge\Box\big(y1\rightarrow(\neg b\,\mathcal{U}\,a)\big)\\&\wedge\Box\big(y1\rightarrow(\neg c\,\mathcal{U}\,a)\big)\wedge\Box\big(y2\rightarrow(\neg b\,\mathcal{U}\,a)\big)\wedge\Box\big(y2\rightarrow(\neg c\,\mathcal{U}\,a)\big)\end{aligned} \tag{$\mathcal{T}$3-Order$\mathcal{F}'$}$$

$$\begin{aligned}&(\neg x\,\mathcal{U}\,a)\wedge(\neg x\,\mathcal{U}\,b)\wedge(\neg x\,\mathcal{U}\,c)\wedge(\neg y1\,\mathcal{U}\,a)\wedge(\neg y1\,\mathcal{U}\,b)\wedge(\neg y1\,\mathcal{U}\,c)\\&\wedge(\neg y2\,\mathcal{U}\,a)\wedge(\neg y2\,\mathcal{U}\,b)\wedge(\neg y2\,\mathcal{U}\,c)\end{aligned} \tag{$\mathcal{T}$3-First$\mathcal{F}'$}$$

$$\begin{aligned}&\Box\big(x\rightarrow\bigcirc(\neg x\,\mathcal{U}\,a)\big)\wedge\Box\big(x\rightarrow\bigcirc(\neg x\,\mathcal{U}\,b)\big)\wedge\Box\big(x\rightarrow\bigcirc(\neg x\,\mathcal{U}\,c)\big)\\&\wedge\Box\big(y1\rightarrow\bigcirc(\neg y1\,\mathcal{U}\,a)\big)\wedge\Box\big(y1\rightarrow\bigcirc(\neg y1\,\mathcal{U}\,b)\big)\wedge\Box\big(y1\rightarrow\bigcirc(\neg y1\,\mathcal{U}\,c)\big)\\&\wedge\Box\big(y2\rightarrow\bigcirc(\neg y2\,\mathcal{U}\,a)\big)\wedge\Box\big(y2\rightarrow\bigcirc(\neg y2\,\mathcal{U}\,b)\big)\wedge\Box\big(y2\rightarrow\bigcirc(\neg y2\,\mathcal{U}\,c)\big)\end{aligned} \tag{$\mathcal{T}$3-After$\mathcal{F}'$}$$

The LTL encoding of $\Big(a \Longrightarrow \big(\{b,c\},\wedge\big) < \big(\{x, y^{[1,2]}\},\wedge\big) \ \mid t\Big)$ is defined by Conjunction B.15. Figures B.18 provides the LTL encoding of the property in SPOT syntax. Figure B.17 shows the corresponding SPOT monitor.

$$\begin{aligned}&\mathcal{T}\text{3-Exclusiveness}\ \wedge\ \mathcal{T}\text{3-MaxOne}\ \wedge\ \mathcal{T}\text{3-Order}\ \wedge\ \mathcal{T}\text{3-Range}\mathcal{F}'\\&\wedge\ \mathcal{T}\text{3-Order}\mathcal{F}'\ \wedge\ \mathcal{T}\text{3-First}\mathcal{F}'\ \wedge\ \mathcal{T}\text{3-After}\mathcal{F}'\end{aligned} \tag{B.15}$$

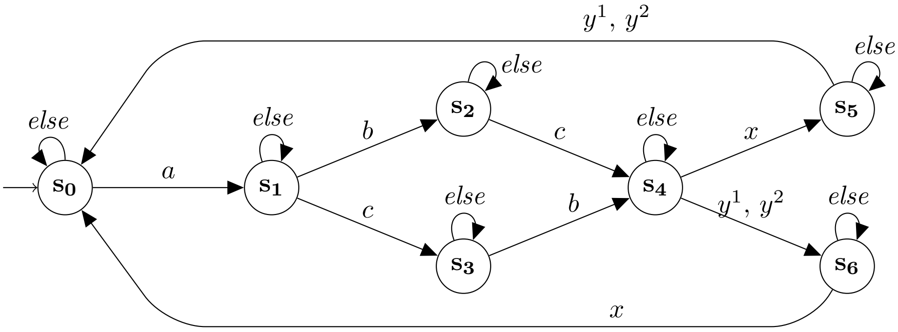


Figure B.17 The SPOT monitor of the LTL encoding of $\big(a \Longrightarrow (\{b, c\}, \wedge) < (\{x, y^{[1,2]}\}, \wedge) \;\mid t\big)$. *else* stands for any name which is not of $\{a, b, c, x, y1, y2\}$. "$y1$, $y2$" means that $y1$ and $y2$ cannot occur simultaneously. Transitions which are not defined are forbidden.

```
~ T3-Exclusiveness
G(!(a && b)) && G(!(a && c)) && G(!(a && x)) && G(!(a && y1)) &&
G(!(a && y2)) && G(!(b && c)) && G(!(b && x)) && G(!(b && y1)) &&
G(!(b && y2)) && G(!(c && x)) && G(!(c && y1)) && G(!(c && y2)) &&
G(!(x && y1)) && G(!(x && y2)) && G(!(y1 && y2)) &&

~ T3-MaxOne
G(a-> X(!a U x)) && G(a-> X((!a U y1) || (!a U y2))) &&
G(b-> X(!b U x)) && G(b-> X((!b U y1) || (!b U y2))) &&
G(c-> X(!c U x)) && G(c-> X((!c U y1) || (!c U y2))) &&

~ T3-Order
G(b-> (!a U x)) && G(b-> ((!a U y1) || (!a U y2))) &&
G(c-> (!a U x)) && G(c-> ((!a U y1) || (!a U y2))) &&

~ T3-RangeF'
G(y1 -> (!y2 U a)) && G(y1 -> (!y2 U b)) && G(y1 -> X(!y2 U c)) &&
G(y2 -> (!y1 U a)) && G(y2 -> (!y1 U b)) && G(y2 -> X(!y1 U c)) &&

~ T3-OrderF'
G(x-> (!b U a)) && G(x -> (!c U a)) && G(y1 -> (!b U a)) &&
G(y1 -> (!c U a)) && G(y2 -> (!b U a)) && G(y2 -> (!c U a)) &&

~ T3-FirstF'
(!x U a) && (!x U b) && (!x U c) && (!y1 U a) && (!y1 U b) && (!y1 U c) &&
(!y2 U a) && (!y2 U b) && (!y2 U c) &&

~ T3-AfterF'
G(x -> X(!x U a)) && G(x -> X(!x U b)) && G(x -> X(!x U c)) &&
G(y1 -> X(!y1 U a)) && G(y1 -> X(!y1 U b)) && G(y1 -> X(!y1 U c)) &&
G(y2 -> X(!y2 U a)) && G(y2 -> X(!y2 U b)) && G(y2 -> X(!y2 U c))
```

Figure B.18 The LTL encoding of $\big(a \Longrightarrow (\{b, c\}, \wedge) < (\{x, y^{[1,2]}\}, \wedge) \;\mid t\big)$ in SPOT syntax.

B.2.4 Encoding of $\Big(a < (\{b,c\},\vee) \Longrightarrow (\{x,y\},\wedge) \ \mid t\Big)$ into LTL

Consider the timed implication constraint $\Big(a < (\{b,c\},\vee) \Longrightarrow (\{x,y\},\wedge) \ \mid t\Big)$. The LTL encoding of the property is defined by Conjunction B.16. Figure B.20 provides the LTL encoding of the property in SPOT syntax. Figure B.19 shows the respective SPOT monitor.

$$\mathcal{T}4\text{-Exclusiveness} \wedge \mathcal{T}4\text{-MaxOne} \wedge \mathcal{T}4\text{-Order} \wedge \mathcal{T}4\text{-Order}\mathcal{F}' \wedge \mathcal{T}4\text{-First}\mathcal{F}' \wedge \mathcal{T}4\text{-After}\mathcal{F}' \tag{B.16}$$

$$\Box(\neg(a\wedge b)) \wedge \Box(\neg(a\wedge c)) \wedge \Box(\neg(a\wedge x)) \wedge \Box(\neg(a\wedge y)) \wedge \Box(\neg(b\wedge c)) \wedge \Box(\neg(b\wedge x)) \wedge \Box(\neg(b\wedge y)) \wedge \Box(\neg(c\wedge x)) \wedge \Box(\neg(c\wedge y)) \wedge \Box(\neg(x\wedge y)) \tag{$\mathcal{T}4$-Exclusiveness}$$

$$\Box(a \to \bigcirc(\neg a\ \mathcal{U}\ x)) \wedge \Box(a \to \bigcirc(\neg a\ \mathcal{U}\ y)) \wedge \Box(b \to \bigcirc(\neg b\ \mathcal{U}\ x)) \wedge \Box(b \to \bigcirc(\neg b\ \mathcal{U}\ y)) \wedge \Box(c \to \bigcirc(\neg c\ \mathcal{U}\ x)) \wedge \Box(c \to \bigcirc(\neg c\ \mathcal{U}\ y)) \tag{$\mathcal{T}4$-MaxOne}$$

$$\Box(b \to (\neg a\ \mathcal{U}\ x)) \wedge \Box(b \to (\neg a\ \mathcal{U}\ y)) \wedge \Box(c \to (\neg a\ \mathcal{U}\ x)) \wedge \Box(c \to (\neg a\ \mathcal{U}\ y)) \tag{$\mathcal{T}4$-Order}$$

$$\Box(x \to (\neg b\ \mathcal{U}\ a)) \wedge \Box(x \to (\neg c\ \mathcal{U}\ a)) \wedge \Box(y \to (\neg b\ \mathcal{U}\ a)) \wedge \Box(y \to (\neg c\ \mathcal{U}\ a)) \tag{$\mathcal{T}4$-Order$\mathcal{F}'$}$$

$$(\neg x\ \mathcal{U}\ a) \wedge ((\neg x\ \mathcal{U}\ b) \vee (\neg x\ \mathcal{U}\ c)) \wedge (\neg y\ \mathcal{U}\ a) \wedge ((\neg y\ \mathcal{U}\ b) \vee (\neg y\ \mathcal{U}\ c)) \tag{$\mathcal{T}4$-First$\mathcal{F}'$}$$

$$\Box\Big(x \to \bigcirc(\neg x\ \mathcal{U}\ a)\Big) \wedge \Box\Big(x \to \bigcirc\big((\neg x\ \mathcal{U}\ b) \vee (\neg x\ \mathcal{U}\ c)\big)\Big) \wedge \Box\Big(y \to \bigcirc(\neg y\ \mathcal{U}\ a)\Big) \wedge \Box\Big(y \to \bigcirc\big((\neg y\ \mathcal{U}\ b) \vee (\neg y\ \mathcal{U}\ c)\big)\Big) \tag{$\mathcal{T}4$-After$\mathcal{F}'$}$$

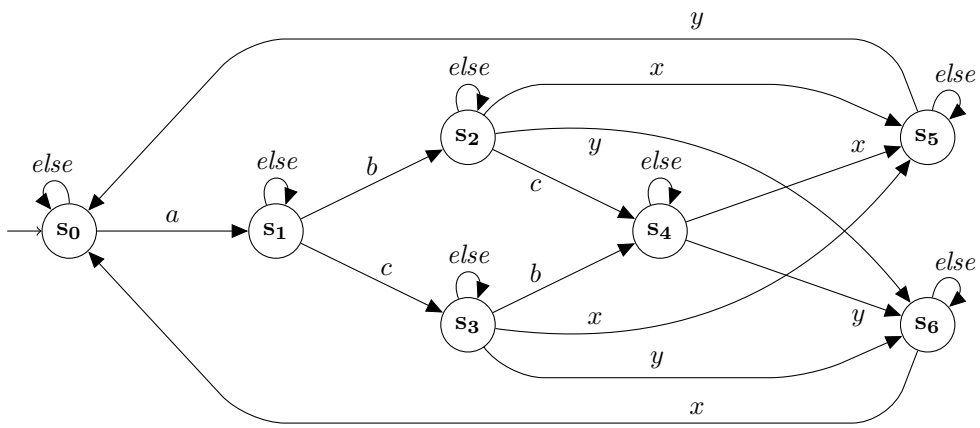


Figure B.19 The SPOT monitor for the LTL encoding of $\big(a < (\{b,c\},\vee) \Longrightarrow (\{x,y\},\wedge) \ \mid t\big)$. *else* stands for any name which is not of $\{a,b,c,x,y\}$. Transitions which are not defined are forbidden.

```
~ T4-Exclusiveness
G(!(a && b)) && G(!(a && c)) && G(!(a && x)) && G(!(a && y)) &&
G(!(b && c)) && G(!(b && x)) && G(!(b && y)) && G(!(c && x)) &&
G(!(c && y)) && G(!(x && y)) &&

~ T4-MaxOne
G(a-> X(!a U x)) && G(a-> X(!a U y)) && G(b-> X(!b U x)) &&
G(b-> X(!b U y)) && G(c-> X(!c U x)) && G(c-> X(!c U y)) &&

~ T4-Order
G(b-> (!a U x)) && G(b-> (!a U y)) &&
G(c-> (!a U x)) && G(c-> (!a U y)) &&

~ T4-OrderF'
G(x-> (!b U a)) && G(x-> (!c U a)) &&
G(y-> (!b U a)) && G(y-> (!c U a)) &&

~ T4-FirstF'
(!x U a) && ((!x U b) || (!x U c)) &&
(!y U a) && ((!y U b) || (!y U c)) &&

~ T4-AfterF'
G(x -> X(!x U a)) && G(x -> X((!x U b) || (!x U c))) &&
G(y -> X(!y U a)) && G(y -> X((!y U b) || (!y U c)))
```

Figure B.20 The LTL encoding of $\big(a < (\{b, c\}, \vee) \Longrightarrow (\{x, y\}, \wedge) \ \mid t\big)$ in SPOT syntax.

B.2.5 Encoding of $\Big(a < \big(\{b, c^{[1,2]}\}, \vee\big) \Longrightarrow \big(\{x, y\}, \wedge\big) \ \mid t\Big)$ into LTL

Consider the timed implication constraint $\Big(a < \big(\{b, c^{[1,2]}\}, \vee\big) \Longrightarrow \big(\{x, y\}, \wedge\big) \ \mid t\Big)$. After the removal of the range $c^{[1,2]}$, the new vocabulary is the set $\{a, b, c1, c2, x, y\}$. The LTL encoding of the property consists of the following components:

$$\begin{aligned}&\Box\big(\neg(a \wedge b)\big) \wedge \Box\big(\neg(a \wedge c1)\big) \wedge \Box\big(\neg(a \wedge c2)\big) \wedge \Box\big(\neg(a \wedge x)\big)\\ &\wedge \Box\big(\neg(a \wedge y)\big) \wedge \Box\big(\neg(b \wedge c1)\big) \wedge \Box\big(\neg(b \wedge c2)\big) \wedge \Box\big(\neg(b \wedge x)\big)\\ &\wedge \Box\big(\neg(b \wedge y)\big) \wedge \Box\big(\neg(c1 \wedge c2)\big) \wedge \Box\big(\neg(c1 \wedge x)\big) \wedge \Box\big(\neg(c1 \wedge y)\big)\\ &\wedge \Box\big(\neg(c2 \wedge x)\big) \wedge \Box\big(\neg(c2 \wedge y)\big) \wedge \Box\big(\neg(x \wedge y)\big)\end{aligned} \tag{$\mathcal{T}$5-Exclusiveness}$$

$$\begin{aligned}&\Box\big(a \rightarrow \bigcirc(\neg a \,\mathcal{U}\, x)\big) \wedge \Box\big(a \rightarrow \bigcirc(\neg a \,\mathcal{U}\, y)\big) \wedge \Box\big(b \rightarrow \bigcirc(\neg b \,\mathcal{U}\, x)\big)\\ &\wedge \Box\big(b \rightarrow \bigcirc(\neg b \,\mathcal{U}\, y)\big) \wedge \Box\big(c1 \rightarrow \bigcirc(\neg c1 \,\mathcal{U}\, x)\big) \wedge \Box\big(c1 \rightarrow \bigcirc(\neg c1 \,\mathcal{U}\, y)\big)\\ &\wedge \Box\big(c2 \rightarrow \bigcirc(\neg c2 \,\mathcal{U}\, x)\big) \wedge \Box\big(c2 \rightarrow \bigcirc(\neg c2 \,\mathcal{U}\, y)\big)\end{aligned} \tag{$\mathcal{T}$5-MaxOne}$$

$$\begin{aligned}&\Box\big(c1 \rightarrow (\neg c2 \,\mathcal{U}\, x)\big) \wedge \Box\big(c1 \rightarrow (\neg c2 \,\mathcal{U}\, y)\big) \wedge \Box\big(c2 \rightarrow (\neg c1 \,\mathcal{U}\, x)\big)\\ &\wedge \Box\big(c2 \rightarrow (\neg c1 \,\mathcal{U}\, y)\big)\end{aligned} \tag{$\mathcal{T}$5-Range}$$

$$\begin{aligned}&\Box\big(b \rightarrow (\neg a \,\mathcal{U}\, x)\big) \wedge \Box\big(b \rightarrow (\neg a \,\mathcal{U}\, y)\big) \wedge \Box\big(c1 \rightarrow (\neg a \,\mathcal{U}\, x)\big)\\ &\wedge \Box\big(c1 \rightarrow (\neg a \,\mathcal{U}\, y)\big) \wedge \Box\big(c2 \rightarrow (\neg a \,\mathcal{U}\, x)\big) \wedge \Box\big(c2 \rightarrow ((\neg a \,\mathcal{U}\, y)\big)\end{aligned} \tag{$\mathcal{T}$5-Order}$$

$$\begin{aligned}&\Box\big(x \rightarrow (\neg b \,\mathcal{U}\, a)\big) \wedge \Box\big(x \rightarrow (\neg c1 \,\mathcal{U}\, a)\big) \wedge \Box\big(x \rightarrow (\neg c2 \,\mathcal{U}\, a)\big)\\ &\wedge \Box\big(y \rightarrow (\neg b \,\mathcal{U}\, a)\big) \wedge \Box\big(y \rightarrow (\neg c1 \,\mathcal{U}\, a)\big) \wedge \Box\big(y \rightarrow (\neg c2 \,\mathcal{U}\, a)\big)\end{aligned} \tag{$\mathcal{T}$5-Order$\mathcal{F}'$}$$

$$\begin{aligned}&\big(\neg x \,\mathcal{U}\, a\big) \wedge \big((\neg x \,\mathcal{U}\, b) \vee (\neg x \,\mathcal{U}\, c1) \vee (\neg x \,\mathcal{U}\, c2)\big)\\ &\wedge \big(\neg y \,\mathcal{U}\, a\big) \wedge \big((\neg y \,\mathcal{U}\, b) \vee (\neg y \,\mathcal{U}\, c1) \vee (\neg y \,\mathcal{U}\, c2)\big)\end{aligned} \tag{$\mathcal{T}$5-First$\mathcal{F}'$}$$

$$\begin{aligned}&\Box\Big(x \rightarrow \bigcirc(\neg x \,\mathcal{U}\, a)\Big) \wedge \Box\Big(x \rightarrow \bigcirc\big((\neg x \,\mathcal{U}\, b) \vee (\neg x \,\mathcal{U}\, c1) \vee (\neg x \,\mathcal{U}\, c2)\big)\Big)\\ &\wedge \Box\Big(y \rightarrow \bigcirc(\neg y \,\mathcal{U}\, a)\Big) \wedge \Box\Big(y \rightarrow \bigcirc\big((\neg y \,\mathcal{U}\, b) \vee (\neg y \,\mathcal{U}\, c1) \vee (\neg y \,\mathcal{U}\, c2)\big)\Big)\end{aligned} \tag{$\mathcal{T}$5-After$\mathcal{F}'$}$$

The LTL encoding of $\Big(a < \big(\{b, c^{[1,2]}\}, \vee\big) \Longrightarrow \big(\{x, y\}, \wedge\big) \ \mid t\Big)$ is defined by Conjunction B.17. Figure B.22 provides the LTL encoding of the property in SPOT syntax. The corresponding SPOT monitor is shown in Figure B.21.

$$\begin{aligned}&\mathcal{T}\text{5-Exclusiveness} \ \wedge \ \mathcal{T}\text{5-MaxOne} \ \wedge \ \mathcal{T}\text{5-Range} \ \wedge \ \mathcal{T}\text{5-Order}\\ &\wedge \ \mathcal{T}\text{5-Order}\mathcal{F}' \ \wedge \ \mathcal{T}\text{5-First}\mathcal{F}' \ \wedge \ \mathcal{T}\text{5-After}\mathcal{F}'\end{aligned} \tag{B.17}$$

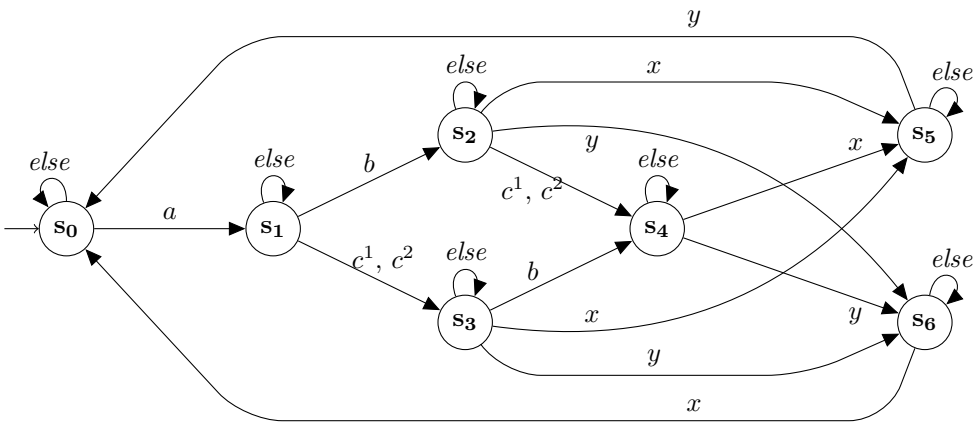


Figure B.21 The SPOT monitor for the LTL encoding of $\big(a < (\{b, c^{[1,2]}\}, \vee) \Longrightarrow (\{x, y\}, \wedge) \ \mid t\big)$. *else* stands for any name which is not of $\{a, b, c1, c2, x, y\}$. "$c1$, $c2$" means that $c1$ and $c2$ cannot occur simultaneously. Transitions which are not defined are forbidden.

```
~ T5-Exclusiveness
G(!(a && b)) && G(!(a && c1)) && G(!(a && c2)) && G(!(a && x)) &&
G(!(a && y)) && G(!(b && c1)) && G(!(b && c2)) && G(!(b && x)) &&
G(!(b && y)) && G(!(c1 && c2)) && G(!(c1 && x)) && G(!(c1 && y)) &&
G(!(c2 && x)) && G(!(c2 && y)) && G(!(x && y)) &&

~ T5-MaxOne
G(a-> X(!a U x))  && G(a-> X(!a U y))   && G(b-> X(!b U x))   &&
G(b-> X(!b U y))  && G(c1-> X(!c1 U x)) && G(c1-> X(!c1 U y)) &&
G(c2-> X(!c2 U x))&& G(c2-> X(!c2 U y)) &&

~ T5-Range
G(c1-> (!c2 U x)) && G(c1-> (!c2 U y)) &&
G(c2-> (!c1 U x)) && G(c2-> (!c1 U y)) &&

~ T5-Order
G(b-> (!a U x))  && G(b-> (!a U y))  && G(c1-> (!a U x))  &&
G(c1-> (!a U y)) && G(c2-> (!a U x)) && G(c2-> ((!a U y)) &&

~ T5-OrderF'
G(x-> (!b U a)) && G(x -> (!c1 U a)) && G(x -> (!c2 U a)) &&
G(y-> (!b U a)) && G(y -> (!c1 U a)) && G(y -> (!c2 U a)) &&

~ T5-FirstF'
(!x U a) && ((!x U b) || (!x U c1) || (!x U c2)) &&
(!y U a) && ((!y U b) || (!y U c1) || (!y U c2)) &&

~ T5-AfterF'
G(x -> X(!x U a)) && G(x -> X((!x U b) || (!x U c1) || (!x U c2))) &&
G(y -> X(!y U a)) && G(y -> X((!y U b) || (!y U c1) || (!y U c2)))
```

Figure B.22 The LTL encoding of $\big(a < (\{b, c^{[1,2]}\}, \vee) \Longrightarrow (\{x, y\}, \wedge) \ \mid t\big)$ in SPOT syntax.

B.2.6 Encoding of $\Big(\big(\{a,b\},\wedge\big) < c \Longrightarrow \big(\{x, y^{[1,2]}\},\wedge\big) \ \mid t\Big)$

Consider the timed implication constraint $\Big(\big(\{a,b\},\wedge\big) < c \Longrightarrow \big(\{x, y^{[1,2]}\},\wedge\big) \ \mid t\Big)$. After the removal of the range $y^{[1,2]}$, the new vocabulary is the set $\{a, b, c, x, y1, y2\}$. The LTL encoding of the property is defined by Conjunction B.18. Figure B.24 provides the LTL encoding of the property in SPOT syntax. The respective SPOT monitor is shown in Figure B.23.

$$\begin{aligned}&\mathcal{T}6\text{-Exclusiveness} \ \wedge \ \mathcal{T}6\text{-MaxOne} \ \wedge \ \mathcal{T}6\text{-Order} \ \wedge \ \mathcal{T}6\text{-Range}\mathcal{F}' \\ &\wedge \ \mathcal{T}6\text{-Order}\mathcal{F}' \ \wedge \mathcal{T}6\text{-First}\mathcal{F}' \ \wedge \ \mathcal{T}6\text{-After}\mathcal{F}'\end{aligned} \tag{B.18}$$

$$\begin{aligned}&\Box\big(\neg(a\wedge b)\big) \wedge \Box\big(\neg(a\wedge c)\big) \wedge \Box\big(\neg(a\wedge x)\big) \wedge \Box\big(\neg(a\wedge y1)\big) \\ &\wedge \Box\big(\neg(a\wedge y2)\big) \wedge \Box\big(\neg(b\wedge c)\big) \wedge \Box\big(\neg(b\wedge x)\big) \wedge \Box\big(\neg(b\wedge y1)\big) \\ &\wedge \Box\big(\neg(b\wedge y2)\big) \wedge \Box\big(\neg(c\wedge x)\big) \wedge \Box\big(\neg(c\wedge y1)\big) \wedge \Box\big(\neg(c\wedge y2)\big) \\ &\wedge \Box\big(\neg(x\wedge y1)\big) \wedge \Box\big(\neg(x\wedge y2)\big) \wedge \Box\big(\neg(y1\wedge y2)\big)\end{aligned} \tag{$\mathcal{T}6$-Exclusiveness}$$

$$\begin{aligned}&\Box\Big(a \rightarrow \bigcirc(\neg a \ \mathcal{U} \ x)\Big) \wedge \Box\Big(a \rightarrow \bigcirc\big((\neg a \ \mathcal{U} \ y1) \vee (\neg a \ \mathcal{U} \ y2)\big)\Big) \\ &\wedge \Box\Big(b \rightarrow \bigcirc(\neg b \ \mathcal{U} \ x)\Big) \wedge \Box\Big(b \rightarrow \bigcirc\big((\neg b \ \mathcal{U} \ y1) \vee (\neg b \ \mathcal{U} \ y2)\big)\Big) \\ &\wedge \Box\Big(c \rightarrow \bigcirc(\neg c \ \mathcal{U} \ x)\Big) \wedge \Box\Big(c \rightarrow \bigcirc\big((\neg c \ \mathcal{U} \ y1) \vee (\neg c \ \mathcal{U} \ y2)\big)\Big)\end{aligned} \tag{$\mathcal{T}6$-MaxOne}$$

$$\begin{aligned}&\Box\Big(c \rightarrow (\neg a \ \mathcal{U} \ x)\Big) \wedge \Box\Big(c \rightarrow \big((\neg a \ \mathcal{U} \ y1) \vee (\neg a \ \mathcal{U} \ y2)\big)\Big) \\ &\wedge \Box\Big(c \rightarrow (\neg b \ \mathcal{U} \ x)\Big) \wedge \Box\Big(c \rightarrow \big((\neg b \ \mathcal{U} \ y1) \vee (\neg b \ \mathcal{U} \ y2)\big)\Big)\end{aligned} \tag{$\mathcal{T}6$-Order}$$

$$\begin{aligned}&\Box\Big(y1 \rightarrow (\neg y2 \ \mathcal{U} \ a)\Big) \wedge \Box\Big(y1 \rightarrow \big((\neg y2 \ \mathcal{U} \ b) \vee (\neg y2 \ \mathcal{U} \ c)\big)\Big) \\ &\wedge \Box\Big(y2 \rightarrow (\neg y1 \ \mathcal{U} \ a)\Big) \wedge \Box\Big(y2 \rightarrow \big((\neg y1 \ \mathcal{U} \ b) \vee (\neg y1 \ \mathcal{U} \ c)\big)\Big)\end{aligned} \tag{$\mathcal{T}6$-Range$\mathcal{F}'$}$$

$$\begin{aligned}&\Box\big(x \rightarrow (\neg c \ \mathcal{U} \ a)\big) \wedge \Box\big(x \rightarrow (\neg c \ \mathcal{U} \ b)\big) \wedge \Box\big(y1 \rightarrow (\neg c \ \mathcal{U} \ a)\big) \\ &\wedge \Box\big(y1 \rightarrow (\neg c \ \mathcal{U} \ b)\big) \wedge \Box\big(y2 \rightarrow (\neg c \ \mathcal{U} \ a)\big) \wedge \Box\big(y2 \rightarrow (\neg c \ \mathcal{U} \ b)\big)\end{aligned} \tag{$\mathcal{T}6$-Order$\mathcal{F}'$}$$

$$\begin{aligned}&(\neg x \ \mathcal{U} \ a) \wedge (\neg x \ \mathcal{U} \ b) \wedge (\neg x \ \mathcal{U} \ c) \wedge (\neg y1 \ \mathcal{U} \ a) \wedge (\neg y1 \ \mathcal{U} \ b) \wedge (\neg y1 \ \mathcal{U} \ c) \\ &\wedge (\neg y2 \ \mathcal{U} \ a) \wedge (\neg y2 \ \mathcal{U} \ b) \wedge (\neg y2 \ \mathcal{U} \ c)\end{aligned} \tag{$\mathcal{T}6$-First$\mathcal{F}'$}$$

$$\begin{aligned}&\Box\big(x \rightarrow \bigcirc(\neg x \ \mathcal{U} \ a)\big) \wedge \Box\big(x \rightarrow \bigcirc(\neg x \ \mathcal{U} \ b)\big) \wedge \Box\big(x \rightarrow \bigcirc(\neg x \ \mathcal{U} \ c)\big) \\ &\wedge \Box\big(y1 \rightarrow \bigcirc(\neg y1 \ \mathcal{U} \ a)\big) \wedge \Box\big(y1 \rightarrow \bigcirc(\neg y1 \ \mathcal{U} \ b)\big) \wedge \Box\big(y1 \rightarrow \bigcirc(\neg y1 \ \mathcal{U} \ c)\big) \\ &\wedge \Box\big(y2 \rightarrow \bigcirc(\neg y2 \ \mathcal{U} \ a)\big) \wedge \Box\big(y2 \rightarrow \bigcirc(\neg y2 \ \mathcal{U} \ b)\big) \wedge \Box\big(y2 \rightarrow \bigcirc(\neg y2 \ \mathcal{U} \ c)\big)\end{aligned} \tag{$\mathcal{T}6$-After$\mathcal{F}'$}$$

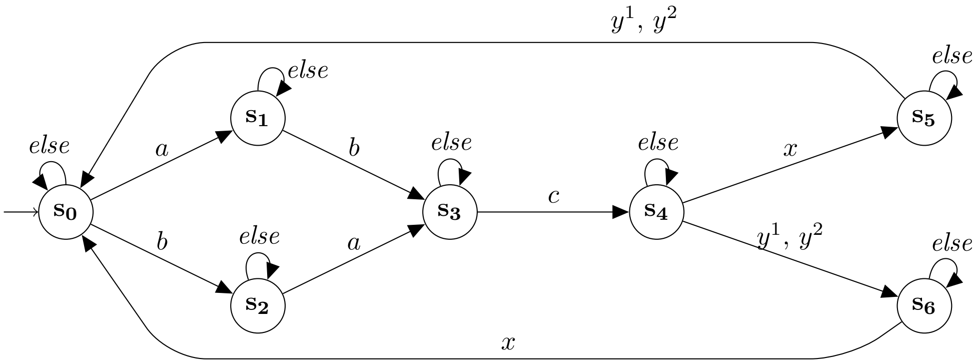


Figure B.23 The SPOT monitor for the LTL encoding of $\big((\{a,b\},\wedge) < c \Longrightarrow (\{x, y^{[1,2]}\},\wedge)\ \mid t\big)$. *else* stands for any name which is not of $\{a,b,c,x,y1,y2\}$. "$y1,\ y2$" means that $y1$ and $y2$ cannot occur simultaneously. Transitions which are not defined are forbidden.

```
~ T6-Exclusiveness
G(!(a && b)) && G(!(a && c)) && G(!(a && x)) && G(!(a && y1)) &&
G(!(a && y2)) && G(!(b && c)) && G(!(b && x)) && G(!(b && y1)) &&
G(!(b && y2)) && G(!(c && x)) && G(!(c && y1)) && G(!(c && y2)) &&
G(!(x && y1)) && G(!(x && y2)) && G(!(y1 && y2))

~ T6-MaxOne
G(a-> X(!a U x)) && G(a-> X((!a U y1) || (!a U y2))) &&
G(b-> X(!b U x)) && G(b-> X((!b U y1) || (!b U y2))) &&
G(c-> X(!c U x)) && G(c-> X((!c U y1) || (!c U y2))) &&

~ T6-Order
G(c-> (!a U x)) && G(c-> ((!a U y1) || (!a U y2))) &&
G(c-> (!b U x)) && G(c-> ((!b U y1) || (!b U y2))) &&

~ T6-RangeF'
G(y1 -> (!y2 U a)) && G(y1 -> (!y2 U b)) && G(y1 -> (!y2 U c)) &&
G(y2 -> (!y1 U a)) && G(y2 -> (!y1 U b)) && G(y2 -> (!y1 U c)) &&

~ T6-OrderF'
G(x-> (!c U a))   && G(x-> (!c U b))   &&  G(y1 -> (!c U a))&&
G(y1 -> (!c U b)) && G(y2 -> (!c U a)) && G(y2 -> (!c U b)) &&

~ T6-FirstF'
(!x U a) && (!x U b) && (!x U c) && (!y1 U a) && (!y1 U b) &&
(!y1 U c) && (!y2 U a) && (!y2 U b) && (!y2 U c) &&

~ T6-AfterF'
G(x -> X(!x U a)) && G(x -> X(!x U b)) && G(x -> X(!x U c)) &&
G(y1 -> X(!y1 U a)) && G(y1 -> X(!y1 U b)) && G(y1 -> X(!y1 U c)) &&
G(y2 -> X(!y2 U a)) && G(y2 -> X(!y2 U b)) && G(y2 -> X(!y2 U c))
```

Figure B.24 The LTL encoding of $\big((\{a,b\},\wedge) < c \Longrightarrow (\{x, y^{[1,2]}\},\wedge)\ \mid t\big)$ in SPOT syntax.

B.2.7 Encoding of $\Big(\big(\{a,b\},\vee\big) \Longrightarrow c < \big(\{x, y^{[1,2]}\},\wedge\big) \ \mid t\Big)$ into LTL

Consider the timed implication constraint $\Big(\big(\{a,b\},\vee\big) \Longrightarrow c < \big(\{x, y^{[1,2]}\},\wedge\big) \ \mid t\Big)$. After the removal of the range $y^{[1,2]}$, the new vocabulary is the set $\{a,b,c,x,y1,y2\}$. The LTL encoding of the property is the conjunction of the following formulas:

$$\begin{aligned}&\Box\big(\neg(a\wedge b)\big)\wedge\Box\big(\neg(a\wedge c)\big)\wedge\Box\big(\neg(a\wedge x)\big)\wedge\Box\big(\neg(a\wedge y1)\big)\\&\wedge\Box\big(\neg(a\wedge y2)\big)\wedge\Box\big(\neg(b\wedge c)\big)\wedge\Box\big(\neg(b\wedge x)\big)\wedge\Box\big(\neg(b\wedge y1)\big)\\&\wedge\Box\big(\neg(b\wedge y2)\big)\wedge\Box\big(\neg(c\wedge x)\big)\wedge\Box\big(\neg(c\wedge y1)\big)\wedge\Box\big(\neg(c\wedge y2)\big)\\&\wedge\Box\big(\neg(x\wedge y1)\big)\wedge\Box\big(\neg(x\wedge y2)\big)\wedge\Box\big(\neg(y1\wedge y2)\big)\end{aligned} \tag{$\mathcal{T}7$-Exclusiveness}$$

$$\begin{aligned}&\Box\Big(a\rightarrow\bigcirc(\neg a\,\mathcal{U}\,x)\Big)\wedge\Box\Big(a\rightarrow\bigcirc\big((\neg a\,\mathcal{U}\,y1)\vee(\neg a\,\mathcal{U}\,y2)\big)\Big)\\&\wedge\Box\Big(b\rightarrow\bigcirc(\neg b\,\mathcal{U}\,x)\Big)\wedge\Box\Big(b\rightarrow\bigcirc\big((\neg b\,\mathcal{U}\,y1)\vee(\neg b\,\mathcal{U}\,y2)\big)\Big)\\&\wedge\Box\Big(c\rightarrow\bigcirc(\neg c\,\mathcal{U}\,x)\Big)\wedge\Box\Big(c\rightarrow\bigcirc\big((\neg c\,\mathcal{U}\,y1)\vee(\neg c\,\mathcal{U}\,y2)\big)\Big)\end{aligned} \tag{$\mathcal{T}7$-MaxOne}$$

$$\begin{aligned}&\Box\Big(c\rightarrow(\neg a\,\mathcal{U}\,x)\Big)\wedge\Box\Big(c\rightarrow\big((\neg a\,\mathcal{U}\,y1)\vee(\neg a\,\mathcal{U}\,y2)\big)\Big)\\&\wedge\Box\Big(c\rightarrow(\neg b\,\mathcal{U}\,x)\Big)\wedge\Box\Big(c\rightarrow\big((\neg b\,\mathcal{U}\,y1)\vee(\neg b\,\mathcal{U}\,y2)\big)\Big)\end{aligned} \tag{$\mathcal{T}7$-Order}$$

$$\begin{aligned}&\Box\Big(y1\rightarrow\big((\neg y2\,\mathcal{U}\,a)\vee(\neg y2\,\mathcal{U}\,b)\big)\Big)\wedge\Box\Big(y1\rightarrow(\neg y2\,\mathcal{U}\,c)\Big)\\&\wedge\Box\Big(y2\rightarrow\big((\neg y1\,\mathcal{U}\,a)\vee(\neg y1\,\mathcal{U}\,b)\big)\Big)\wedge\Box\Big(y2\rightarrow(\neg y1\,\mathcal{U}\,c)\Big)\end{aligned} \tag{$\mathcal{T}7$-Range$\mathcal{F}'$}$$

$$\begin{aligned}&\Box\Big(x\rightarrow\big((\neg c\,\mathcal{U}\,a)\vee(\neg c\,\mathcal{U}\,b)\big)\Big)\wedge\Box\Big(y1\rightarrow\big((\neg c\,\mathcal{U}\,a)\vee(\neg c\,\mathcal{U}\,b)\big)\Big)\\&\wedge\Box\Big(y2\rightarrow\big((\neg c\,\mathcal{U}\,a)\vee(\neg c\,\mathcal{U}\,b)\big)\Big)\end{aligned} \tag{$\mathcal{T}7$-Order$\mathcal{F}'$}$$

$$\begin{aligned}&\big((\neg x\,\mathcal{U}\,a)\vee(\neg x\,\mathcal{U}\,b)\big)\wedge(\neg x\,\mathcal{U}\,c)\wedge\big((\neg y1\,\mathcal{U}\,a)\vee(\neg y1\,\mathcal{U}\,b)\big)\\&\wedge(\neg y1\,\mathcal{U}\,c)\wedge\big((\neg y2\,\mathcal{U}\,a)\vee(\neg y2\,\mathcal{U}\,b)\big)\wedge(\neg y2\,\mathcal{U}\,c)\end{aligned} \tag{$\mathcal{T}7$-First$\mathcal{F}'$}$$

$$\begin{aligned}&\Box\Big(x\rightarrow\bigcirc\big((\neg x\,\mathcal{U}\,a)\vee(\neg x\,\mathcal{U}\,b)\big)\Big)\wedge\Box\Big(x\rightarrow\bigcirc(\neg x\,\mathcal{U}\,c)\Big)\\&\wedge\Box\Big(y1\rightarrow\bigcirc\big((\neg y1\,\mathcal{U}\,a)\vee(\neg y1\,\mathcal{U}\,b)\big)\Big)\wedge\Box\Big(y1\rightarrow\bigcirc(\neg y1\,\mathcal{U}\,c)\Big)\\&\wedge\Box\Big(y2\rightarrow\bigcirc\big((\neg y2\,\mathcal{U}\,a)\vee(\neg y2\,\mathcal{U}\,b)\big)\Big)\wedge\Box\Big(y2\rightarrow\bigcirc(\neg y2\,\mathcal{U}\,c)\Big)\end{aligned} \tag{$\mathcal{T}7$-After$\mathcal{F}'$}$$

The encoding of $\Big(\big(\{a,b\},\vee\big) \Longrightarrow c < \big(\{x, y^{[1,2]}\},\wedge\big) \ \mid t\Big)$ is defined by Conjunction B.19. Figure B.26 provides the LTL encoding of the property in SPOT syntax. The respective SPOT monitor is shown in Figure B.25.

$$\begin{aligned}&\mathcal{T}7\text{-Exclusiveness}\ \wedge\ \mathcal{T}7\text{-MaxOne}\ \wedge\ \mathcal{T}7\text{-Order}\ \wedge\ \mathcal{T}7\text{-Range}\mathcal{F}'\\&\wedge\ \mathcal{T}7\text{-Order}\mathcal{F}'\ \wedge\ \mathcal{T}7\text{-First}\mathcal{F}'\ \wedge\ \mathcal{T}7\text{-After}\mathcal{F}'\end{aligned} \tag{B.19}$$

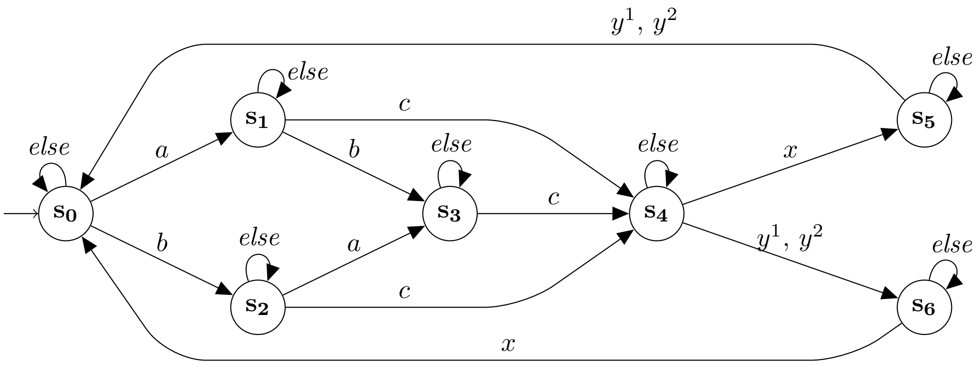


Figure B.25 The SPOT monitor for the LTL encoding of $\big((\{a,b\},\vee) \implies c < (\{x, y^{[1,2]}\},\wedge) \;\mid t\big)$. *else* stands for any name which is not of $\{a,b,c,x,y1,y2\}$. "$y1$, $y2$" means that $y1$ and $y2$ cannot occur simultaneously. Transitions which are not defined are forbidden.

```
~ T7-Exclusiveness
G(!(a && b)) && G(!(a && c)) && G(!(a && x)) && G(!(a && y1)) &&
G(!(a && y2)) && G(!(b && c)) && G(!(b && x)) && G(!(b && y1)) &&
G(!(b && y2)) && G(!(c && x)) && G(!(c && y1)) && G(!(c && y2)) &&
G(!(x && y1)) && G(!(x && y2)) && G(!(y1 && y2)) &&

~ T7-MaxOne
G(a-> X(!a U x)) && G(a-> X((!a U y1) || (!a U y2))) &&
G(b-> X(!b U x)) && G(b-> X((!b U y1) || (!b U y2))) &&
G(c-> X(!c U x)) && G(c-> X((!c U y1) || (!c U y2))) &&

~ T7-Order
G(c-> (!a U x)) && G(c-> ((!a U y1) || (!a U y2))) &&
G(c-> (!b U x)) && G(c-> ((!b U y1) || (!b U y2))) &&

~ T7-RangeF'
G(y1 -> ((!y2 U a) || (!y2 U b))) && G(y1 -> (!y2 U c)) &&
G(y2 -> ((!y1 U a) || (!y1 U b))) && G(y2 -> (!y1 U c)) &&

~ T7-OrderF'
G(x-> ((!c U a) || (!c U b))) &&
G(y1 -> ((!c U a) || (!c U b))) &&
G(y2 -> ((!c U a) || (!c U b))) &&

~ T7-FirstF'
((!x U a) || (!x U b)) && (!x U c) &&
((!y1 U a) || (!y1 U b)) && (!y1 U c) &&
((!y2 U a) || (!y2 U b)) && (!y2 U c) &&

~ T7-AfterF'
G(x -> X((!x U a) || (!x U b))) &&  G(x -> X(!x U c)) &&
G(y1 -> X((!y1 U a) || (!y1 U b))) && G(y1 -> X(!y1 U c)) &&
G(y2 -> X((!y2 U a) || (!y2 U b))) && G(y2 -> X(!y2 U c))
```

Figure B.26 The LTL encoding of $\big((\{a,b\},\vee) \implies c < (\{x, y^{[1,2]}\},\wedge) \;\mid t\big)$ in SPOT syntax.

B.2.8 Encoding of $\Big(\big(\{a,b\},\vee\big) < \big(\{c,d\},\wedge\big) \Longrightarrow \big(\{x,y^{[1,2]}\},\wedge\big) \ \mid t\Big)$ into LTL

Consider the timed implication constraint $\Big(\big(\{a,b\},\vee\big) < \big(\{c,d\},\wedge\big) \Longrightarrow \big(\{x,y^{[1,2]}\},\wedge\big) \ \mid t\Big)$. After the removal of the range $y^{[1,2]}$, the new vocabulary is the set $\{a,b,c,d,x,y1,y2\}$. The LTL encoding of the property consists of the following components:

$$\begin{aligned}
&\Box\big(\neg(a\wedge b)\big)\wedge\Box\big(\neg(a\wedge c)\big)\wedge\Box\big(\neg(a\wedge d)\big)\wedge\Box\big(\neg(a\wedge x)\big) && (\mathcal{T}8\text{-Exclusiveness})\\
&\wedge\Box\big(\neg(a\wedge y1)\big)\wedge\Box\big(\neg(a\wedge y2)\big)\wedge\Box\big(\neg(b\wedge c)\big)\wedge\Box\big(\neg(b\wedge d)\big)\\
&\wedge\Box\big(\neg(b\wedge x)\big)\wedge\Box\big(\neg(b\wedge y1)\big)\wedge\Box\big(\neg(b\wedge y2)\big)\wedge\Box\big(\neg(c\wedge d)\big)\\
&\wedge\Box\big(\neg(c\wedge x)\big)\wedge\Box\big(\neg(c\wedge y1)\big)\wedge\Box\big(\neg(c\wedge y2)\big)\wedge\Box\big(\neg(d\wedge x)\big)\\
&\wedge\Box\big(\neg(d\wedge y1)\big)\wedge\Box\big(\neg(d\wedge y2)\big)\wedge\Box\big(\neg(x\wedge y1)\big)\wedge\Box\big(\neg(x\wedge y2)\big)\wedge\Box\big(\neg(y1\wedge y2)\big)
\end{aligned}$$

$$\begin{aligned}
&\Box\Big(a\rightarrow\bigcirc(\neg a\,\mathcal{U}\,x)\Big)\wedge\Box\Big(a\rightarrow\bigcirc\big((\neg a\,\mathcal{U}\,y1)\vee(\neg a\,\mathcal{U}\,y2)\big)\Big) && (\mathcal{T}8\text{-MaxOne})\\
&\wedge\Box\Big(b\rightarrow\bigcirc(\neg b\,\mathcal{U}\,x)\Big)\wedge\Box\Big(b\rightarrow\bigcirc\big((\neg b\,\mathcal{U}\,y1)\vee(\neg b\,\mathcal{U}\,y2)\big)\Big)\\
&\wedge\Box\Big(c\rightarrow\bigcirc(\neg c\,\mathcal{U}\,x)\Big)\wedge\Box\Big(c\rightarrow\bigcirc\big((\neg c\,\mathcal{U}\,y1)\vee(\neg c\,\mathcal{U}\,y2)\big)\Big)\\
&\wedge\Box\Big(d\rightarrow\bigcirc(\neg d\,\mathcal{U}\,x)\Big)\wedge\Box\Big(d\rightarrow\bigcirc\big((\neg d\,\mathcal{U}\,y1)\vee(\neg d\,\mathcal{U}\,y2)\big)\Big)
\end{aligned}$$

$$\begin{aligned}
&\Box\Big(c\rightarrow(\neg a\,\mathcal{U}\,x)\Big)\wedge\Box\Big(c\rightarrow\big((\neg a\,\mathcal{U}\,y1)\vee(\neg a\,\mathcal{U}\,y2)\big)\Big) && (\mathcal{T}8\text{-Order})\\
&\wedge\Box\Big(c\rightarrow(\neg b\,\mathcal{U}\,x)\Big)\wedge\Box\Big(c\rightarrow\big((\neg b\,\mathcal{U}\,y1)\vee(\neg b\,\mathcal{U}\,y2)\big)\Big)\\
&\wedge\Box\Big(d\rightarrow(\neg a\,\mathcal{U}\,x)\Big)\wedge\Box\Big(d\rightarrow\big((\neg a\,\mathcal{U}\,y1)\vee(\neg a\,\mathcal{U}\,y2)\big)\Big)\\
&\wedge\Box\Big(d\rightarrow(\neg b\,\mathcal{U}\,x)\Big)\wedge\Box\Big(d\rightarrow\big((\neg b\,\mathcal{U}\,y1)\vee(\neg b\,\mathcal{U}\,y2)\big)\Big)
\end{aligned}$$

$$\begin{aligned}
&\Box\Big(y1\rightarrow\big((\neg y2\,\mathcal{U}\,a)\vee(\neg y2\,\mathcal{U}\,b)\big)\Big)\wedge\Box\Big(y1\rightarrow(\neg y2\,\mathcal{U}\,c)\Big) && (\mathcal{T}8\text{-Range}\mathcal{F}')\\
&\wedge\Box\Big(y1\rightarrow(\neg y2\,\mathcal{U}\,d)\Big)\wedge\Box\Big(y2\rightarrow\big((\neg y1\,\mathcal{U}\,a)\vee(\neg y1\,\mathcal{U}\,b)\big)\Big)\\
&\wedge\Box\Big(y2\rightarrow(\neg y1\,\mathcal{U}\,c)\Big)\wedge\Box\Big(y2\rightarrow(\neg y1\,\mathcal{U}\,d)\Big)
\end{aligned}$$

$$\begin{aligned}
&\Box\Big(x\rightarrow\big((\neg c\,\mathcal{U}\,a)\vee(\neg c\,\mathcal{U}\,b)\big)\Big)\wedge\Box\Big(x\rightarrow\big((\neg d\,\mathcal{U}\,a)\vee(\neg d\,\mathcal{U}\,b)\big)\Big) && (\mathcal{T}8\text{-Order}\mathcal{F}')\\
&\wedge\Box\Big(y1\rightarrow\big((\neg c\,\mathcal{U}\,a)\vee(\neg c\,\mathcal{U}\,b)\big)\Big)\wedge\Box\Big(y1\rightarrow\big((\neg d\,\mathcal{U}\,a)\vee(\neg d\,\mathcal{U}\,b)\big)\Big)\\
&\wedge\Box\Big(y2\rightarrow\big((\neg c\,\mathcal{U}\,a)\vee(\neg c\,\mathcal{U}\,b)\big)\Big)\wedge\Box\Big(y2\rightarrow\big((\neg d\,\mathcal{U}\,a)\vee(\neg d\,\mathcal{U}\,b)\big)\Big)
\end{aligned}$$

$$\begin{aligned}
&\big((\neg x\,\mathcal{U}\,a)\vee(\neg x\,\mathcal{U}\,b)\big)\wedge(\neg x\,\mathcal{U}\,c)\wedge(\neg x\,\mathcal{U}\,d)\wedge\big((\neg y1\,\mathcal{U}\,a)\vee(\neg y1\,\mathcal{U}\,b)\big) && (\mathcal{T}8\text{-First}\mathcal{F}')\\
&\wedge(\neg y1\,\mathcal{U}\,c)\wedge(\neg y1\,\mathcal{U}\,d)\wedge\big((\neg y2\,\mathcal{U}\,a)\vee(\neg y2\,\mathcal{U}\,b)\big)\wedge(\neg y2\,\mathcal{U}\,c)\wedge(\neg y2\,\mathcal{U}\,d)
\end{aligned}$$

$$\begin{aligned}
&\Box\Big(x\rightarrow\bigcirc\big((\neg x\,\mathcal{U}\,a)\vee(\neg x\,\mathcal{U}\,b)\big)\Big)\wedge\Box\Big(x\rightarrow\bigcirc(\neg x\,\mathcal{U}\,c)\Big) && (\mathcal{T}8\text{-After}\mathcal{F}')\\
&\wedge\Box\Big(x\rightarrow\bigcirc(\neg x\,\mathcal{U}\,d)\Big)\wedge\Box\Big(y1\rightarrow\bigcirc\big((\neg y1\,\mathcal{U}\,a)\vee(\neg y1\,\mathcal{U}\,b)\big)\Big)\\
&\wedge\Box\Big(y1\rightarrow\bigcirc(\neg y1\,\mathcal{U}\,c)\Big)\wedge\Box\Big(y1\rightarrow\bigcirc(\neg y1\,\mathcal{U}\,d)\Big)\\
&\wedge\Box\Big(y2\rightarrow\bigcirc\big((\neg y2\,\mathcal{U}\,a)\vee(\neg y2\,\mathcal{U}\,b)\big)\Big)\wedge\Box\Big(y2\rightarrow\bigcirc(\neg y2\,\mathcal{U}\,c)\Big)\wedge\Box\Big(y2\rightarrow\bigcirc(\neg y2\,\mathcal{U}\,d)\Big)
\end{aligned}$$

The LTL encoding of $\Big(\big(\{a,b\},\vee\big) < \big(\{c,d\},\wedge\big) \Longrightarrow \big(\{x,y^{[1,2]}\},\wedge\big) \ \mid t\Big)$ is defined by Conjunction B.20. Figures B.28 and B.27 show respectively the LTL encoding of the property in SPOT syntax and the corresponding SPOT monitor.

$$\begin{aligned}&\mathcal{T}8\text{-Exclusiveness} \ \wedge \ \mathcal{T}8\text{-MaxOne} \ \wedge \ \mathcal{T}8\text{-Order} \ \wedge \ \mathcal{T}8\text{-Range}\mathcal{F}' \\ &\wedge \ \mathcal{T}8\text{-Order}\mathcal{F}' \ \wedge \ \mathcal{T}8\text{-First}\mathcal{F}' \ \wedge \ \mathcal{T}8\text{-After}\mathcal{F}'\end{aligned} \tag{B.20}$$

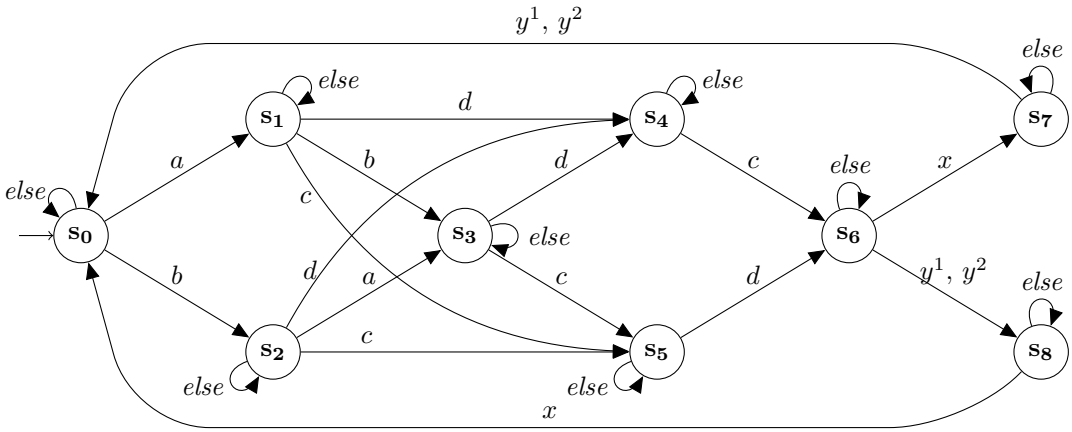


Figure B.27 The SPOT monitor for the LTL encoding of $\big((\{a,b\},\vee) < (\{c,d\},\wedge) \Longrightarrow (\{x,y^{[1,2]}\},\wedge) \ \mid t\big)$. *else* stands for any name which is not of $\{a,b,c,d,x,y1,y2\}$. "$y1$, $y2$" means that $y1$ and $y2$ cannot occur simultaneously. Transitions which are not defined are forbidden.

```
~ T8-Exclusiveness
G(!(a && b)) && G(!(a && c)) && G(!(a && d)) && G(!(a && x)) &&
G(!(a && y1)) && G(!(a && y2)) && G(!(b && c)) && G(!(b && d)) &&
G(!(b && x)) && G(!(b && y1)) && G(!(b && y2)) && G(!(c && d)) &&
G(!(c && x)) && G(!(c && y1)) && G(!(c && y2)) && G(!(d && x)) &&
G(!(d && y1)) && G(!(d && y2)) && G(!(x && y1)) && G(!(x && y2)) &&
G(!(y1 && y2)) &&

~ T8-MaxOne
G(a-> X(!a U x)) && G(a-> X((!a U y1) || (!a U y2))) &&
G(b-> X(!b U x)) && G(b-> X((!b U y1) || (!b U y2))) &&
G(c-> X(!c U x)) && G(c-> X((!c U y1) || (!c U y2))) &&
G(d-> X(!d U x)) && G(d-> X((!d U y1) || (!d U y2))) &&

~ T8-Order
G(c-> (!a U x)) && G(c-> ((!a U y1) || (!a U y2))) &&
G(c-> (!b U x)) && G(c-> ((!b U y1) || (!b U y2))) &&
G(d-> (!a U x)) && G(d-> ((!a U y1) || (!a U y2))) &&
G(d-> (!b U x)) && G(d-> ((!b U y1) || (!b U y2))) &&

~ T8-RangeF'
G(y1 -> ((!y2 U a)|| (!y2 U b))) && G(y1 -> (!y2 U c)) &&
G(y1-> (!y2 U d)) &&
G(y2 -> ((!y1 U a)|| (!y1 U b))) && G(y2 -> (!y1 U c)) &&
G(y2->  (!y1 U d)) &&

~ T8-OrderF'
G(x-> ((!c U a) || (!c U b))) && G(x-> ((!d U a) || (!d U b))) &&
G(y1 -> ((!c U a) || (!c U b))) && G(y1 -> ((!d U a) || (!d U b))) &&
G(y2 -> ((!c U a) || (!c U b))) && G(y2 -> ((!d U a) || (!d U b))) &&

~ T8-FirstF'
((!x U a) || (!x U b)) && (!x U c) && (!x U d) &&
((!y1 U a) || (!y1 U b)) && (!y1 U c) && (!y1 U d) &&
((!y2 U a) || (!y2 U b)) && (!y2 U c) && (!y2 U d) &&

~ T8-AfterF'
G(x -> X((!x U a) || (!x U b))) &&  G(x -> X(!x U c)) &&
G(x -> X(!x U d)) &&
G(y1 -> X((!y1 U a) || (!y1 U b))) && G(y1 -> X(!y1 U c)) &&
G(y1 -> X(!y1 U d)) &&
G(y2 -> X((!y2 U a) || (!y2 U b))) && G(y2 -> X(!y2 U c)) &&
G(y2 -> X(!y2 U d))
```

Figure B.28 The LTL encoding of $\big((\{a,b\},\vee) < (\{c,d\},\wedge) \Longrightarrow (\{x,y^{[1,2]}\},\wedge)\ \mid t\big)$ in SPOT syntax.

Bibliography

[18505] IEEE Std 1850-2005. *IEEE Standard for Property Specification Language (PSL)*. 2005 (pp. 3, 14, 30, 32–34, 65, 66, 138).

[Aba+00] Yael Abarbanel et al. "FoCs – Automatic Generation of Simulation Checkers from Formal Specifications". In: *Computer Aided Verification: 12th International Conference, CAV 2000, Chicago, IL, USA, July 15-19, 2000. Proceedings*. Ed. by E. Allen Emerson and Aravinda Prasad Sistla. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, pp. 538–542. ISBN: 978-3-540-45047-4 (pp. 107, 108, 170).

[AO14a] Inc. Accellera Organization. *Accelera Standard. Open Verification Library (OVL) v.2.8.1*. http://www.accellera.org. 2014 (pp. 13, 173).

[AO14b] Inc. Accellera Organization. *Accelera Standard. Universal Verificatoin Methodology (UVM) v.1.2. Class Reference*. http://www.accellera.org. 2014 (pp. 13, 134).

[Acc] *Accellera Systems Initiative*. www.accellera.org. 2016 (p. 13).

[Ake78] S. B. Akers. "Binary Decision Diagrams". In: *IEEE Trans. Comput.* 27.6 (June 1978), pp. 509–516. ISSN: 0018-9340 (p. 14).

[AH01] Luca de Alfaro and Thomas A. Henzinger. "Interface Automata". In: *Proceedings of the Ninth Annual Symposium on Foundations of Software Engineering (FSE), ACM*. Press, 2001, pp. 109–120 (pp. 60, 174).

[AS87] Bowen Alpern and Fred B Schneider. "Recognizing Safety and Liveness". In: *Distributed computing* 2.3 (1987), pp. 117–126 (pp. 135, 138).

[Alt+03] Karine Altisen et al. "Using Controller Synthesis to Build Property-Enforcing Layers". In: *European Symposium on Programming (ESOP)*. 2003 (p. 25).

[And03] David J. Anderson. *Agile Management for Software Engineering: Applying the Theory of Constraints for Business Results*. Prentice Hall, 2003. ISBN: 0131424602 (p. 133).

[And96a] C. André. "Representation and Analysis of Reactive Behaviors: A Synchronous Approach". In: *Computational Engineering in Systems Applications (CESA)*. Lille (F): IEEE-SMC, 1996, pp. 19–29 (p. 27).

[And96b] Charles André. *SyncCharts: a Visual Representation of Reactive Behaviors*. Tech. rep. RR 95–52, rev. RR (96–56). Sophia-Antipolis, France: I3S, 1996 (p. 27).

[Arg] *Argosim*. URL: http://argosim.com (p. 134).

[AMZ04] Sigal Asaf, Eitan Marcus, and Avi Ziv. "Defining Coverage Views to Improve Functional Coverage Analysis". In: *Proceedings of the 41st Annual Design Automation Conference*. DAC '04. San Diego, CA, USA: ACM, 2004, pp. 41–44. ISBN: 1-58113-828-8 (p. 13).

[BP06] F. Balarin and R. Passerone. "Functional Verification Methodology Based on Formal Interface Specification and Transactor Generation". In: *Proceedings of the Design Automation Test in Europe Conference*. Vol. 1. 2006, pp. 1–6 (p. 173).

[BP07] F. Balarin and R. Passerone. "Specification, Synthesis, and Simulation of Transactor Processes". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 26.10 (2007), pp. 1749–1762. ISSN: 0278-0070 (p. 173).

[Bar+09] Clark W Barrett et al. "Satisfiability Modulo Theories". In: *Handbook of satisfiability* 185 (2009), pp. 825–885 (p. 14).

[Bar+01] Detlef Bartetzko et al. *Jass – Java with Assertions*. 2001 (pp. 36, 134).

[Bau+12] Sebastian S. Bauer et al. "Moving from Specifications to Contracts in Component-Based Design". English. In: *Fundamental Approaches to Software Engineering*. Ed. by Juan de Lara and Andrea Zisman. Vol. 7212. LNCS. Springer Berlin Heidelberg, 2012, pp. 43–58. ISBN: 978-3-642-28871-5 (pp. 135, 174).

[Bee+01] Ilan Beer et al. "The Temporal Logic Sugar". In: *Computer Aided Verification: 13th International Conference, CAV 2001 Paris, France, July 18–22, 2001 Proceedings*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 363–367. ISBN: 978-3-540-44585-2 (p. 32).

[BA14] Zeineb Belhadj Amor. "Validation of complex systems on a chip, from TLM level to RTL". Theses. Université Grenoble Alpes, Dec. 2014. URL: https://tel.archives-ouvertes.fr/tel-01228059 (pp. 75, 126, 171).

[BG90] A. Benveniste and P. Le Guernic. "Hybrid dynamical systems theory and the Signal language". In: *IEEE Transactions on Automatic Control* 35.5 (1990), pp. 535–546. ISSN: 0018-9286 (p. 27).

[Ben+08] Albert Benveniste et al. "Multiple Viewpoint Contract-Based Specification and Design". English. In: *Formal Methods for Components and Objects*. Ed. by FrankS. de Boer et al. Vol. 5382. Lecture Notes in Computer Science. 2008, pp. 200–225. ISBN: 978-3-540-92187-5 (pp. 135, 174).

[Ber+06] Janick Bergeron et al. *Verification Methodology Manual for SystemVerilog*. Springer, 2006. ISBN: 978-0-387-25538-5 (p. 172).

[Ber92] Gerard Berry. "A Hardware Implementation of Pure Esterel". In: *Academy Proceedings in Engineering Sciences, Indian Academy of Sciences* 17 (1992), pp. 95–130 (p. 27).

[BG92] Gérard Berry and Georges Gonthier. "The ESTEREL Synchronous Programming Language: Design, Semantics, Implementation". In: *Sci. Comput. Program.* 19.2 (Nov. 1992), pp. 87–152. ISSN: 0167-6423 (p. 27).

[Ber02] Gérard Berry. "Real Time Programming: Special Purpose or General Purpose Languages." In: *IFIP Congress*. Jan. 3, 2002, pp. 11–17. URL: http://dblp.uni-trier.de/db/conf/ifip/ifip89.html#Berry89 (p. 24).

[Ber+09] Nathalie Bertrand et al. "A Compositional Approach on Modal Specifications for Timed Systems". In: *Formal Methods and Software Engineering: 11th International Conference on Formal Engineering Methods ICFEM 2009, Rio de Janeiro, Brazil, December 9-12, 2009. Proceedings*. Ed. by Karin Breitman and Ana Cavalcanti. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 679–697. ISBN: 978-3-642-10373-5 (p. 174).

[Bie+08] Dariusz Biernacki et al. "Clock-directed modular code generation for synchronous data-flow languages". In: *Proceedings of the 2008 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'08), Tucson, AZ, USA, June 12-13, 2008*. 2008, pp. 121–130 (p. 27).

[BFF05] N. Bombieri, A. Fedeli, and F. Fummi. "On PSL Properties Re-use in SoC Design Flow Based on Transaction Level Modeling". In: *2005 Sixth International Workshop on Microprocessor Test and Verification*. 2005, pp. 127–132 (pp. 108, 173).

[BF06] N. Bombieri and F. Fummi. "On the Automatic Transactor Generation for TLM-based Design Flows". In: *2006 IEEE International High Level Design Validation and Test Workshop*. 2006, pp. 85–92 (p. 173).

[Bom+07] N. Bombieri et al. "Hybrid, Incremental Assertion-Based Verification for TLM Design Flows". In: *IEEE Design Test of Computers* 24.2 (2007), pp. 140–152. ISSN: 0740-7475 (pp. 14, 174).

[BFD08] Nicola Bombieri, Franco Fummi, and Nicola Deganello. "Integrating RTL IPs into TLM Designs Through Automatic Transactor Generation". In: *Design, Automation Test in Europe Conference Exhibition* 00.undefined (2008), pp. 15–20 (p. 173).

[BFP07] Nicola Bombieri, Franco Fummi, and Graziano Pravadelli. "Incremental ABV for functional validation of TL-to-RTL design refinement." In: *DATE*. Ed. by Rudy Lauwereins and Jan Madsen. EDA Consortium, San Jose, CA, USA, 2007, pp. 882–887. ISBN: 978-3-9810801-2-4. URL: http://dblp.uni-trier.de/db/conf/date/date2007.html#BombieriFP07 (pp. 108, 173).

[BFP06] Nicola Bombieri, Franco Fummi, and Graziano Pravadelli. "On the Evaluation of Transactor-based Verification for Reusing TLM Assertions and Testbenches at RTL". In: *Proceedings of the Conference on Design, Automation and Test in Europe: Proceedings*. DATE '06. Munich, Germany: European Design and Automation Association, 2006, pp. 1007–1012. ISBN: 3-9810801-0-6. URL: http://dl.acm.org/citation.cfm?id=1131481.1131761 (p. 173).

[Bor+06] Dominique Borrione et al. "PSL-Based Online Monitoring of Digital Systems". In: *Applications of Specification and Design Languages for SoCs: Selected papers from FDL 2005*. Ed. by A. Vachoux. Dordrecht: Springer Netherlands, 2006, pp. 5–22. ISBN: 978-1-4020-4998-9 (p. 90).

[Bou+92] Ahmed Bouajjani et al. "Minimal State Graph Generation". In: *Sci. Comput. Program.* 18.3 (1992), pp. 247–269 (p. 27).

[Bra+00] Dhananjay S. Brahme et al. *The Transaction-Based Verification Methodology*. Tech. rep. Cadence Berkeley Labs, 2000 (p. 173).

[BB09] Robert Brummayer and Armin Biere. "Boolector: An Efficient SMT Solver for Bit-Vectors and Arrays". In: *Proceedings of the 15th International Conference on Tools and Algorithms for the Construction and Analysis of Systems: Held As Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009*, TACAS '09. York, UK: Springer-Verlag, 2009, pp. 174–177. ISBN: 978-3-642-00767-5 (p. 14).

[CDSa] Inc. Cadence Design Systems. *Open Verification Methodology Multi-Language (OVM-ML)* (p. 173).

[CDSb] Inc. Cadence Design Systems. *Universal Reuse Methodology* (p. 172).

[CDSc] Inc. Cadence Design Systems. *Universal Verification Methodology Multi-Language (OVM-ML)* (p. 173).

[CG03] Lukai Cai and Daniel Gajski. "Transaction Level Modeling: An Overview". In: *Proceedings of the 1st IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*. CODES+ISSS '03. Newport Beach, CA, USA: ACM, 2003, pp. 19–24. ISBN: 1-58113-742-7 (p. 21).

[Cas+87] P. Caspi et al. "Lustre, a Declarative Language for Programming Synchronous Systems". In: *14th Symposium on Principles of Programming Languages*. Munich, Jan. 1987 (p. 103).

[Cer+14] Eduard Cerny et al. *SVA: The Power of Assertions in SystemVerilog*. 2nd. Springer Publishing Company, Incorporated, 2014. ISBN: 3319071386, 9783319071381 (p. 14).

[CVK04] Ben Cohen, Srinivasan Venkataramanan, and Ajeetha Kumari. *Using PSL/Sugar for Formal and Dynamic Verification: Guide to Property Specification Language for Assertion-Based Verification*. Cohen Publishing, 2004 (pp. 66, 134, 167).

[Cop+04] M. Coppola et al. "Spidergon: a novel on-chip communication network". In: *System-on-Chip, 2004. Proceedings. 2004 International Symposium on*. 2004, pp. 15– (p. 11).

[Cor08] Jérôme Cornet. "Separation of Functional and Non-Functional Aspects in Transactional Level Models of Systems-on-Chip". PhD thesis. Institut National Polytechnique de Grenoble, 2008 (pp. 4, 12, 42, 46).

[Cou99] Jean-Michel Couvreur. "On-the-Fly Verification of Linear Temporal Logic". In: *FM'99 - Formal Methods, World Congress on Formal Methods in the Development of Computing Systems, Toulouse, France, September 20-24, 1999, Proceedings, Volume I*. 1999, pp. 253–271 (p. 79).

[Dah+05] Anat Dahan et al. "Combining System Level Modeling with Assertion Based Verification". In: *6th International Symposium on Quality of Electronic Design (ISQED 2005), 21-23 March 2005, San Jose, CA, USA*. 2005, pp. 310–315 (p. 170).

[DCBK10] Bill Bunton David C. Black Jack Donovan and Anna Keist. *SystemC: From the Ground Up. Second Edition*. Springer, 2010. ISBN: 978-1489982667 (pp. 15, 18–20, 22).

[DGV13] Giuseppe De Giacomo and Moshe Y. Vardi. "Linear Temporal Logic and Linear Dynamic Logic on Finite Traces". In: *Proceedings of the Twenty-Third International Joint Conference on Artificial Intelligence*. IJCAI '13. Beijing, China: AAAI Press, 2013, pp. 854–860. ISBN: 978-1-57735-633-2. URL: http://dl.acm.org/citation.cfm?id=2540128.2540252 (p. 34).

[DMB08] Leonardo De Moura and Nikolaj Bjørner. "Z3: An Efficient SMT Solver". In: *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. TACAS'08/ETAPS'08. Budapest, Hungary: Springer-Verlag, 2008, pp. 337–340. ISBN: 3-540-78799-2, 978-3-540-78799-0. URL: http://dl.acm.org/citation.cfm?id=1792734.1792766 (p. 14).

[Dev91] Srinivas Devadas. "Optimizing Interacting Finite State Machines Using Sequential Don't Cares". In: *IEEE Trans. on CAD of Integrated Circuits and Systems* 10.12 (1991), pp. 1473–1484 (pp. 36, 135, 174).

[DFM13] Alexandre Donzé, Thomas Ferrère, and Oded Maler. "Efficient Robust Monitoring for STL". In: *Computer Aided Verification: 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 264–279. ISBN: 978-3-642-39799-8 (p. 30).

[DL16] Alexandre Duret-Lutz. *Spot's Temporal Logic Formulas*. Tech. rep. Available online: https://spot.lrde.epita.fr/tl.pdf. 2016 (pp. 33, 79).

[DLP04] Alexandre Duret-Lutz and Denis Poitrenaud. "SPOT: an Extensible Model Checking Library using Transition-based Generalized Büchi Automata". In: *Proceedings of the 12th IEEE/ACM International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS'04)*. Volendam, The Netherlands: IEEE Computer Society, Oct. 2004, pp. 76–83 (p. 78).

[DL+16] Alexandre Duret-Lutz et al. "Spot 2.0 — a framework for LTL and ω-automata manipulation". In: *Proceedings of the 14th International Symposium on Automated Technology for Verification and Analysis (ATVA'16)*. Vol. 9938. Lecture Notes in Computer Science. To appear. Springer, Oct. 2016, pp. 122–129 (p. 78).

[Eck+06a] W. Ecker et al. "Specification Language for Transaction Level Assertions". In: *2006 IEEE International High Level Design Validation and Test Workshop*. 2006, pp. 77–84 (p. 168).

[Eck+07] Wolfgang Ecker et al. "Interactive presentation: Implementation of a transaction level assertion framework in SystemC". In: *2007 Design, Automation and Test in Europe Conference and Exposition, DATE 2007, Nice, France, April 16-20, 2007*. 2007, pp. 894–899 (p. 168).

[Eck+06b] Wolfgang Ecker et al. "Requirements and Concepts for Transaction Level Assertions". In: *24th International Conference on Computer Design (ICCD 2006), 1-4 October 2006, San Jose, CA, USA*. 2006, pp. 286–293 (pp. 3, 168).

[Eck+10] Wolfgang Ecker et al. "TLM+ modeling of embedded HW/SW systems". In: *Design, Automation and Test in Europe, DATE 2010, Dresden, Germany, March 8-12, 2010*. 2010, pp. 75–80 (p. 175).

[ES03] Niklas Eén and Niklas Sörensson. "An Extensible SAT-solver". In: *Theory and Applications of Satisfiability Testing, 6th International Conference, SAT 2003. Santa Margherita Ligure, Italy, May 5-8, 2003 Selected Revised Papers*. 2003, pp. 502–518 (p. 14).

[EF06] Cindy Eisner and Dana Fisman. *A Practical Introduction to PSL*. Series on Integrated Circuits and Systems. Springer, 2006. ISBN: 978-0-387-35313-5 (pp. 3, 30, 32, 33).

[EF09] Cindy Eisner and Dana Fisman. "Structural Contradictions". In: *Hardware and Software: Verification and Testing: 4th International Haifa Verification Conference, HVC 2008, Haifa, Israel, October 27-30, 2008. Proceedings*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 164–178. ISBN: 978-3-642-01702-5 (p. 32).

[Cad] JasperGold Formal Verification Platform, *Cadence Inc.* Online; accessed 08-November-2016. 2016. URL: https://www.cadence.com/content/cadence-www/global/en_US/home/tools/system-design-and-verification/formal-and-static-verification/jasper-gold-verification-platform.html (p. 107).

[Men] ModelSim, *Mentor Graphics Inc.* https://www.mentor.com/products/fv/modelsim/. Online; accessed 08-November-2016. 2016 (p. 107).

[Syna] VCS, *Synopsys Inc.* http://www.synopsys.com/Tools/Verification/FunctionalVerification/Pages/VCS.aspx. Online; accessed 08-November-2016. 2016 (p. 107).

[Eng15] Jakob Engblom. "Continuous Integration for Embedded Systems using Simulation". In: *Embedded World Exhibition and Conference*. 2015 (p. 174).

[EJ] Nicolas Halbwachs Erwan Jahier Pascal Raymond. *The Lustre V6 Reference Manual*. http://www-verimag.imag.fr/DIST-TOOLS/SYNCHRONE/lustre-v6/doc/lv6-ref-man.pdf. Accessed: 2016-08-22 (pp. 27, 103).

[EPH06] Brett D. Estrade, A. Louise Perkins, and John M. Harris. "Explicitly Parallel Regular Expressions". In: *Proceedings of the First International Multi-Symposiums on Computer and Computational Sciences - Volume 1 (IMSCCS'06) - Volume 01*. IMSCCS '06. Washington, DC, USA: IEEE Computer Society, 2006, pp. 402–409. ISBN: 0-7695-2581-4-01 (p. 78).

[FSO+12] Marcio F. S. Oliveira et al. "A SystemC Library for Advanced TLM Verification". In: *Proceeding of Design and Verification Conference (DVCON)*. 2012 (p. 173).

[FP09] Georgios E. Fainekos and George J. Pappas. "Robustness of Temporal Logic Specifications for Continuous-time Signals". In: *Theor. Comput. Sci.* 410.42 (Sept. 2009), pp. 4262–4291. ISSN: 0304-3975 (p. 30).

[Fel68] William Feller. "An Introduction to Probability Theory and Its Applications". In: Wiley, 3rd edition, 1968, pp. 28–31. ISBN: 0471257087 (p. 76).

[Fer11] Luca Ferro. "Verification of temporal properties for SystemC TLM specifications". Theses. Université Grenoble Alpes, July 2011. URL: https://tel.archives-ouvertes.fr/tel-00633069 (pp. 87, 108, 171).

[FP10] Luca Ferro and Laurence Pierre. "ISIS: Runtime Verification of TLM Platforms". In: *Advances in Design Methods from Modeling Languages for Embedded Systems and SoC's: Selected Contributions on Specification, Design, and Verification from FDL 2009*. Ed. by Dominique Borrione. Dordrecht: Springer Netherlands, 2010, pp. 213–226. ISBN: 978-90-481-9304-2 (pp. 75, 125, 126, 171).

[FG05] Cormac Flanagan and Patrice Godefroid. "Dynamic Partial-Order Reduction for Model Checking Software". In: *ACM Sigplan Notices*. Vol. 40. 1. ACM. 2005, pp. 110–121 (pp. 65, 136).

[Fos09] Harry Foster. "Applied Assertion-Based Verification: An Industry Perspective". In: *Found. Trends Electron. Des. Autom.* 3.1 (Jan. 2009), pp. 1–95. ISSN: 1551-3939 (p. 14).

[Fos06] Harry Foster. *Introduction to the New Accellera Open Verification Library*. 2006 (p. 173).

[Fur05] Steve Furber. "Future trends in SoC interconnect". In: *2005 IEEE VLSI-TSA International Symposium on VLSI Design, Automation and Test,(VLSI-TSA-DAT)* 2005 (2005), pp. 295–298 (p. 11).

[Fur00] Steve Furber. "The Advanced Microcontroller Bus Architecture (AMBA)". In: *ARM System-on-Chip Architecture (2nd Edition)*. Addison-Wesley Professional, 2000, pp. 216–220. ISBN: 978-0201675191 (p. 10).

[Gam86] B. Gamatie. "Towards specification and proof of asynchronous systems". In: *STACS 86: 3rd Annual Symposium on Theoretical Aspects of Computer Science Orsay, France, January 16–18, 1986*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1986, pp. 262–276. ISBN: 978-3-540-39758-8 (p. 28).

[Gam+95] Erich Gamma et al. *Design Patterns: Elements of Reusable Object-oriented Software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995. ISBN: 0-201-63361-2 (pp. 109, 168).

[Gla09] Mark Glasser. *Open Verification Methodology Cookbook*. Springer, 2009. ISBN: 978-1-4419-0967-1 (p. 173).

[GHR03] Laure Gonnord, Nicolas Halbwachs, and Pascal Raymond. "From Discrete Duration Calculus to Symbolic Automata". In: *Third International Workshop on Synchronous Languages, Applications, and Programs (SLAP 2004)*. Vol. 153. 4. Barcelona, Spain, Mar. 2003, pp. 3–18. URL: https://hal.archives-ouvertes.fr/hal-00198433 (pp. 90, 140).

[GD03] Daniel Große and Rolf Drechsler. "Formal verification of LTL formulas for SystemC designs". In: *Proceedings of the 2003 International Symposium on Circuits and Systems, ISCAS 2003, Bangkok, Thailand, May 25-28, 2003*. 2003, pp. 245–248 (p. 12).

[GED07] Daniel Große, Rüdiger Ebendt, and Rolf Drechsler. "Improvements for Constraint Solving in the SystemC Verification Library". In: *Proceedings of the 17th ACM Great Lakes Symposium on VLSI 2007, Stresa, Lago Maggiore, Italy, March 11-13, 2007*. 2007, pp. 493–496 (p. 14).

[Gro+08] Daniel Große et al. "Contradiction Analysis for Constraint-based Random Simulation". In: *Forum on Specification and Design Languages, FDL 2008, September 23-25, 2008, Stuttgart, Germany, Proceedings*. 2008, pp. 130–135 (p. 14).

[Gro+09] Daniel Große et al. "Debugging Contradictory Constraints in Constraint-Based Random Simulation". In: *Languages for Embedded Systems and their Applications: Selected Contributions on Specification, Design, and Verification from FDL'08*. Dordrecht: Springer Netherlands, 2009, pp. 273–290. ISBN: 978-1-4020-9714-0 (p. 14).

[Hae+12] Finn Haedicke et al. "CRAVE: An advanced constrained random verification environment for SystemC". In: *2012 International Symposium on System on Chip, ISSoC 2012, Tampere, Finland, October 10-12, 2012*. 2012, pp. 1–7 (pp. 14, 134).

[HLR93] N. Halbwachs, F. Lagnier, and P. Raymond. "Synchronous observers and the verification of reactive systems". In: *Third Int. Conf. on Algebraic Methodology and Software Technology, AMAST'93*. Ed. by M. Nivat et al. Twente: Workshops in Computing, Springer Verlag, 1993 (pp. 27, 28).

[Hal+91] N. Halbwachs et al. "The Synchronous Dataflow Programming Language LUSTRE". In: *Proceedings of the IEEE*. 1991, pp. 1305–1320 (pp. 27, 103).

[Hal93] Nicolas Halbwachs. *Synchronous programming of reactive systems*. The Kluwer international series in engineering and computer science. Dordrecht: Kluwer Academic publ, 1993. ISBN: 0-7923-9311-2. URL: http://opac.inria.fr/record=b1081507 (p. 26).

[HRR91] Nicolas Halbwachs, Pascal Raymond, and Christophe Ratel. "Generating Efficient Code From Data-Flow Programs". In: *In Third International Symposium on Programming Language Implementation and Logic Programming*. 1991, pp. 207–218 (p. 27).

[HP85] D. Harel and A. Pnueli. "On the Development of Reactive Systems". In: *Logics and Models of Concurrent Systems*. Ed. by Krzysztof R. Apt. New York, NY, USA: Springer-Verlag New York, Inc., 1985, pp. 477–498. ISBN: 0-387-15181-8. URL: http://dl.acm.org/citation.cfm?id=101969.101990 (p. 24).

[HMMC09] C. Helmstetter, F. Maraninchi, and L. Maillet-Contoz. "Full Simulation Coverage for SystemC Transaction-Level Models of Systems-on-a-Chip". In: *Formal Methods in System Design* 35.2 (2009), pp. 152–189. ISSN: 1572-8102 (p. 13).

[Hel+06] Claude Helmstetter et al. "Automatic Generation of Schedulings for Improving the Test Coverage of Systems-on-a-Chip". In: *Formal Methods in Computer Aided Design (FMCAD)*. San Jose, CA, USA: IEEE Computer Society, 2006, pp. 171–178 (pp. 13, 65, 68, 136).

[Hog16] Jim Hogan. *9 Major and 23 Minor Formal ABV Tool Metrics - Plus Their Gotchas*. 2016. URL: http://www.deepchip.com/items/0558-03.html (p. 33).

[Hop+00] John E. Hopcroft et al. *Introduction to Automata Theory, Languages and Computability*. 2nd. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2000. ISBN: 0201441241 (pp. 34, 71, 78).

[HF10] Jez Humble and David Farley. *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*. Addison-Wesley Professional, 2010. ISBN: 0321601912 (p. 133).

[HB15] Brian Hunter and Janick Bergeron. *Advanced UVM*. CreateSpace Independent Publishing Platform, 2015. ISBN: 978-1519599292 (p. 173).

[Foc] *IBM* FoCs *Property Checkers Generator*. https://www.research.ibm.com/haifa/projects/verification/focs/start.html. Online; accessed 08-November-2016 (p. 108).

[Sys] *IEEE Standard for SystemC Language Manual*. Computer Society Std. 2011 (pp. 3, 15, 21, 39, 65, 172).

[IJ04] Sasan Iman and Sunita Joshi. *The e Hardware Verification Language*. English. Dordrecht: Kluwer Academic Publishers. xxii, 349 p., 2004 (pp. 13–15, 66, 108, 134, 167, 172).

[Inc16a] Menthor Graphics Inc. *Advanced Verifiation Methodology*. Verification Methodology Cookbooks. 2016. URL: https://verificationacademy.com/cookbook/doc/glossary/advanced_verification_methodology (p. 173).

[Inc05] Menthor Graphics Inc. *Open Verifiation Methodology*. Doulos. 2005-2016. URL: https://www.doulos.com/knowhow/sysverilog/ovm/ (p. 173).

[Inc16b] Synopsys Inc. *VMM Verification Methodology*. https://www.vmmcentral.org/. [Online; accessed 3-October-2016]. 2016 (p. 172).

[Int] *Intel EP80579 Integrated Processor*. http://www.intel.com/content/www/us/en/intelligent-systems/tolapai/embedded-intel-ep80579-integrated-processor-featuring-quickassist.html. [Online; accessed 22-September-2016]. 2016 (p. 10).

[JRB06] Erwan Jahier, Pascal Raymond, and Philippe Baufreton. "Case Studies with Lurette V2". In: *Software Tools for Technology Transfer* 8.6 (2006), pp. 517–530 (p. 134).

[JJ03] Rohit Jindal and Kshitiz Jain. "Verification of Transaction-Level SystemC Models Using RTL Testbenches". In: *Proceedings of the First ACM and IEEE International Conference on Formal Methods and Models for Co-Design*. MEMOCODE '03. Washington, DC, USA: IEEE Computer Society, 2003, pp. 199–. ISBN: 0-7695-1923-7. URL: http://dl.acm.org/citation.cfm?id=823453.823851 (p. 173).

[Kal+09] M. Kallel et al. "Aspect-based ABV for SystemC transaction level models". In: *2009 International Conference on Microelectronics - ICM*. 2009, pp. 304–307 (pp. 107, 108).

[Kal+10] M. Kallel et al. "Verification of SystemC transaction level models using an aspect-oriented and generic approach". In: *Design and Technology of Integrated Systems in Nanoscale Era (DTIS), 2010 5th International Conference on*. 2010, pp. 1–6 (pp. 107, 108).

[Kam68a] Hans Kamp. "Tense Logic and the Theory of Linear Order". PhD thesis. Ucla, 1968 (p. 32).

[Kam68b] Hans W. Kamp. "Tense Logic and the Theory of Linear Order". PhD thesis. Computer Science Department, University of California at Los Angeles, USA, 1968 (p. 31).

[KT07] A. Kasuya and T. Tesfaye. "Verification Methodologies in a TLM-to-RTL Design Flow". In: *2007 44th ACM/IEEE Design Automation Conference*. 2007, pp. 199–204 (p. 173).

[Kra98] R. Kramer. "iContract - The Java(tm) Design by Contract(tm) Tool". In: *TOOLS '98: Proceedings of the Technology of Object-Oriented Languages and Systems*. Washington, DC, USA: IEEE Computer Society, 1998, p. 295. ISBN: 0-8186-8482-8 (pp. 36, 134).

[Lam78] Leslie Lamport. "Time, Clocks, and the Ordering of Events in a Distributed System". In: *Commun. ACM* 21.7 (July 1978), pp. 558–565. ISSN: 0001-0782 (p. 108).

[LD14] Hoang M. Le and Rolf Drechsler. "CRAVE 2.0: The Next Generation Constrained Random Stimuli Generator for SystemC". In: *Design and Verification Conference - Proceedings*. 2014 (pp. 14, 134).

[LGD10] Hoang M. Le, Daniel Große, and Rolf Drechsler. "Towards Analyzing Functional Coverage in SystemC TLM Property Checking". In: *HLDVT*. IEEE Computer Society, 2010, pp. 67–74 (p. 13).

[LeG+91] P. LeGuernic et al. "Programming real-time applications with SIGNAL". In: *Proceedings of the IEEE* 79.9 (1991), pp. 1321–1336. ISSN: 0018-9219 (p. 27).

[Bor] *Leveraging System Models for RTL Functional Verification*. http://www.eetimes.com/document.asp?doc_id=1271584. [Online; accessed 24-September-2016]. 2007 (pp. 4, 11).

[LP85] Orna Lichtenstein and Amir Pnueli. "Checking That Finite State Concurrent Programs Satisfy Their Linear Specification". In: *Proceedings of the 12th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. POPL '85. New Orleans, Louisiana, USA: ACM, 1985, pp. 97–107. ISBN: 0-89791-147-4 (p. 79).

[Lus] *Lustre V4 Manuals*. http://www-verimag.imag.fr/DIST-TOOLS/SYNCHRONE/lustre-v4/distrib/lv4-html/index.html. Accessed: 2016-08-20 (p. 27).

[MN04] Oded Maler and Dejan Nickovic. "Monitoring Temporal Properties of Continuous Signals". In: *Formal Techniques, Modelling and Analysis of Timed and Fault-Tolerant Systems: Joint International Conferences on Formal Modeling and Analysis of Timed Systmes, FORMATS 2004, and Formal Techniques in Real-Time and Fault -Tolerant Systems, FTRTFT 2004, Grenoble, France, September 22-24, 2004. Proceedings*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 152–166. ISBN: 978-3-540-30206-3 (p. 30).

[MP91] Zohar Manna and Amir Pnueli. *Completing the Temporal Picture*. 1991 (pp. 29, 31).

[Mar92] Florence Maraninchi. "Operational and compositional semantics of synchronous automaton compositions". In: *CONCUR '92: Third International Conference on Concurrency Theory Stony Brook, NY, USA, August 24–27, 1992 Proceedings*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1992, pp. 550–564. ISBN: 978-3-540-47293-3 (p. 25).

[MM04] Florence Maraninchi and Lionel Morel. "Logical-Time Contracts for Reactive Embedded Components". In: *In 30th EUROMICRO Conference on Component-Based Software Engineering Track, ECBSE'04*. 2004 (pp. 135, 174).

[MR01] Florence Maraninchi and Yann Rémond. "Argos: an Automaton-Based Synchronous Language". In: *Computer Languages* 27 (2001), pp. 61–92 (pp. 24–27).

[McM99] Kenneth L. McMillan. "Circular Compositional Reasoning about Liveness". In: *Correct Hardware Design and Verification Methods, 10th IFIP WG 10.5 Advanced Research Working Conference, CHARME '99, Bad Herrenalb, Germany, September 27-29, 1999, Proceedings*. 1999, pp. 342–345 (pp. 135, 174).

[McN06] Robert McNaughton. "Testing and Generating Infinite Sequences by a Finite Automaton". In: *Information and Control* 9.5 (Apr. 25, 2006), pp. 521–530. URL: http://dblp.uni-trier.de/db/journals/iandc/iandc9.html#McNaughton66 (p. 79).

[Mea55] George H. Mealy. "A method for synthesizing sequential circuits". In: *Bell System Technical Journal, The* 34.5 (1955), pp. 1045–1079. ISSN: 0005-8580 (p. 24).

[MH03] Formal Methods and Technologies Group IBM Reasearch Lab in Haifa. *FoCs. Foraml Checkers – a Productivity Tool. User's Manual*. Version Version 1.0. 2003. 107 pp. (p. 108).

[Mey92] Bertrand Meyer. "Applying "Design by Contract"". In: *Computer* 25.10 (Oct. 1992), pp. 40–51. ISSN: 0018-9162 (pp. 36, 134).

[Mey97] Bertrand Meyer. *Object-oriented Software Construction (2Nd Ed.)* Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1997. ISBN: 0-13-629155-4 (p. 36).

[Mey14] Scott Meyers. *Effective Modern C++: 42 Specific Ways to Improve Your Use of C++11 and C++14*. O'Reilly Media. 336 p., 2014. ISBN: 978-1491903995 (p. 18).

[Mil80] Robin Milner. *A Calculus of Communicating Systems*. Vol. 92. Lecture Notes in Computer Science. Springer, 1980. ISBN: 3-540-10235-3 (p. 28).

[Mil83] Robin Milner. "Calculi for Synchrony and Asynchrony". In: *Theor. Comput. Sci.* 25 (1983), pp. 267–310 (p. 28).

[MVS08] M. Moadeli, W. Vanderbauwhede, and A. Shahrabi. "Quarc: A novel network-on-chip architecture". In: *Proceedings of the International Conference on Parallel and Distributed Systems - ICPADS* December (2008), pp. 705–712. ISSN: 15219097 (p. 11).

[Mor+04] Fernando Moraes et al. "HERMES: an infrastructure for low area overhead packet-switching networks on chip". In: *Integration, the {VLSI} Journal* 38.1 (2004), pp. 69 –93. ISSN: 0167-9260. URL: http://www.sciencedirect.com/science/article/pii/S0167926004000185 (p. 11).

[MAB07a] K. Morin-Allory and D. Borrione. "On-line monitoring of properties built on regular expressions sequences". In: *Advances in Design and Specification Languages for Embedded Systems (Selected Contributions from FDL'06)*. Ed. by Sorin A. Huss. ISBN :978-1-4020-6147-9. Springer, 2007, pp. 197–207. URL: https://hal.archives-ouvertes.fr/hal-00229249 (p. 66).

[MAB06] K. Morin-Allory and D. Borrione. "Proven correct monitors from PSL specifications". In: *Proceedings of the Design Automation Test in Europe Conference*. Vol. 1. 2006, pp. 1–6 (pp. 90, 107, 171).

[MAB07b] Katell Morin-Allory and Dominique Borrione. "Online Monitoring of Properties Built on Regular Expressions Sequences". In: *Advances in Design and Specification Languages for Embedded Systems: Selected Contributions from FDL'06*. Ed. by Sorin A. Huss. Dordrecht: Springer Netherlands, 2007, pp. 197–207. ISBN: 978-1-4020-6149-3 (pp. 107, 108, 171).

[MGB07] Katell Morin-Allory, Eric Gascard, and Dominique Borrione. "Synthesis of Property Monitors for Online Fault Detection". In: *Journal of Circuits, Systems, and Computers* 16.6 (2007), pp. 943–960 (p. 171).

[NHd06] B. Niemann, Ch. Haubelt, and Integrierte Schaltungen Hardware software-co design. *Assertion-Based Verification of Transaction Level Models*. 2006 (pp. 107, 108).

[Oli+12] Marcio F.S. Oliveira et al. "The System Verification Methodology for Advanced TLM Verification". In: *Proceedings of the Eighth IEEE/ACM/IFIP CODES+ISSS*. Tampere, Finland: ACM, 2012, pp. 313–322. ISBN: 978-1-4503-1426-8 (pp. 134, 173).

[Opea] *Open Source Computer Vision Library (OpenCV)*. http://opencv.org/. Accessed: 2016-07-05 (p. 44).

[Opeb] *Open SystemC Initiative, SystemC Verification Library*. v2. 2014. URL: http://www.accellera.org/downloads/standards/systemc (pp. 14, 134).

[Synb] *OpenVera Language Reference Manual: Testbench, Synopsys Inc.* http://www.open-vera.com/. Online; accessed 09-November-2016. 2016 (p. 108).

[Per+04] Prakash Mohan Peranandam et al. "Transactional Level Verification and Coverage Metrics by Means of Symbolic Simulation." In: *MBMV*. Ed. by Dominik Stoffel and Wolfgang Kunz. Shaker, 2004, pp. 260–269 (p. 12).

[PF10] L. Pierre and L. Ferro. "Enhancing the assertion-based verification of TLM designs with reentrancy". In: *Eighth ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE 2010)*. 2010, pp. 103–112 (pp. 169, 171).

[Pie+12] L. Pierre et al. "Integrating PSL properties into SystemC transactional modeling – Application to the verification of a modem SoC". In: *7th IEEE International Symposium on Industrial Embedded Systems (SIES'12)*. 2012, pp. 220–228 (p. 171).

[Pie07] Laurence Pierre. *A Model for Assertion-Based Verification of TLM Designs*. Tech. rep. TIMA Laboratory, 2007 (pp. 66, 109, 126, 127, 169).

[PA13] Laurence Pierre and Zeineb Bel Hadj Amor. "Automatic Refinement of Requirements for Verification Throughout the SoC Design Flow". In: *Proceedings of the International Conference on Hardware/Software Codesign and System Synthesis, CODES+ISSS 2013, Montreal, QC, Canada, September 29 - October 4, 2013*. IEEE, 2013, pp. 1–10 (p. 173).

[PF08] Laurence Pierre and Luca Ferro. "A Tractable and Fast Method for Monitoring SystemC TL Specifications". In: *IEEE Transactions on Computers* 57.10 (2008), pp. 1346–1356. ISSN: 0018-9340 (pp. 3, 66, 75, 87, 90, 108, 125–127, 168, 169, 171).

[PZ06] A. Pnueli and A. Zaks. "PSL Model Checking and Run-Time Verification Via Testers". In: *FM 2006: Formal Methods: 14th International Symposium on Formal Methods, Hamilton, Canada, August 21-27, 2006. Proceedings*. Ed. by Jayadev Misra, Tobias Nipkow, and Emil Sekerinski. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 573–586. ISBN: 978-3-540-37216-5 (p. 170).

[Pnu77] Amir Pnueli. "The Temporal Logic of Programs". In: *Proceedings of the 18th Annual Symposium on Foundations of Computer Science*. SFCS '77. Washington, DC, USA: IEEE Computer Society, 1977, pp. 46–57 (p. 31).

[Pop+10] Katalin Popovici et al. "Embedded Systems Design: Hardware and Software Interaction". In: *Embedded Software Design and Programming of Multiprocessor System-on-Chip: Simulink and System C Case Studies*. New York, NY: Springer New York, 2010, pp. 1–48. ISBN: 978-1-4419-5567-8 (p. 9).

[Rac+11] Jean-Baptiste Raclet et al. "A Modal Interface Theory for Component-Based Design". In: *Fundamenta Informaticae* 108.1 (2011), pp. 119–149 (p. 174).

[Ray+98] P. Raymond et al. "Automatic Testing of Reactive Systems". In: *19th IEEE Real-Time Systems Symposium*. Madrid, Spain, Dec. 1998 (p. 134).

[Ray91] Pascal Raymond. "Compilation efficace d'un langage déclaratif synchrone : le générateur de code Lustre-V3. (Efficient Compilation of a Declarative Synchronous Language:the Lustre-V3 Code Generator)". PhD thesis. Grenoble Institute of Technology, France, 1991. URL: https://tel.archives-ouvertes.fr/tel-00198546 (p. 27).

[Ray10] Pascal Raymond. "Synchronous Program Verification with Lustre/Lesar". In: *Modeling and Verification of Real-Time Systems*. ISTE, 2010, pp. 171–206. ISBN: 9780470611012 (pp. 27, 28).

[Ris11] Tanguy Risset. "SoC (System on Chip)". In: *Encyclopedia of Parallel Computing*. Boston, MA: Springer US, 2011, pp. 1837–1842. ISBN: 978-0-387-09766-4 (p. 9).

[RH92] Frédéric Rocheteau and Nicolas Halbwachs. "Implementing reactive programs on circuits a hardware implementation of LUSTRE". In: *Real-Time: Theory in Practice: REX Workshop Mook, The Netherlands, June 3–7, 1991 Proceedings*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1992, pp. 195–208. ISBN: 978-3-540-47218-6 (p. 27).

[RGH94] Frédéric Rocheteau, Imag lgi Grenoble, and Nicolas Halbwachs. *POLLUX: A LUSTRE based hardware design environment*. 1994 (p. 27).

[RM16] Yuliia Romenska and Florence Maraninchi. "Efficient Monitoring of Loose-Ordering Properties for SystemC TLM". In: *Design, Automation, and Test in Europe (DATE)*. Dresden, Germany, 2016 (p. 6).

[RM13] Sharon Rosenberg and Kathleen Meade. *A Practical Guide to Adopting the Universal Verification Methodology (UVM)*. lulu.com, 2013. ISBN: 978-1300535935 (p. 173).

[RDS92] Valérie Roy and Robert De Simone. "Auto/Autograph". In: *Formal Methods in System Design* 1.2 (1992), pp. 239–249. ISSN: 1572-8102 (pp. 27, 28).

[Rub12] Kenneth S. Rubin. *Essential Scrum: A Practical Guide to the Most Popular Agile Process*. Addison-Wesley Professional, 2012. ISBN: 0137043295 (p. 133).

[Sdl] *Simle DirectMedia Layer (SDL) Development Library*. https://www.libsdl.org/. Accessed: 2016-07-05 (p. 44).

[ST12] Chris Spear and Greg Tumbush. *SystemVerilog for Verification: A Guide to Learning the Testbench Language Features*. Springer, 2012. ISBN: 978-1461407140 (pp. 30, 33, 172).

[SGSP02] Olaf Spinczyk, Andreas Gal, and Wolfgang Schröder-Preikschat. "AspectC++: An Aspect-oriented Extension to the C++ Programming Language". In: *Proceedings of the Fortieth International Conference on Tools Pacific: Objects for Internet, Mobile and Embedded Applications*. CRPIT '02. Sydney, Australia: Australian Computer Society, Inc., 2002, pp. 53–60. ISBN: 0-909925-88-7. URL: http://dl.acm.org/citation.cfm?id=564092.564100 (p. 108).

[SSF06] Simon Davidmann Stuart Sutherland and Peter Flake. *SystemVerilog for Design Second Edition: A Guide to Using SystemVerilog for Hardware Design and Modeling*. Springer, 2006. ISBN: 978-0387333991 (pp. 30, 33).

[Swi93] Robert M. Switzer. *Eiffel, an introduction*. New York: Prentice Hall, 1993. ISBN: 0-13-105909-2. URL: http://opac.inria.fr/record=b1099588 (pp. 36, 134).

[Syn05] Synopsys. *Reference Verification Methodology User Guide*. 2005 (p. 172).

[TV10] D. Tabakov and M. Y. Vardi. "Monitoring temporal SystemC properties". In: *Eighth ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE 2010)*. 2010, pp. 123–132. DOI: 10.1109/MEMCOD.2010.5558640 (p. 170).

[TRV12] Deian Tabakov, Kristin Y. Rozier, and Moshe Y. Vardi. "Optimized Temporal Monitors for SystemC". In: *Formal Methods in System Design* 41.3 (2012), pp. 236–268. DOI: DOI: 10.1007/s10703-011-0139-8. URL: http://www.springerlink.com/content/lq16400q55t72k16/ (p. 170).

[Tab+08] Deian Tabakov et al. "A Temporal Language for SystemC". In: *Formal Methods in Computer-Aided Design, 2008. FMCAD'08*. IEEE. 2008, pp. 1–9 (pp. 168, 170).

[TA12] Andrew S. Tanenbaum and Todd Austin. *Structured Computer Organization (6th Edition)*. Pearson. 800 p., 2012. ISBN: 978-0132916523 (p. 10).

[Tom+09] K. Tomasena et al. "A Transaction Level Assertion Verification Framework in SystemC: An Application Study". In: *Advances in Circuits, Electronics and Micro-electronics, 2009. CENICS '09. Second International Conference on*. 2009, pp. 75–80 (pp. 3, 90, 140, 168–170).

[Var96] Moshe Y. Vardi. "An automata-theoretic approach to linear temporal logic". In: *Logics for Concurrency: Structure versus Automata, volume 1043 of Lecture Notes in Computer Science*. Springer-Verlag, 1996, pp. 238–266 (p. 79).

[Vas06] Srivatsa Vasudevan. *Effective Functional Verification: Principles and Processes*. Springer, 2006. ISBN: 978-1588298683 (p. 12).

[Vas16] Srivatsa Vasudevan. *Practical UVM*. CreateSpace, 2016. ISBN: 978-0997789607 (p. 173).

[Ver86] Didier Vergamini. *Verification by means of observational equivalence on automata*. Tech. rep. RR-0501. INRIA, 1986. URL: http://opac.inria.fr/record=b1013901 (p. 27).

[Erm] *Verisity Design e Reuse Methodology Developer Manual*. 2002-2004 (pp. 134, 172).

[Wei+05] R. J. Weiss et al. "Efficient and customizable integration of temporal properties into systemc". In: *Forum on Specification and Design Languages (FDL)*. 2005 (p. 12).

[Wil+09] Robert Wille et al. "SMT-Based Stimuli Generation in the SystemC Verification Library". In: *FDL*. IEEE, 2009, pp. 1–6 (p. 14).

[Wil+07] Robert Wille et al. "SWORD: A SAT like Prover Using Word Level Information". In: *VLSI-SoC: Advanced Topics on Systems on a Chip - A Selection of Extended Versions of the Best Papers of the Fourteenth International Conference on Very Large Scale Integration of System on Chip (VLSI-SoC2007), October 15-17, 2007, Atlanta, USA*. 2007, pp. 1–17 (p. 14).

[Wol81] Pierre Wolper. "Temporal logic can be more expressive". In: *2013 IEEE 54th Annual Symposium on Foundations of Computer Science* (1981), pp. 340–348 (p. 80).

[Xil] *Xilinx OPB Interrupt Conctroller (v1.00c). Product Specification*. Xilinx. 2005 (p. 44).

[XBZ10] Zhaorong Xiong, Jinian Bian, and Yanni Zhao. "An assertion-based verification method for SystemC TLM". In: *Communications, Circuits and Systems (ICCCAS), 2010 International Conference on*. 2010, pp. 842–846 (p. 169).

[YWD14] Shuo Yang, Robert Wille, and Rolf Drechsler. "Determining Cases of Scenarios to Improve Coverage in Simulation-based Verification". In: *Proceedings of the 27th Symposium on Integrated Circuits and Systems Design, Aracaju, Brazil, September 1-5, 2014*. 2014, p. 11 (p. 13).

Abstract

The work presented in this thesis deals with modeling, specification and testing of models of Systems-on-a-Chip (SoCs) at the transaction abstraction level and higher. SoCs are heterogeneous: they comprise both hardware components and processors to execute embedded software, which closely interacts with hardware. SystemC-based Transaction Level Modeling (TLM) has been very successful in providing high-level executable component-based models for SoCs, also called *virtual prototypes* (VPs). These models can be used early in the design flow for the development of the software and the validation of the actual hardware. For SystemC/TLM virtual prototypes, Assertion-Based Verification (ABV) allows property checking early in the design cycle, helping to find bugs early in the model and to save time and effort that are needed for their fixing. TL models can be *over-constrained*, which means that they do not represent all the behaviors of the hardware, and thus, do not allow detection of some malfunctions of the prototype. Our contributions consist of two orthogonal and complementary parts: On the one hand, we identify sources of over-constraints in TL models appearing due to the order of interactions between components, and propose a notion of *loose-ordering* which allows to remove these over-constraints. On the other hand, we propose a *generalized stubbing mechanism* which allows the very early simulation with SystemC/TLM virtual prototypes.

We propose a set of patterns to capture loose-ordering properties, and define a direct translation of these patterns into SystemC monitors. Our generalized stubbing mechanism enables the early simulation with SystemC/TLM virtual prototypes, in which some components are not entirely determined on the values of the exchanged data, the order of the interactions and/or the timing. Those components have very abstract specifications only, in the form of constraints between inputs and outputs. We show that essential synchronization problems between components can be captured using our simulation with stubs. The mechanism is *generic*; we focus only on key concepts, principles and rules which make the stubbing mechanism implementable and applicable for real, industrial case studies. Any specification language satisfying our requirements (e.g., loose-orderings) can be used to specify the components, i.e., it can be plugged in the stubbing framework. We provide a proof of concept to demonstrate the interest of using the simulation with stubs for very early detection and localization of synchronization bugs of the design.

Keywords. System-on-a-Chip (SoCs), Transaction Level Modeling (TLM), Assertion-Based Verification (ABV), Contracts and Specifications, Stubbing, Early Simulation.

Résumé

Les travaux présentés dans cette thèse portent sur la modélisation, la spécification et la vérification des modèles des Systèmes sur Puce (SoCs) au niveau d'abstraction transactionnel et à un niveau d'abstraction plus élevé. Les SoCs sont hétérogènes: ils comprennent des composants matériels et des processeurs pour réaliser le logiciel incorporé, qui est en lien direct avec du matériel. La modélisation transactionnelle (TLM) basée sur SystemC a été très fructueuse à fournir des modèles exécutables des SoCs à un haut niveau d'abstraction, aussi appelés *prototypes virtuels* (VPs). Ces modèles peuvent être utilisés plus tôt dans le cycle de développement des logiciels, et la validation des matériels réels. La vérification basée sur assertions (ABV) permet de vérifier les propriétés tôt dans le cycle de conception de façon à trouver les défauts et faire gagner du temps et de l'effort nécessaires pour la correction de ces défauts. Les modèles TL peuvent être sur-contraints, c'est-à-dire qu'ils ne presentent pas tous les comportements du matériel. Ainsi, ceci ne permet pas la détection de tous les défauts de la conception. Nos contributions consistent en deux parties orthogonales et complémentaires: D'une part, nous identifions les sources des sur-contraintes dans les modèles TLM, qui apparaissent à cause de l'ordre d'interaction entre les composants. Nous proposons une notion *d'ordre mou* qui permet la suppression de ces sur-contraintes. D'autre part, nous présentons un *mécanisme généralisé de stubbing* qui permet la simulation précoce avec des prototypes virtuels SystemC/TLM.

Nous offrons un jeu de patrons pour capturer les propriétés d'ordre mou et définissons une transformation directe de ces patrons en moniteurs SystemC. Notre mécanisme généralisé du stubbing permet la simulation précoce avec les prototypes virtuels SystemC/TLM, dans lesquels certains composants ne sont pas entièrement déterminés sur les valeurs des données échangées, l'ordre d'interaction et/ou le timing. Ces composants ne possèdent qu'une spécification abstraite, sous forme de contraintes entre les entrées et les sorties. Nous montrons que les problèmes essentielles de la synchronisation entre les composants peuvent être capturés à l'aide de notre simulation avec les stubs. Le mécanisme est *générique*; nous mettons l'accent uniquement sur les concepts-clés, les principes et les règles qui rendent le mécanisme de stubbing implémentable et applicable aux études de cas industriels. N'importe quel language de spécification satisfaisant nos exigences (par ex. le langage des ordres mou) peut être utilisé pour spécifier les composants, c'est-à-dire il peut être branché au framework de stubbing. Nous fournissons une preuve de concept pour démontrer l'intérêt d'utiliser la simulation avec stubs pour la détection anticipée et la localisation des défauts de synchronisation du modèle.

Mots Clés. Systèmes sur Puce (SoCs), Modélisation au Niveau Transactionnel (TLM), Vérification Basée sur Assertions (ABV), Contracts et Spécifications, Stubbing, Simulation Précoce.