Techniques and Tools for the Verification of Systems-on-a-Chip at the Transaction Level

Matthieu Moy

▶ To cite this version:

Matthieu Moy. Techniques and Tools for the Verification of Systems-on-a-Chip at the Transaction Level. Computer Science [cs]. Institut National Polytechnique de Grenoble - INPG, 2005. English. NNT : . tel-00311033

HAL Id: tel-00311033
https://theses.hal.science/tel-00311033

Submitted on 12 Aug 2008

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.

INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE

T H E S I S

To obtain the grade of

INPG DOCTOR

Speciality: « COMPUTER SYSTEMS AND COMMUNICATIONS »
(INFORMATIQUE : SYSTÈMES ET COMMUNICATION)

Prepared in the VERIMAG laboratory
Graduate school « **MATHEMATICS AND COMPUTER SCIENCE** »
(MATHÉMATIQUES, SCIENCES ET TECHNOLOGIES DE L'INFORMATION, INFORMATIQUE)

presented and defended publicly

by

Matthieu MOY

on December, 9th 2005

Title:

Techniques and Tools for the Verification of Systems-on-a-Chip at the Transaction Level

Supervised by

Florence Maraninchi

JURY

Gérard Michel	President
Stephen Edwards	Reviewer
Jean-Pierre Talpin	Reviewer
Florence Maraninchi	Director
Laurent Maillet-Contoz	Examiner

Acknowledgments

First of all, I would like to express my gratitude to the members of my thesis jury:

Gérard Michel, director of the INPG department of Telecommunications, to have accepted to be the president of my thesis jury.

Jean-Pierre Talpin, Scientific leader in INRIA Rennes and *Stephen A. Edwards*, assistant professor in the Computer Science Department of Columbia University, to have accepted to be reviewers for this work. To be a reviewer, for those who don't know, means reading the full document with a critical mind, look for problems and potential enhancements, and is a considerable amount of work. A wink at the first for the few anecdotes stories, reassuring (or not) about Ph.D presentations, and a special thank for the second, who is coming from New York for my presentation. Both of them gave me a few precious advices about the organization of my Ph.D document.

Laurent Maillet-Contoz, project manager at STMicroelectronics, who supervised me all along the thesis, from the definition of the subject (whereas the words "SystemC" and "TLM" did not mean anything for me!) until the last corrections of the documents, and the presentation. Once again, for those who don't know, a project manager is someone having an important load of work, with strong solicitations and short deadlines from the clients and his own manager (do "the project to be finished day before yesterday" evoke something for you?), but still, he took the time to take part in our research activities. It is in particular thanks to Laurent that we could carry out a research work with a strong link to reality.

Florence Maraninchi, Professor in Institut National Polytechnique de Grenoble to have supervised this thesis. You'll certainly feel like I'm repeating myself, but supervising a Ph.D is a big amount of work, and means a big amount of time to spend. Actually, not everybody is lucky enough to have a Ph.D director who takes the time to supervise correctly, but Florence is likes the work well done regarding this. I have learned a lot during those three years, scientifically speaking as well as regarding the way to write a paper, to increase the standing of my own work, ...

Generally, I would like to thank all the people involved in the supervising of this thesis, to have at the same time provided me the means to work and enough freedom to do a "real" Ph.D. *Frank Ghenassia* and *Alain Clouard*, respectively manager and deputy manager of the SPG team, are probably the persons to thank most for this. Some people may think that CIFRE (french thesis in cooperation with the industry) Ph.D students are not more than cheap engineers, but this is not the case with Frank and Alain. I also thank them to have given me the opportunity to meet other interesting personalities inside STMicroelectronics, such as *Andrea Fedeli*, specialist of formal verification tools, who kept an eye on my thesis, *Marc Benveniste*, interested in particular in interactive theorem proving tools, and *Laurent Ducousso*, who explained me the problems that his team were facing regarding the validation of designs.

I am grateful to the Verimag laboratory as a whole. It was a very pleasant framework during the three years of this Ph.D, an also the year before, for my master degree. I would cite in particular *Joseph Sifakis*, director of the laboratory, the secretaries (*Elisabeth*, *Martine*, *Valérie*, *Christine* and *Anne*, who make administrative tasks pleasant), and *Nicolas Kowalski*, the system administrator faster than his shadow, now assisted by *Jean-Noël Bouvier*.

I would like to thank equally those who made this thesis possible. *Éric Rutten* is somewhat the grandfather of this Ph.D, being the one who started discussions with STMicroelectronics, which gradually took the form of a thesis proposal. *Verimag's coffee machine*, which is to my knowledge the best tool to find a job for computer science students (whether it is to find a Ph.D or a post-doc).

Off course, I thank all my colleagues. *Laure*, friendly office neighbor, and occasionally organizer of barbecues for Ph.D students, *Jérôme*, another office neighbor, for the numerous discussions, vigorous but

nevertheless very interesting. *Romain*, the last occupant of our Verimag office, with whom I shared a few stressing moments by the end of the writing of this document ("do you plan to finish before Christmas?"). *Claude*, who, apart from the long technical discussions, made me discover the nuts beer made in Grenoble. *Alain Kauffman*, who set up a tradition of tea breaks in ST (I've been told this was not very "corporate" . . .). More generally, all the members of the "synchronous" team in Verimag and SPG in ST, who would be too numerous to be enumerated.

Finally, I thank my family to have supported me in some difficult moments (if you have ever met a Ph.D student during the end of the writing of his report, you probably know what I'm talking about).

Contents

I An approach for the Verification of Systems-on-a-Chip 9

1 Introduction 11
1.1 Technical Context . . . 11
1.2 General Objectives for the Thesis . . . 12
1.3 Work Context and Motivations . . . 13
1.3.1 Motivations in the Industrial Context of STMicroelectronics . . . 13
1.3.2 Motivations in the Academic Context of Verimag . . . 14
1.4 Approach and Technical Choices . . . 14
1.5 Summary of Contributions and Limitations of the Approach . . . 14
1.6 Outline of the Document . . . 15

2 Modeling Systems-on-a-Chip at the Transaction Level in SystemC 17
2.1 Introduction . . . 17
2.2 The Systems-on-a-Chip Design Flow . . . 18
2.2.1 Hardware – Software Partitioning . . . 18
2.2.2 Different Levels of Abstraction . . . 19
2.2.3 Verification, Validation, Test . . . 21
2.3 The Transaction Level Model . . . 22
2.3.1 Example of a TLM platform . . . 22
2.3.2 TLM Concepts and Terminology . . . 22
2.3.3 Importance of TLM in the Design Flow . . . 23
2.4 SystemC and the TLM API . . . 24
2.4.1 Need for a new "language" . . . 24
2.4.2 The SystemC Library . . . 25
2.4.3 TLM in SystemC . . . 30
2.5 A Larger Example: The EASY Platform . . . 33
2.5.1 Description of the Platform . . . 33
2.5.2 A TLM model for EASY . . . 34

3 Overview of LusSy: A Toolbox for the Analysis of Systems-on-a-Chip at the Transaction Level 37
3.1 Introduction . . . 37
3.2 Techniques and Tools for the Verification of SoCs . . . 38
3.2.1 Algorithms for Formal Verification . . . 38
3.2.2 Candidate Tools for the Verification of SoCs . . . 41
3.3 Our Verification Approach . . . 42
3.3.1 Expressing Properties . . . 42
3.3.2 Synchronization Code Vs. Complex Algorithms . . . 43
3.3.3 The LusSy Tool Chain . . . 43

II Automatic Extraction of Formal Semantics of SystemC Models **47**

4 PINAPA: Extracting Architecture and Behavior Information From SystemC Models **49**
4.1 Introduction . . . 49
4.1.1 Static and dynamic information in SystemC . . . 50
4.1.2 PINAPA: Requirements . . . 50
4.1.3 Contributions . . . 50
4.2 Related Work . . . 51
4.2.1 Existing SystemC Tools . . . 51
4.2.2 Other combinations of static and dynamic analyzers . . . 52
4.3 PINAPA Principles, Limitations and Uses . . . 52
4.3.1 Specifications of PINAPA . . . 52
4.3.2 Limitations . . . 58
4.3.3 Other possible approaches . . . 59
4.3.4 Pinapa: Current and Future Uses . . . 59
4.4 Implementation of PINAPA . . . 60
4.4.1 Links from ELAB to AST . . . 60
4.4.2 Links from AST to ELAB . . . 60
4.4.3 Architecture of PINAPA . . . 62
4.4.4 Modifications of GCC and SystemC . . . 65
4.4.5 Practical Considerations . . . 66
4.5 Parsing the EASY Platform With PINAPA . . . 67
4.6 Conclusion . . . 68

5 HPIOM: Heterogeneous Parallel Input/Output Machines **69**
5.1 Introduction . . . 69
5.1.1 The Need For an Intermediate Formalism . . . 69
5.1.2 Design Choices . . . 70
5.1.3 Contents of the chapter . . . 70
5.2 HPIOM Basic Concepts . . . 71
5.2.1 Syntax of HPIOM . . . 71
5.2.2 Semantics of HPIOM . . . 74
5.2.3 Non-determinism . . . 76
5.3 Additional constructs . . . 76
5.3.1 Convenience constructs . . . 76
5.3.2 Abstract Addresses . . . 77
5.4 Expression of Properties in HPIOM . . . 81
5.5 Implementation of and Manipulation of HPIOM . . . 82
5.5.1 HPIOM Expressions and the Composite Design Pattern . . . 82
5.5.2 HPIOM Expressions and the Visitor Design Pattern . . . 83
5.5.3 On-the-fly Optimizations . . . 88
5.5.4 Future improvements . . . 90

6 BISE: Semantics of SystemC and TLM Constructs in Terms of Automata **91**
6.1 Introduction . . . 91
6.1.1 Principles . . . 92
6.1.2 Expressiveness of HPIOM, Abstractions, and Limitations of BISE . . . 93
6.2 Semantics of process code into HPIOM . . . 93
6.2.1 Simple Operations and Sequence . . . 93
6.2.2 Control Structures . . . 94
6.3 Semantics of the Synchronization Primitives and the Scheduler . . . 95
6.3.1 The SystemC Scheduling Mechanism . . . 95
6.3.2 Model for the `sc_event` . . . 98
6.4 Communication Mechanisms . . . 98

6.4.1 SystemC Signal . . . 98
6.4.2 Direct Semantics of TLM Constructs . . . 99
6.4.3 Unmanaged SystemC Constructs . . . 105
6.5 Encoding Properties . . . 105
6.5.1 Assertions . . . 105
6.5.2 Multiple `write` on a Signal During the Same δ-cycle . . . 105
6.5.3 Process Termination . . . 106
6.5.4 Mutual exclusion . . . 106
6.5.5 Concurrent TLM Port Accesses . . . 107
6.6 Related Work . . . 107
6.6.1 Other Formal Semantics for SystemC . . . 107
6.6.2 SC2PROM . . . 107
6.6.3 SystemC Translation Into Signal . . . 107
6.7 Conclusion . . . 108

III Using HPIOM for Formal Verification **109**

7 BIRTH: Back-end Independent Reduction and Transformations of HPIOM **111**
7.1 Introduction . . . 111
7.2 Results of BIRTH on Some Examples, Without Optimization . . . 112
7.3 Semantic Preserving Transformations . . . 113
7.3.1 Abstract Addresses Expansion . . . 113
7.3.2 Non-Deterministic Choices Expansion . . . 113
7.3.3 Reducing the Number of Variables and Inputs . . . 113
7.3.4 Reduce the Number of States by Parallelizing Transitions . . . 117
7.4 An Approximation Conservative for Safety Properties: Abstracting Away Numerical Values 118
7.5 Non-conservative Approximations . . . 118
7.5.1 Initialize variables deterministically . . . 119
7.5.2 Limit the Depth of the Proof . . . 119
7.5.3 Specify the Initial State Manually . . . 119
7.6 Conclusion . . . 120

8 Back-Ends: Connecting HPIOM to Verification Tools **121**
8.1 Introduction . . . 121
8.2 Presentation of the Verification Tools . . . 122
8.2.1 The LUSTRE Tool-Chain . . . 122
8.2.2 SMV Language and Model-Checker . . . 127
8.3 Encoding in LUSTRE . . . 129
8.3.1 Encoding State Machines in a Data-Flow Language, the Case of LUSTRE . . . 129
8.3.2 Encoding Variables . . . 131
8.3.3 Encoding HPIOM Communication in LUSTRE . . . 131
8.3.4 Results of the LUSTRE Generator for the EASY Platform . . . 131
8.4 Encoding in SMV . . . 131
8.4.1 Defining a Sequence: `next` Vs `pre` . . . 131
8.4.2 State Machines . . . 132
8.4.3 Encoding Variables . . . 133
8.4.4 HPIOM Communication in SMV . . . 133
8.4.5 Results of the SMV Generator for the EASY Platform . . . 134
8.5 Validation and Debugging . . . 134
8.5.1 Validating one Back-End at a Time . . . 134
8.5.2 Comparing LUSTRE and SMV Code to SystemC . . . 135
8.5.3 Comparing the Back-Ends . . . 135
8.6 Tools Comparison . . . 136

8.6.1 Input Languages . . . 136
8.6.2 Performances of the Proof Engines . . . 137
8.7 Validating Real-Life Platforms . . . 142

Conclusion and Perspectives **145**

9 Conclusion **145**
9.1 Context . . . 145
9.2 Results and Discussion . . . 146
9.2.1 Contributions of the Thesis . . . 146
9.2.2 The Choice of SystemC . . . 148

10 Perspectives **151**
10.1 Possible Improvements and Uses for LUSSY . . . 151
10.2 Towards a Complete Development Environment for SystemC . . . 152

Appendixes **155**

A Example of translation into HPIOM: `switch` and `while` statement **155**
A.1 Introduction . . . 155
A.2 The platform . . . 155
A.3 C++ constructs . . . 157
A.3.1 `while` Loop . . . 157
A.3.2 `switch` Statement . . . 158
A.4 SystemC Constructs . . . 162
A.4.1 Processes and the Scheduler . . . 162
A.4.2 Communication: the Example of the `sc_signal` . . . 165
A.5 Global Picture . . . 165
A.5.1 Main Files . . . 165

B Why Formal Languages Should be Simple . . . and Formal **171**
B.1 Introduction . . . 171
B.1.1 Formal and Informal Languages . . . 171
B.1.2 Formal Proof, 100% Safety? . . . 172
B.2 The Rudimentary Approach: LUSTRE . . . 172
B.3 A Higher Level Language: SMV . . . 172
B.4 An Example of Problem Using Higher Level Language . . . 173
B.4.1 The Problem . . . 173
B.4.2 Trying to Understand . . . 173
B.4.3 Consequence on LUSSY . . . 174
B.5 Executability and Determinism: a Way to Clarify the Semantics of Informal Languages . . . 175
B.6 Conclusion . . . 175

Bibliography **182**

Index **183**

Part I

An approach for the Verification of Systems-on-a-Chip

Chapter 1

Introduction

Contenu

1.1 Technical Context **11**
1.2 General Objectives for the Thesis **12**
1.3 Work Context and Motivations **13**
1.3.1 Motivations in the Industrial Context of STMicroelectronics 13
1.3.2 Motivations in the Academic Context of Verimag 14
1.4 Approach and Technical Choices **14**
1.5 Summary of Contributions and Limitations of the Approach **14**
1.6 Outline of the Document **15**

1.1 Technical Context

In 1980, Gordon Moore, Engineer at Fairchild Semiconductor, one of the two co-founders of Intel, stated a law according to which the number of transistors on silicon chips was doubling every eighteen months. Although it is not a real physical law, this prediction revealed to be incredibly exact. Between 1971 and 2001, the density of transistors actually doubled every 1,96 years. As a consequence, electronic devices became more and more powerful and less and less costly.

Surprisingly, this evolution was accompanied by a miniaturization quite as continuous. The supercomputers of the 1960's filled in a complete room, weighted several tons, for a computing capacity of only a few thousands of instructions per second. The ATLAS, for example, was the most powerful computer in the world in 1964, with two hundred thousands of instructions per second. This order of magnitude of computing power gradually became the standard for micro-computers, calculators, and more and more miniature objects which are now part of our daily life.

While it has been possible to envisage a miniaturization of the computers for a long time, very few people had imagined that the miniaturization would reach such a point. In March 1949, for example, the magazine "popular mechanics" wrote: "Where a calculator like the ENIAC today is equipped with 18,000 vacuum tubes and weighs 30 tons, computers in the future may have only 1,000 vacuum tubes and perhaps weigh only 1.5 tons.".

Today, the specialists in the domain think that nothing should prevent Moore's law from continuing for at least the next 10 years. If the current evolution is extrapolated, the size of a transistor should reach the scale of the atom in the years 2020. It is probable that Moore's law cease to apply at this point. The speed of computers will then either stagnate, or leap forward with the arrival of quantum computers.

While the physical limits of miniaturization and computing speed appear to be far from being reached, other significant limitations are already coming into the picture today. The number of transistors, the size of the memories, the frequency of the chips grow in an exponential way with time but other parameters are also growing in the same proportions. For instance, the cost of a micro-electronics factory, the fixed costs

for the manufacture of a chip, and its electric consumption are also growing exponentially with time. This growth is mainly compensated by the growth of the market, which allows to share the fixed costs between a larger number of chips, so that the total cost of each of them remains reasonable, but one should not neglect that the growth of the market cannot be infinite.

Another limitation is the amount of code that the designers are able to write in a reasonable time. Being able to increase the number of transistors on a chip is really interesting only if one is also able to write the code which once synthesized, will give the organization of these transistors on the chip. With traditional techniques, the productivity of the designers increases by only 30% a year, whereas the number of transistors can increase by 50%. Without major innovation, the difference between the productivity of the design and the capacity of the chips would thus increase approximately 20% per year. This divergence is generally referred to as the “design gap”.

New techniques must consequently be set up continuously to fill in this gap. Tools for automatic synthesis for the Register Transfer Level (RTL) to the gate-level have been a real revolution, but is not sufficient anymore today.

The repartition of the functionalities between software and hardware solves part of the problems of the “design gap”. It makes it possible to re-use the same chip in different contexts, thus sharing the fixed costs between several projects. Programmable generic components, ranging from the entirely generic processor to the DSP (Digital Signal Processor) and including the micro-controllers, can be used to design the chip. It is generally admitted that a given functionality is approximately implemented ten times faster in software than in hardware (but on the other hand, the performances of the software are not as good as ones of hardware).

A new category of systems has been emerging during the last few years, including one or more processors, dedicated components and input/output modules, the whole fitting on a single chip. In other words, current technologies make it possible to implement in a single chip the equivalent of the contents of a computer motherboard. These systems are called *System-on-a-Chip* (SoCs).

The design of a System-on-a-Chip is thus a joint development of software and hardware. The hardware only makes sense with its corresponding embedded software, and the software needs the hardware to run. The constraints of the market force the development cycle of a chip to remain as short as possible, and it is not acceptable to wait for the chip to physically exist to start the development of the corresponding software. The development of the software has therefore to be carried out, or at least started, on a simulator.

A solution would be to use the synthesizable version of the hardware part of the chip as a simulator on which to develop the embedded software. This approach (RTL simulation) is used in some contexts, but the slowness of simulation does not allow scaling up. Moreover, this approach does not make it possible to start the development of the software before the code for the synthesizable version of the hardware is written.

The reason for which the simulation of the synthesizable model is so slow is that the level of abstraction used (Register Transfer Level, or RTL) includes too many details of implementation, of protocols, which are not relevant for the software. This led recently to the introduction of a new level of abstraction, called *Transaction Level Modeling* or TLM, in which only what is necessary for the software to run is modeled. At this level, a platform is a set of modules connected by communication channels. The communication channels transport transactions, which are an atomic exchange of data between two modules, whose size can vary, and is in general unrelated to the size of the bus.

The transaction level models must appear early in the design flow of a circuit, and become the reference models for the following developments. The reliability of these models is thus particularly important. As many bugs as possible must be found, as soon as possible. The systems considered are usually not “critical” systems in the sense that the price of each bug is counted in human lives, but are systems for which the price of a bug is counted in millions of dollars.

1.2 General Objectives for the Thesis

The most general objective of the thesis is to develop methods and tools to increase the reliability of the chips by using the transactional models. Since TLM to RTL synthesis is believed to be infeasible automatically and efficiently, the ideal product would be a formal comparison tool for TLM and RTL

models. Such a tool would be extremely complex to develop both theoretically (to compare two programs which use different concepts is a very difficult problem) and practically (one would need extraction tools for both the TLM *and* the RTL levels).

Within the framework of a thesis (three man.year), it seemed much more reasonable to concentrate on the transaction level, and to keep the questions of comparisons with lower abstraction levels for possible future works. The objective of this thesis will thus be to check properties on the transaction level model, independently of the levels of abstraction into which these models will be refined. Since TLM models become the reference models for future development and validation, ensuring properties on TLM models has consequences on the reliability of the final product.

Our idea is to start implementing a set of reusable building blocks, common denominators of any tool handling the transactional models. A first application of this tool chain will be the formal verification of transactional models by existing model-checking and abstract interpretation tools, as described in the final chapter of this document. Other applications could be added thereafter.

1.3 Work Context and Motivations

This document presents the work carried out on the verification of transactional models, as part of a CIFRE Ph.D (Industrial Conventions of Training by REsearch) between the “System Platform Group” (SPG) team of STMicroelectronics and the “Synchronous Languages and Reactive Systems” team of the Verimag laboratory, between October 2002 and December 2005.

It was the first thesis in co-operation between these two teams, and the first year of co-operation between STMicroelectronics and Verimag. The interest was thus not only the scientific contents of the thesis, but also the creation of links between these two teams, which became one of the basis for a more durable co-operation between STMicroelectronics and Verimag. Indeed, two other Ph.D students (Claude Helmstetter and Jérôme Cornet) joined the project and are working on related subjects. A common laboratory between Verimag and STMicroelectronics is being created, as well as a Minalogic (a competitiveness center dedicated to software for micro and nanotechnologies) project called OpenTLM.

The first stage was to understand the principles of the transaction level from a theoretical point of view. From the point of view of STMicroelectronics, the development infrastructure for transactional models is mainly guided by the immediate needs of the users. From a “research” point of view, it is important to make sure that these developments are done on solid theoretical bases. The questionings raised by works on the tool LUSSY, for example, helped us to understand the concepts of asynchronous execution and time in SystemC, and contributed to the development of coding directives for the models with a notion of time: the distinction between the levels “Programmer View” and “Programmer View plus Timing” presented in the next chapter (section 2.2.2.2).

1.3.1 Motivations in the Industrial Context of STMicroelectronics

For the SPG team of STMicroelectronics, the general objective is to create a complete development environment for models at the transaction level, based on SystemC. Before the beginning of the thesis, SystemC components to model abstract communication channels had already been developed, as well as a prototype of configuration management and build system to simplify the task of the users. Meanwhile, these tools evolved. Bridges between various languages and protocols, a finer definition of the various levels of abstraction inside the transaction level, the integration of tools and simulators commercial, etc . . . were added.

A number of these improvements are short-term evolutions developed as a response to user requests. In parallel, a longer-term approach, in co-operation with public laboratories, aims at improving methodologies, and anticipating future problems. The co-operation with the Verimag laboratory is a way to have formal methods applied to concrete problems. One of the motivations for the strong presence of STMicroelectronics in the region of Grenoble is to benefit from its rich scientific environment.

1.3.2 Motivations in the Academic Context of Verimag

The Verimag laboratory has both a number of tools and a very good qualification level regarding program verification. The laboratory, and the synchronous team in particular, tackles at the same time rather fundamental research topics and industrial research ones.

A necessary condition to make good quality applied research is the availability of real case studies. One of the motivations to study the transaction level modeling could be that a publication seems to have more chance to be accepted if it contains key words like "TLM" or "System-on-a-Chip" for example, than if it deals only with automata, or Moore or Mealy machines. We do not intend to take a theoretical problem and try to convert it into an applied research topic by only adding some appreciated keywords that the industry likes. Our starting point is actually a concrete problem to solve, and the way to do it is to look for techniques applicable to it. The fundamental research topics make it possible to solve problems from industrial research, and the case studies give the direction to be followed for the more theoretical questions.

For Verimag, a co-operation with STMicroelectronics is a way to understand the design flow of Systems-on-a-Chip, from the inside.

1.4 Approach and Technical Choices

One of the characteristics of the problem which we are dealing with here is that we control neither the language (SystemC is defined by a consortium on which we have only little influence), nor the execution model (which is part of SystemC).

A solution could have been to study the execution model of SystemC, to redefine it in terms of automata or another well known formalism, and to work directly and only on this formalism. This solution is not very satisfying since it does not make it possible to test the approach directly on real case studies in an automatic way.

A significant part of this work is thus the design and implementation of extraction tools. It is a mandatory first step for any formal work exploiting the characteristics of a new language. This extraction is done in two phases: A "syntactic" extraction, which is a little particular in the case of SystemC, and a "semantic" extraction. The one first is implemented in the tool PINAPA. It is already re-used in projects of STMicroelectronics and other external research projects. The second one, implemented in the component BISE of the tool LUSSY, could also be re-used in future work. The task of BISE can be summarized with a transformation of the output of PINAPA, which is a strongly decorated syntactic tree representing the SystemC program, into a simple and well defined structure of automata.

A series of transformations is then applied. Some are generic, others are specific to the models we analyze. Some are exact transformations, others are abstractions. This is implemented in the component BIRTH.

The last stage is to generate code for the external tools on which we rely for the proofs themselves. We can today use all the tool chain of the LUSTRE [BCH+85] language, as well as the tool SMV [McM01, McM93].

The tool LUSSY was entirely designed and implemented during this thesis. Some parts were carried out with the assistance of students in second year of engineering school (Cédric Bonnot and Remi Emonet), and a master thesis student (Muzammil Muhammad Shahbaz).

Our approach is similar to those of traditional compilers, with a front-end part, intermediate transformations, and a code generator. This approach allows us to have extremely few theoretical limitations, at the syntactical as well as at the semantic level (in practice, some constructs were not useful for us and were not implemented, but they could be easily so if necessary).

1.5 Summary of Contributions and Limitations of the Approach

The main contributions of this thesis are:

A method to extract information from a SystemC model and an implementation of this method called PINAPA. We will see how SystemC is different from traditional languages, and how a SystemC

front-end should be different from a usual compiler front-end. The approach followed for PINAPA has no published equivalent to the best of our knowledge.

An executable formal semantics for TLM models written in full SystemC, with an operational translation tool; The output of this tool is reusable for other purposes.

A way of expressing safety properties in SystemC without the help of a new language;

An intermediate formalism, HPIOM, with a set of transformations and optimizations. This intermediate representation can, and will probably be used outside LUSSY, as an intermediate format between some of Verimag tools.

A working connection to several verification tools. This allowed us to prove some properties on small platforms, and provided an interesting benchmark comparing the verification tools.

The tool LUSSY is still in prototyping phase. Some C++ or SystemC constructs have not yet been taken into account, by lack of time, but could be added without real theoretical problem. As opposed to this, the approach followed presents some intrinsic and theoretical limitations that would be much harder to work around:

Extraction of information from SystemC is a difficult problem, not only to solve, but also do define in the general case. We will see how, in the case of a program using dynamic data-structure to access SystemC objects, it doesn't really make sense to try to find the SystemC object represented by a given variable. PINAPA uses an approach that avoids most of the limitations present in similar tools, but can not make the link between variables of the program and SystemC objects in this case.

The HPIOM formalism is limited in terms of expressiveness. We had to take some design decisions and find the trade-off between expressiveness and possibilities regarding formal and automatic proofs for this formalism. We have chosen a simple automata formalism. The control structure is fixed, dynamic process creation is not allowed, and the variables of the automata can not contain complex data-structures. If the source program uses dynamic data-structures, recursive function calls, etc. then the translation will have to lose some information. This loss of information is done in a conservative way for safety properties, except in the case of function calls with side effects.

Translation from SystemC into HPIOM also introduces, optionally, a number of approximations preserving safety. They improve the performance of the proof engine, but prevent some properties from being proved.

State explosion in the proof engine is actually the main limitation regarding the use of LUSSY for formal verification. On medium size platforms like EASY presented in section 2.5, LUSSY is able to perform the complete translation and can generate either LUSTRE or SMV, but in both case, no property could be proved. However, LUSSY provides all the building blocks to apply other proof techniques, such as modular model-checking, that could allow scaling up better.

1.6 Outline of the Document

This document is divided into three parts:

I. An approach for the Verification of Systems-on-a-Chip: This first part gives a general vision of the concept of transaction level models, and possible formal approaches for their verification. Chapter 2, *"Modeling Systems-on-a-Chip at the Transaction Level in SystemC"* introduces the transaction level, and the SystemC implementation we're interested in thereafter. Chapter 3, *"Overview of* LUSSY*: A Toolbox for the Analysis of Systems-on-a-Chip at the Transaction Level"* presents our verification approach as well as the alternatives. One also finds there a short presentation of the tool LUSSY, his components and the way in which they are organized.

II. Automatic Extraction of Formal Semantics of SystemC Models: In this second part, the extraction we used are presented. Chapter 4, *"PINAPA: Extracting Architecture and Behavior Information From SystemC Models"* presents the first component, whose role is similar to the one of a compiler front-end, for SystemC programs. Chapter 5, *"HPIOM: Heterogeneous Parallel Input/Output Machines"*, presents the intermediate structure we use to represent the semantics of SystemC. The translation itself is detailed in chapter 6, *"BISE: Semantics of SystemC and TLM Constructs in Terms of Automata"*, which transforms the output of PINAPA into HPIOM.

III. Using HPIOM for Formal Verification: This part presents the use of the data extracted from the formal checking. Initially, a series of transformations is applied to the generated automata, as presented in chapter 7, *"BIRTH: Back-end Independent Reduction and Transformations of HPIOM"*. Finally, chapter 8, *"Back-Ends: Connecting HPIOM to Verification Tools"* introduces the generation of LUSTRE and SMV code, which makes it possible to use model-checkers, abstract interpreters and SAT solvers to prove properties on the models. A comparison between these various tools is provided.

Conclusion: The last part is the conclusion, and presents the perspectives.

The attentive reader will notice that no chapter is dedicated to bibliography. Indeed, the topics treated in the various chapters are different, and we preferred distributing the bibliographical references and related works in each chapter. Similarly, the implementation of the tool LUSSY is described all along the document, and not in a dedicated part.

Chapter 2

Modeling Systems-on-a-Chip at the Transaction Level in SystemC

Contents

2.1 Introduction **17**

2.2 The Systems-on-a-Chip Design Flow **18**

2.2.1 Hardware – Software Partitioning 18

2.2.2 Different Levels of Abstraction 19

2.2.3 Verification, Validation, Test 21

2.3 The Transaction Level Model **22**

2.3.1 Example of a TLM platform 22

2.3.2 TLM Concepts and Terminology 22

2.3.3 Importance of TLM in the Design Flow 23

2.4 SystemC and the TLM API **24**

2.4.1 Need for a new “language” 24

2.4.2 The SystemC Library 25

2.4.3 TLM in SystemC 30

2.5 A Larger Example: The EASY Platform **33**

2.5.1 Description of the Platform 33

2.5.2 A TLM model for EASY 34

2.1 Introduction

Quality and productivity constraints in the development tools for the design of embedded systems are increasing quickly. The physical capacity of chips can usually grow fast enough to satisfy those needs: The evolution of the number of transistors on a chip has been following Moore’s law (growth of 50% per year) for decades, and should keep on following it for at least 10 years. But one of the design flow bottlenecks is the design productivity: with traditional techniques, it grows only by around 30% per year, leaving a gap, increasing by 20% per year between the capacity of the chips, and the amount of code the designers are able to write. This problem is often referred to as the *design gap*. New techniques have to be settled continuously to be able to fill in this gap.

This chapter will first present today’s design flow, with the different levels of abstraction used to describe a chip (section 2.2). Then, we will detail the *Transaction Level Modeling* level of abstraction that will be studied in this document (section 2.3), and present the way it is implemented in SystemC (section 2.4.3). The chapter ends with a small case study: The EASY platform, in section 2.5.

2.2 The Systems-on-a-Chip Design Flow

One of the past technological revolution in the hardware domain has been the introduction of the *Register Transfer Level* (RTL) to replace the *gate-level* as an entry point for the design flow. The gate-level description uses only the simple logic operators (*and*, *or*, *not*, ...) to describe the design, whereas the RTL level allows the notion of register (one-word memory), and a data-flow description of the transfers between registers at each clock cycle. Since the translation between RTL and gate-level descriptions can be done automatically and efficiently, the gate-level description is today mainly an intermediate representation synthesized from the RTL code, used for the chip manufacturing, but seldom written by hand (we can compare the role of the gate-level description in the chip design flow with the role of the assembly language in the compilation of software).

2.2.1 Hardware – Software Partitioning

Raising the abstraction level above the gate-level has been a real progress, but is not sufficient to fill in today's design gap. It is necessary to maximize the reusability of the chip components, usually called *Intellectual Properties* (IPs). This can be achieved by using software components instead of *Application Specific Integrated Circuits* (ASIC).

Software components can be easily reused, modified at all steps of the design flow (a bug can be fixed or a feature can be added even after the device has been sold to the final customer). On the other hand, they are much less efficient both in terms of computation time and in terms of power consumption.

Therefore, designers need to find the compromise between software and hardware: implement the performance-critical operations using dedicated hardware components, and the non-critical parts using software. Deciding which feature will be implemented in software and which one will be implemented in hardware is called *hardware/software partitioning*. The result is a mixture of software and hardware, intermediate between general purpose CPU and ASIC and containing several components executing in parallel. It is called a *System-on-a-Chip*.

Since one of the main tasks of the embedded software is to program the hardware components, the software, or at least its low-level layers, will be highly hardware-dependent, and will not run unmodified on a standard computer. There are mainly two ways to execute such software: 1) execute it on the physical chip, and 2) execute it on a simulator.

The first option is not acceptable during the development because of time-to-market constraints: the software needs to be finished and validated a few weeks after the chip comes out of the factory. Furthermore, developing embedded software can help in finding bugs in the hardware, and the cost of a bug discovered on the physical chip is several orders of magnitude higher than a bug found before manufacturing the first chip: the first step of the manufacturing is to build the mask that will be used for the lithography of all chips. Any non trivial bug fix needs a complete rebuild of the mask, which costs around one million dollars (the cost of masks tends to grow exponentially with time, so we can't expect this problem to be solved by itself in the next few years).

The "trivial" way to simulate the hardware part of the chip is to use the RTL description. Unfortunately, due to the highly parallel nature of hardware, simulation of a large RTL design is extremely slow. It is possible to replace some components of an RTL simulation by a simulator, typically written in C, to increase the simulation speed. A common technique is to replace the processor by an instruction set simulator (ISS), and the memory by a simple array. This technique, mixing behavioral and RTL components is called cosimulation.

The next step is to get rid of RTL components and clocks completely. This raises the level of abstraction above the RTL. An emerging level of abstraction is the *Transaction Level Model (TLM)*, and will be detailed later.

Hardware devices dedicated to RTL simulation also exist (hardware accelerators, or hardware emulators based on programmable hardware like FPGA). See for example [GNJ+96]. They are extremely costly and offer very few or no debugging capabilities. They are usually used to run a large test-bench in batch mode, but not for interactive debugging.

Figure 2.1 shows the simulation time we get on the same computation with those different techniques.

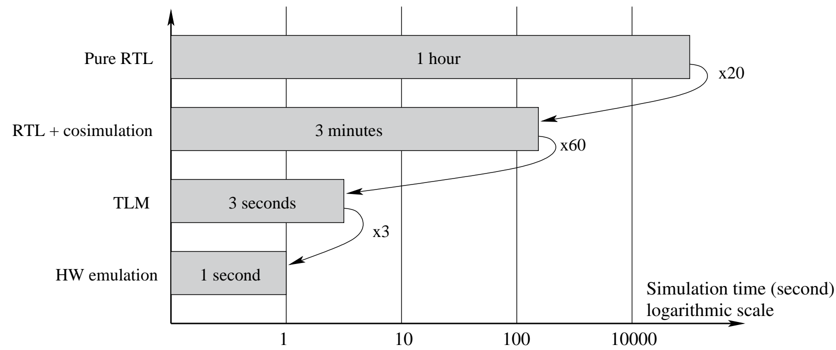


Figure 2.1: Simulation time for the encoding + decoding of one image in a MPEG4 codec

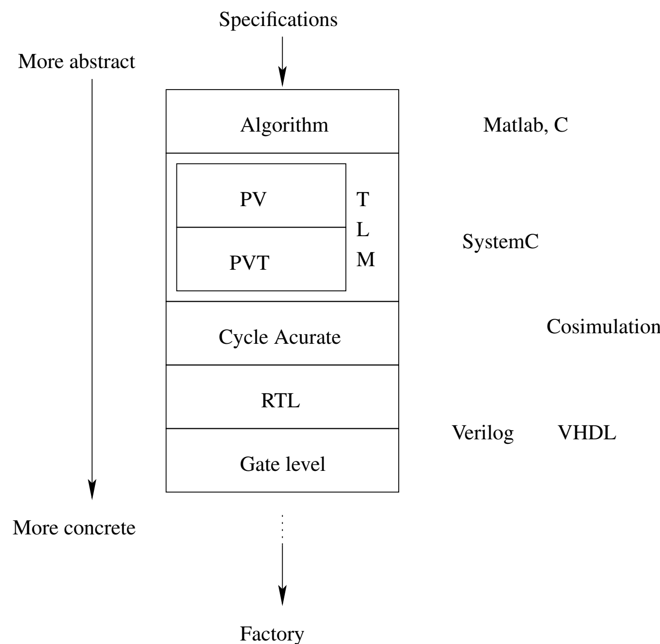


Figure 2.2: Illustration of the Different Levels of Abstractions

2.2.2 Different Levels of Abstraction

This section details one by one the different levels of abstraction used to describe hardware. Ideally, the design flow should start by the highest level, and refine, either automatically or manually to the lowest level. Figure 2.2 illustrate the relationship between the different levels of abstraction. On the left are the levels of abstractions, and on the right are examples of commonly used technologies at this level of abstraction. On this picture, the ideal design flow starts from the top and refines to the bottom.

The distinction between those levels of abstraction is widely (but not quite universally) accepted. Of course, some intermediate levels can be added. See for example [Pas02] (written by an STMicroelectronics employee) or [CG03]. The way to implement them is still subject to discussion, although a standardization effort is ongoing.

2.2.2.1 Algorithm

The highest level (below the specification) is the algorithm. For multimedia devices, at least part of the algorithm is provided by a norm, and reference implementations often exist. Algorithms are usually designed in high level programming languages, such as Matlab™, or in C. At this level, the notion of software or

hardware components does not exist, and the notion of parallelism is not yet taken into account.

2.2.2.2 Programmer View: PV

Then comes the Transaction Level Model, which actually splits into two flavors: the *Programmer View* (PV), and the *Programmer View plus Timing* (PVT).

At this level, the chip is modeled as a *platform* made of several *modules*. Each module shows to the external world all its functionalities, and only its functionalities. The timing aspects, for example, are not yet taken into account.

Communication between modules is done through a model of interconnect (the *interconnect* itself is the set of channels existing on the chip), made of one or several *communication channels*, whose role is to route a piece of data from a module to another one. This exchange of data is called a *transaction*.

At this level, the size of a transaction is not necessarily related to the data-width of the bus (which is not necessarily known when the PV platform is written). For an image processing algorithm, for example, the PV model can decide to transmit the data line by line, block by block, or even image by image.

An important part of the PV level is the system synchronization. At this level of abstraction, we have no real notion of timing, so the system is mostly asynchronous. A set of independant processes could hardly give a consistant behavior. For example, when a process needs a piece of data that is the output of another process, we have to ensure that the value will be produced before being used. The communication channels are not only usefull to transmit data from one module to another, but can also be used for the system synchronization. Some channels can also be dedicated to synchronization and transmit no data. This is the case for the model of an interrupt signal. It can be modeled as a boolean signal (we will observe the edges of the value, but not the value itself), or even a channel without value.

At this level of abstraction, the *model* contains all the necessary and only the necessary information to run the embedded software (thus its name).

The organization of the program is completely different from the one of the algorithmic level. The first partitioning is done, the algorithms are parallelized, even if the decision of implementing some blocks in hardware or software is not necessarily taken (since hardware/software partitioning is done based on the results of performance analysis, and since the PV level does not take performance into account, we still miss some information). Some tools can help taking partitioning decisions, and some code can be reused, but an automatic translation from the algorithm level to the PV level can not exist.

2.2.2.3 Programmer View plus Timing: PVT

While TLM was originally created to ease the development of embedded software, it also proved to be useful for preliminary performance analysis. Since the TLM model contains less details than the RTL, it can be written faster (it is usually admitted that a PV model can be written ten times faster than its RTL equivalent), and should be available earlier than the RTL in the life cycle of a chip. It is therefore reasonable to use it to take decisions about the RTL design (identify the bottlenecks, dimension the modules and communication channels).

Unfortunately, the PV level does not contain the timing information necessary to perform such analysis. It is therefore necessary to enrich the PV model with timing information, with some constraints: The functionality of the PV model must not be modified (the functional equivalence between PV and PVT must be correct-by-construction as much as possible), and the effort to write the PVT model based on the PV model must be as low as possible, and in particular, lower than the effort to write a PVT model from scratch.

At this level, the architecture of the model must correspond to the one of the actual platform. Each data-transfer in the RTL model must correspond to a transaction of the same size in the PVT model.

By adding timing information on each data treatment or transfer, we get a good approximation of the timing of the platform. Ideally, it should be cycle-count accurate (one knows how long a transaction takes), but not cycle-accurate (one doesn't know exactly what happens at a given clock cycle).

2.2.2.4 Cycle-Accurate: CA

The next step increasing the precision of the model is to add the notion of clock. A *cycle-accurate* model describes what happens at each clock cycle. It can still abstract some details, but needs to be aware of the micro-architecture of each component (for example, a model of a processor needs to model the pipeline to be cycle-accurate).

2.2.2.5 Register Transfer Level: RTL

The Register Transfer Level has been discussed earlier. It is the first level of abstraction to be precise enough to be synthesizeable automatically and efficiently. It is bit-accurate, cycle-accurate, and describes all the internal details of each component. Designs at this level of abstraction are usually implemented using the *VHDL* (Very large scale integrated circuits Hardware Description Language) or *Verilog* language.

The difference between RTL and the different flavors of TLM is important. The TLM model does not contain as much information as the RTL version does, and they use different kinds of description language. The RTL level is intrinsically highly parallel and data-flow oriented, while the TLM is usually written in a traditional imperative language, with a lower degree of parallelism. One could imagine an efficient synthesis tool from “slightly above than RTL” to RTL, but an automatic translation from a real TLM model to its RTL version is not possible, or at least not possible in a reasonably efficient way.

2.2.2.6 Gate Level, Back-End

The design flow from the RTL design to the factory is well established (although in constant progress). The *gate-level* netlist is automatically generated from the RTL, then comes placement and routing to transform the netlist into a two-dimensional view, that will be used to draw the lithography mask to be sent to the factory. Synthesis from RTL to gate level is possible efficiently, and is a well established methodology.

2.2.3 Verification, Validation, Test

2.2.3.1 Terminology

Ensuring the reliability of systems is a general problem in all disciplines. The methods may differ, but the terminology also differs from the hardware community to the software community.

For a software engineer, *validation* means verifying that software is reliable. *Test* is one way to achieve validation, by executing the software and observing its potential failures. *Verification* is another way to achieve the same goal, by proving formally that the software is correct.

For a hardware engineer, *validation* also means verifying that system (hardware and software) is reliable. *Verification* is somewhat synonymous, and doesn’t have the “formal proof” connotation that the word has for the software community. *Test* is something completely different, it means verifying that the physical chip has been correctly manufactured. Finding bugs by executing the description of the hardware is called *simulation*.

2.2.3.2 Overview of the Methodologies

Testing a chip is done at the end of manufacturing, for each chip. This is the equivalent of checking that a CD has been pressed correctly for software. It will not find bugs in the design (like checking data integrity on a CD will not ensure the software on it is bug-free). Since testing requires stressing internal parts of the chip, and since the machines used for testing are extremely expensive, a part of the test material is usually integrated in the chip (this is called BIST, for “built-in self test”). This portion of the chip will be used only once in the life of the chip. It is usually not modeled at levels higher than RTL, since it would be both useless and irrelevant.

As explained above, the embedded software is ideally developed on a simulator, and available at the time the chip is obtained from the factory. However, the final validation can, and has to be done on the physical chip. In some cases, it is even possible to update the embedded software after distributing it to the final customer, so, software bugs are problematic, but less so than hardware bugs.

The main priority is to avoid sending an incorrect design to the factory. Doing so would cost at least the price of the mask, i.e. millions of dollars. The focus is therefore on the RTL design. Assuming the synthesis tools are correct, a correct RTL design should lead to a correct chip.

Formal verification is used when applicable (small devices, or well-isolated parts of components, such as bus protocols management), but simulation is today the main way to verify a design. A typical RTL test-bench will run during several hundreds of hours, and may need several days even when parallelized on a large cluster (a common practice is to run regression test-benches every week-end on all the workstations available). It clearly needs to be completely automated.

Test-bench automation means having an automatic execution environment, and distribution on a cluster of computers, but it also means the diagnosis must be automated. Manual examination of execution traces would take millenniums and can not be generalized. Usually, verification by simulation will be done by comparing the system under test with a *reference implementation*.

2.2.3.3 Reference Model

Comparison can be done by comparing the state of the memory of the system at the end of the execution of the test-case (or at some well-defined milestones). If the system contains a processor, then it is possible to run a piece of embedded software on it. Otherwise, it is possible to build a verification environment containing a processor, a RAM, and the system under test. This raise a number of technical problems, to be able to examine the state of the memory, to allow a VHDL system under test to run in the same environment as a SystemC reference implementation for example. It also requires having relevant test-cases. The last problem is to have a reference implementation.

For simple designs, the reference implementation could be written in C, but with the growing complexity of Systems-on-a-Chip, writing the reference implementation represents a big amount of work. It requires taking parallelism into account, having a notion of reusable components, and fast simulation speed. Actually, those requirements are almost the same as the one of the PV level. A good reference implementation is the PV model of the platform.

The TLM levels are therefore not only useful for software development and architecture analysis, but also for design verification. Since the TLM models often become the reference implementation, their reliability is very important.

2.3 The Transaction Level Model

2.3.1 Example of a TLM platform

Just to get the flavor of what a TLM model is, observe Figure 2.3. It shows a subset of the typical architecture of a system-on-a-chip, namely the ARM's PrimeXsys wireless platform. The architecture is made of components and several kinds of connections. The main initiator module is a processor. High level communication is done through the data and instruction buses, and interrupts are managed through synchronous signals. A *transaction* from A to B is the encapsulation of a potentially complex data structure, being transmitted according to a complex protocol that may involve several low-level information transfers in both directions (the data sent, the acknowledgement, etc.). This kind of design is mainly *asynchronous*, in the sense that the designer cannot rely on a global time scale: a component is developed without knowing whether the others components will have the same clock, and that is why synchronization between them involves general protocols.

2.3.2 TLM Concepts and Terminology

[Ghe05] gives a very complete description of the TLM abstraction level, its usage and the way it is implemented on top of SystemC in STMicroelectronics. We will present here the most important concepts.

A component of a TLM platform is called a *module*. A module usually corresponds to a hardware component in the corresponding RTL platform. It defines a state and a behavior, modeled by a set of concurrent processes. A process is typically implemented in a general purpose programming language

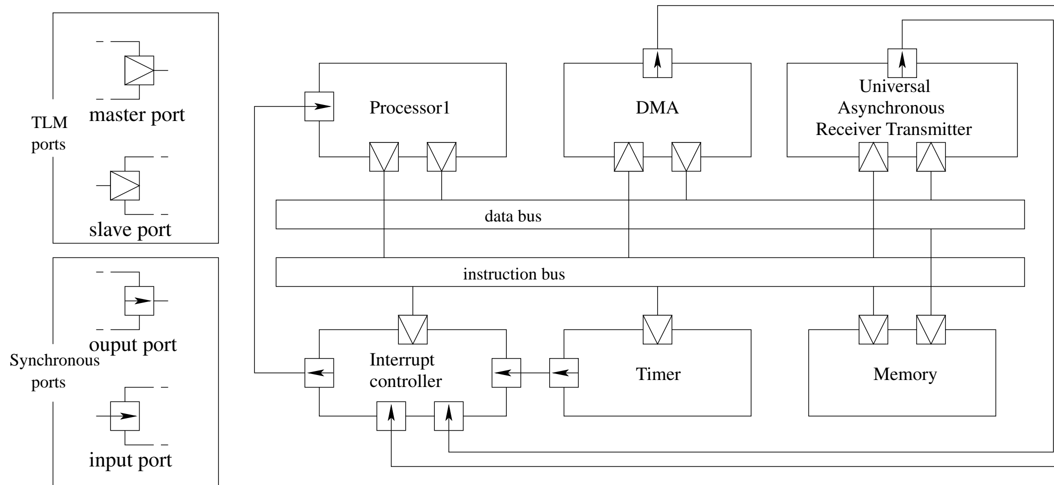


Figure 2.3: An Example TLM design

such as C or C++. Note that a full platform can itself be considered as a module of a larger one. The interactions between the modules are the system synchronization and data transmission.

Although the implementation of the chip may be synchronous, the communication between the different modules can not rely on precise timing information. For example, if a component programs the DMA to write to a given memory zone, it would not be reasonable for another module to read to this memory zone by just waiting "a sufficient amount of time" and then to read the data. A system must clearly characterize the causal relation between its different processes in order to ensure proper system behavior. In our example, the DMA should raise an interrupt and a module requiring the written data should wait for this interrupt.

The execution model of TLM is not so different from the model of distributed systems: in both cases, the components have no shared variable, and can not rely on a global clock or time scale. Both the communication and the system synchronization have to be made explicit.

The modules communicate with the rest of the platform through *ports*. They are connected together using a model of *interconnect*. This can be a simple bus model, but some recent chip designs also use much more complex routing mechanisms, replacing buses by so-called *Networks-on-a-Chip* (NoC). At the transaction level, we put no restriction on the type of interconnect that will be used in the implementation.

In TLM, data-transmission between modules is done through transactions. A transaction is an atomic exchange of data between two modules through a communication channel.

The port from which the transaction is triggered is called the *initiator port*, and the other one the *target port* (the literature also uses the terms "master" and "slave" to refer to the same concepts). An *initiator module* (resp. *target module*) is a module containing an initiator port (resp. target port).

The data is not necessarily transmitted from the initiator to the target: in the case of a read in a memory, for example, the transaction is initiated by an initiator module (typically, a CPU or a DMA), routed to the memory, which acts as a target module to serve the transaction and return the data to the initiator. This can be compared to network protocols where the client initiates a request, which is processed by the server. The protocol can implement a mode for transactions to specify how the transaction should be processed by the target. It is usually `READ`, `WRITE`, or another protocol-specific constant.

In the case of a model of a bus, the address will simply be a n-bits integer value (where n is the width of the bus), and the data can be any multiple of the bus width, plus some meta-data like the *byte-enable* (which byte is relevant in a word or set of words – a *byte* being the smallest data manageable by the system, and a *word* a piece of data of the width of the bus).

2.3.3 Importance of TLM in the Design Flow

The primary goal of TLM was to allow early embedded software development, and to tighten hardware development and software development in the SoC design flow. However, the same level of abstraction can

be used for several different tasks.

We have seen in section 2.2.2.3 that with a reasonable effort, a purely functional TLM model (PV) can be enriched to provide a good approximation of the timed behavior, with a fast simulation speed (PVT). The decisions taken using this model are crucial for the rest of the design flow. A wrong estimation of the performance can lead to a complete reorganization of the chip.

As TLM models appear early in the design flow, they also become *de facto* reference models for the final chip. The verification of the RTL design will be done mainly by comparing the result of the execution of TLM model with the execution of its RTL equivalent.

The correctness of the TLM models are therefore critical. The later a bug is found in the design flow, the more costly it is.

2.4 SystemC and the TLM API

2.4.1 Need for a new "language"

We have defined the TLM level of abstraction. To be applicable in practice, we need a way to write and execute such models.

The main technical requirements are the following:

Efficient simulation: The main reason to write TLM models instead of RTL models is to gain several orders of magnitude in terms of simulation speed.

Modular design and code reuse: Classical software paradigms such as genericity, support for abstract data-type or object oriented programming can be helpful at this level.

Parallel execution semantics: The components must execute their behavior in parallel, synchronize themselves, and communicate together.

Additionally, one can require interfacing with standard Hardware Description Language (*HDL*) languages (VHDL, Verilog), easy debugging, and powerful tools for visualizing execution traces.

It is more than desirable to build the TLM technologies on open standards for several reasons. First the models written in this language will become crucial in the life cycle of the chip, so, a dependency to a CAD (Computer Aided Design) vendor would mean a very strong dependency. Furthermore, the code reuse objective can only be achieved if the modules to be reused are compatible. This means the technology has to be supported by all the IP providers, which is unlikely to happen for a proprietary technology.

A number of other approaches have been proposed for the description of heterogeneous hardware/software systems with an emphasis on formal analysis. See, for instance, Metropolis [BLP+02]. In this type of approach, the definition of the description language is part of the game. The language can be defined formally, and tailored to allow easy connections to validation tools.

The need for efficient simulation and the preference for well-known languages lead to consider a C-based, or C++-based approach. [Arn99], although oriented towards the promotion of a commercial tool, gives a good overview of the motivations behind this approach. A similar approach is to create a new language, inspired from C and C++, with the additional required features. This is the approach followed by SpecC [FN01]. The approach didn't get a lot of success in the industry, partly because it requires a specific compiler.

SystemC 2.0 has been designed to meet the above requirements. It is open source and relies on an ISO standard language: C++. This is crucial for two reasons: first, it guarantees a fast learning-curve for the engineers of the domain; second, it guarantees that the models of systems developed in SystemC can be exploited even if the tool that was used to build them is no longer available. SystemC itself is defined by the *Open SystemC Consortium Initiative* (OSCI), involving the major actors of the domain. SystemC is now an IEEE standard (IEEE 1666). It is actually a set of classes for C++, providing modules, signals to model the low-level communications and synchronizations of the system, and a notion of simulation time. SystemC designs are made of several processes that run *"in parallel"*, according to a scheduling policy that is part of the SystemC library specifications. They are described in full C++ code. Unlike Metropolis, SystemC has not been defined with formal analysis in mind. Originally, it is mainly a simulation and coordination language, aiming at accepting all kinds of hardware or software descriptions in a single simulation.

Developing tools for SystemC is therefore interesting. SoC case-studies written directly in SystemC can be obtained from industry, and used to validate an analysis approach. The use of any academic formal language as the core of an analysis tool would imply that any case study is first *rewritten* into this language, before analysis can be performed. Then the validity of the results obtained, with respect to the original system, can be questioned.

2.4.2 The SystemC Library

We give here a brief tutorial of the SystemC library. Complete information can be found in the SystemC Language Reference Manual [Ope03].

A model written in SystemC is executed in two phases:

Elaboration: The entry point for the execution of the model is the function `sc_main()` (the SystemC library itself provides the `main()` function, which is a very simple wrapper around `sc_main()`). This starts the *elaboration phase* during which the modules will be instantiated in the usual C++ way, and connected (*bound) together. Then, the program calls the function* `sc_start` *that will launch the simulation.*

Simulation: *After the call to* `sc_start()`*, the architecture of the model is fixed. No module can be instantiated, and the binding can not change. The* SystemC kernel *will call the member functions of the modules that have been registered as processes. This is called the* simulation phase.

During the simulation, SystemC distinguishes two kinds of processes: `SC_THREAD` and `SC_METHOD` (plus the semi-deprecated `SC_CTHREAD`). An `SC_THREAD` is an explicit infinite loop. It is a function that never terminates, but that hands back the control to the scheduler using the `wait` statement or equivalent. An `SC_METHOD` in a process that executes in zero time, implemented by a C++ function that must terminate, and that will be called repeatedly by the scheduler.

2.4.2.1 Example of a SystemC Model

Figures 2.4 and 2.5 give an example of a SystemC program. They are not meant to be examples of good programming practice, but to illustrate the possibilities of SystemC, and later the features and limitations of our tools.

2.4.2.2 Architecture of a Model

In SystemC, the modules are instances of the class `sc_module`. Each module may contain one or more processes. Communication internal to a module can be done in several ways (shared variables, events, etc.), but inter-module communication should be expressed with SystemC communication primitives: *ports* and *channels*.

A module contains *ports*, which are the interface to the external world. The ports of different modules are bound together with communication channels to enable communication. SystemC provides a set of communication interfaces such as `sc_signal` (synchronous signals), and abstract classes to let the user derive his own communication *channels*.

In the pieces of code of Figures 2.4 and 2.5, we have here two definitions of modules, one of which is instantiated twice. The model is represented graphically in Figure 2.6.

2.4.2.3 Simulation Phase

The elaboration phase ends with a call to the function `sc_start()` that hands the control back to the SystemC kernel (line 70 in our example). The last part of the execution is the simulation of the program's behavior. The SystemC kernel executes the processes one by one, according to the algorithm presented in Figure 2.7.

Initially, all *processes* are eligible. Processes are ran one by one, non-preemptively, and explicitly suspend themselves when reaching a waiting instruction. There are two kinds of waiting instructions: a

```
1   #include "systemc.h"
2   #include <iostream>
3   #include <vector>
4
5   struct module1 : public sc_module {
6     sc_out<bool> port;
7     bool m_val;
8     void code1 () {
9       if (m_val) {
10        port.write(true);
11      }
12    }
13    SC_HAS_PROCESS(module1);
14    module1(sc_module_name name, bool val)
15      : sc_module(name), m_val(val) {
16      // register "void code1()"
17      // as an SC_THREAD
18      SC_THREAD(code1);
19    }
20  };
21
22  struct module2 : public sc_module {
23    sc_in<bool> ports[2];
24    void code2 () {
25      std::cout << "module2.code2"
26                << std::endl;
27      int x = ports[1].read();
28      for(int i = 0; i < 2; i++) {
29        sc_in<bool> & port = ports[i];
30        if (port.read()) {
31          std::cout << "module2.code2: exit"
32                    << std::endl;
33        }
34        wait(); // wait with no argument.
35                // Use static sensitivity list.
36      }
37    }
38    SC_HAS_PROCESS(module2);
39    module2(sc_module_name name)
40      : sc_module(name) {
41      // register "void code2()"
42      // as an SC_METHOD
43      SC_METHOD(code2);
44      dont_initialize();
45      // static sensitivity list for code2
46      sensitive << ports[0];
47      sensitive << ports[1];
48    }
49  };
```

Figure 2.4: Example of SystemC Program: Definition of Modules

```
50  int sc_main(int argc, char ** argv) {
51    bool init1 = true;
52    bool init2 = true;
53    if (argc > 2) {
54      init1 = !strcmp(argv[1], "true");
55      init2 = !strcmp(argv[2], "true");
56    }
57    sc_signal<bool> signal1, signal2;
58    // instantiate modules
59    module1 * instance1_1 =
60      new module1("instance1_1", init1);
61    module1 * instance1_2 =
62      new module1("instance1_2", init2);
63    module2 * instance2 =
64      new module2("instance2");
65    // connect the modules
66    instance1_1->port.bind(signal1);
67    instance1_2->port.bind(signal2);
68    instance2->ports[0].bind(signal1);
69    instance2->ports[1].bind(signal2);
70    sc_start(-1);
71  }
```

Figure 2.5: Example of SystemC Program: Main Function

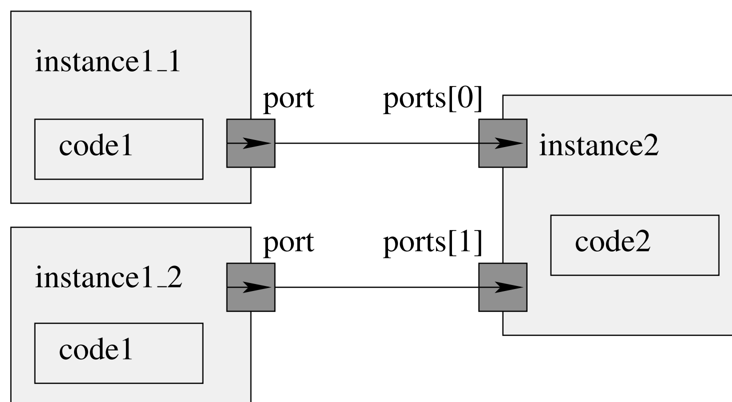


Figure 2.6: Graphical View of the SystemC Program

process may wait for some time to elapse, or for an event to occur. While running, it may send events, or write on signals. These actions are stored in the data structure F, until the *update phase*, at the end of the *evaluation phase*; then they are taken into account one by one: an event awakes the processes that were waiting for it; a signal write is stored in V for the next reads. When there is no more eligible process at the end of an update phase, the scheduler lets time elapse, awaking the processes that have the earliest deadline. Note that this algorithm gives only the simpler cases. Uninitialized processes, instantaneous notification for example would add particular cases.

E: the set of eligible processes

S: a set of tuples (P, e) of sleeping processes
waiting for events

T: a set of tuples (P, t) of processes
associated with time events

F: a set of event or signal consequences:
(EV, e) or (SIG, s, v)

V: a set of tuples (s, v) for signal values

E := { all processes, except those on which `dont_initialize()` has been called.}

loop until the end of simulation

- while E $\neq \emptyset$ one execution of this loop body is a δ-cycle
  - // the so-called *evaluation phase:*
  - while E $\neq \emptyset$
    - P := one element in E ; E := E $-$ { P }
    - run P, while filling F and reading signal values in V, until it stops:
      - if P emits an event e: F := F $\cup$ { (EV, e) }
      - if P writes a value v on a signal s: F := F $-$ { (s, ...) } $\cup$ { (s, v) }
    - if P stopped on a wait-time (t) T := T $\cup$ { (P, t) }
    - if P stopped on a wait-event S := S $\cup$ { (P, e) }
  - end
  - // the so-called *update phase:*
  - For each element f in F
    - if f = (EV, e) then
      - for each (P, e) in S: E := E $\cup$ { P }; S := S $-$ { (P, e) }
    - if f = (SIG, s, v) then V := V $-$ { (s, ...) } $\cup$ { (s, v) }
    - F := F - f
  - end
- end
- min = minimum value of the t_i's in the set T = { (P_i, t_i) } // let time elapse:
- for each element x=(P_k, min) in T
  - T := T - { x } ; E := E $\cup$ { P_k }
- end

end loop

Figure 2.7: The SystemC scheduler algorithm.

This algorithm may appear strange to someone used to software scheduling. This idea actually comes from the RTL Hardware Description Languages that use the notion of δ-cycle to approach the synchronous semantics: for instance, suppose one wants to write a n-bits adder, by composing n one-bit adders, each containing one process. In the physical execution, the carry will propagate until all signals are stabilized. If the carry propagation is longer than the clock cycle, then the timing is incorrect. In a digital simulation, we do not have this notion of physical propagation, but the simulation semantics should be as close as possible to the physical behavior. If we execute the processes only once in a clock cycle, in an arbitrary order, then, the result will be dependent on the order of execution. If the least-significant bits are added first, then carry propagation will occur normally. If most-significant bits are added first, then the carry will be ignored.

One solution is to statically check the causal dependencies and compute an order of execution that will respect the causal dependencies (this is the approach followed by synchronous languages such as LUSTRE [BCH+85] or ESTEREL [Ber00]). The other approach, followed by the standard HDLs and SystemC, is to reschedule the execution of each process until the signals get stabilized. Most actions (`sc_signal` value update for example, the `notify()` function in SystemC being a notable exception) are actually taken into account after all the scheduled process have finished their execution (at the end of the so-called *δ-cycle*). If those actions wake up other processes, then, those processes will be executed during the next δ-cycle.

2.4.3 TLM in SystemC

2.4.3.1 User-defined Channels in SystemC 2.0

In the version 1.0 of SystemC, the library contained only low-level constructs, and could hardly be used to model communication at a higher level than RTL.

Starting from version 2.0, SystemC allows user-defined communication channels. The mechanism is the following:

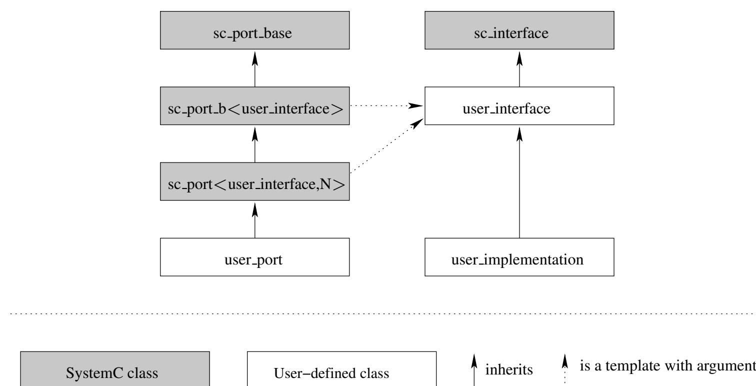


Figure 2.8: Inheritance Diagram for User-Defined Communication Channels

SystemC defines the abstract class `sc_interface` from which the user can derive his own interface, and implementation. The user-defined interface is a contract between the modules and the channel (see figure 2.8).

It also defines the classes `sc_port_base`, `sc_port_b<>` and `sc_port<>`, and lets the user derive his own port from `sc_port`<*user_if*, *N*>, *user_if* being the user-defined interface, and *N* the maximum number of channels that can be bound to this port. The user-defined port exports some functions to the module containing it, which are mainly wrappers around the functions of the interface of the channel.

The SystemC primitive channel `sc_signal` and the corresponding ports are a good example of channel built using this mechanism. It defines the interfaces `sc_signal_in_if<>` and `sc_signal_inout_if<>` defining in particular the methods `read()` for the first one, and in addition `write()` for the second one. Those interfaces are implemented among others in `sc_signal<>`. The

corresponding ports, sc_in and sc_inout define the same methods, which are trivial wrappers around the methods of the interface. For example, the sc_in<data_type>::read() method is defined by

```
const data_type& read() const
    { return (*this)->read(); }
```

(the operator -> has been overloaded to return the interface bound to the port).

2.4.3.2 The TLM interface

The System Platform Group team of STMicroelectronics proposed a common interface for several TLM protocols. The interface itself, called tlm_if contains only one function:

```
virtual tlm_status transport(tlm_transaction& transaction) = 0;
```

The role of the transport method is to route a transaction from an initiator module to a target module. The corresponding module tlm_port is defined with the appropriate binding methods.

At this level, a transaction only contains a pointer to the initiator and target ports, a pointer to the data and one to the meta-data (the data will be used to carry the actual data that will circulate on the bus, the meta-data will typically be used to carry information about the success or failure of the transaction). Both the data and the meta-data are empty classes that will be derived in protocol implementation.

This is of course insufficient to actually write a TLM model. This interface, with the associated guidelines, give only the common ground. Real protocols will be layered on top of it. The user of the protocols will only see the protocols, but not the TLM interface itself.

Later, this proposal has been reviewed, improved, and finally standardized by the Open SystemC Consortium Initiative. One of the changes is that the TLM interface is now a template:

```
template<REQ, RSP>
class tlm_transport_if : public sc_interface {
public:
  virtual RSP transport(const REQ&) = 0;
};
```

In [RSP+05], the standard is presented, and some guidelines are given to build protocols on top of this interface. The protocols presented below correspond to the former proposal, and are being re-written to use the version of the OSCI standard.

2.4.3.3 The BASIC protocol

The BASIC protocol was designed as an example of protocol that can be implemented using the TLM interface. It was implemented by STMicroelectronics and is actually not used for production by anyone. It defines two modes of transactions: READ and WRITE. The transaction data contains the mode, an address and a piece of data. The address is of type unsigned long, and the data is an array of unsigned long (it can contain a block of data instead of a single piece of data).

The transaction is routed according to an address map specified in a separate configuration file. The BASIC protocol is implemented in two flavors: the basic_router is a highly available channel. Transactions are served as they arrive, in parallel in the case of concurrent accesses. The basic_arbiter, on the other hand, serializes the transactions and applies a simple arbitration policy to serve the most prioritized transactions first.

2.4.3.4 The TAC Protocol

The TAC protocol is an extension of the BASIC protocol. Unlike BASIC, it is intended to be used in production. Its features are a superset of the functionalities of BASIC. It should contain all the necessary, and only the necessary to model a SoC at the PV level, and therefore enable software development and

platform verification. It was designed and implemented by the SPG team of STMicroelectronics, and is widely used internally.

The first addition of `TAC` compared to `BASIC` is the fact that `TAC` channels are C++ templates on the addresses and data-type. This is mandatory to allow bit-accurate modeling as well as flexibility.

`TAC` also models some features of some bus protocols that could hardly be built on top of a simple `READ`/`WRITE` interface. It is possible to send `LOCK`/`UNLOCK` commands to the channel itself. The semantics is the following:

- When the channel receives a transaction of mode `LOCK`, it becomes locked;
- When the channel receives a transaction of mode other than `LOCK`, with the option `LOCK_ACCESS` set, the channel processes the transaction normally, and becomes locked afterwards.
- When the channel is locked and receives a transaction, the transaction is queued, even if there are no other transactions pending,
- The channel will be released by the next non-locking transaction.

Another addition of `TAC` compared to `BASIC` is the ability to consider only a subset of the bytes of the transaction. For example, on a 32-bits wide bus, an initiator may want to write a single byte in memory. This is called byte-enable. In `TAC`, this is just an additional information in the transaction that can be set by the initiator, and taken into account by the target.

2.4.3.5 Standardization of a Generic Channel

The TAC protocol relies on open standard, but is not a standard itself. STMicroelectronics whishes it to become a standard, for at least two reasons:

- We need a standard to be able to exchange IP blocks at the TLM level, without the need for protocol adapters.
- We prefer the standard to be our proposal, since we are already using it and supporting it. It would allow us to stay in advance compared to our concurrents.

The standardization process has not begun, yet, but some candidates are already there: the TAC protocol, the Generic User Bus from GreenSoCs[1], and probably others in the future.

2.4.3.6 Didactic Example of Model Using the `TAC` Protocol

The following example (see Figure 2.9) will be used later to illustrate the extraction of the semantics of a SystemC model. It is voluntarily minimalist, and for clarity, we only show the body of the processes, and the methods called to process transactions in the target modules. The system contains two initiator modules and two target modules. They are connected through a `tac_seq` channel. The program contains *assertions* for some properties we will prove (or falsify) later (chapters 6 and 8).

Take the example of the statement `port.write(address, &x)` in the module `status_initiator`. We are going to follow the transaction sent by this statement step by step:

2.4.3.6.1 Build the transaction, transmit it to the channel. The process calls the method `write` on the port of the module. The port will create a transaction of mode `WRITE`, with the address `address` and the data `x`. The transaction is then passed to the member function `transport` of the channel.

2.4.3.6.2 Wait for channel availability. Since the channel is a `tac_seq`, it will not serve the transaction if it is already either locked or serving another transaction. If the channel is busy, the transaction will be queued, and it will be woken up when the previous transaction finishes.

2.4.3.6.3 Resolve addresses. The channel has loaded its addressmap from the file `CHANNEL.map` during the elaboration phase. The file contains:

[1] `http://www.greensocs.com/projects-gub.html`

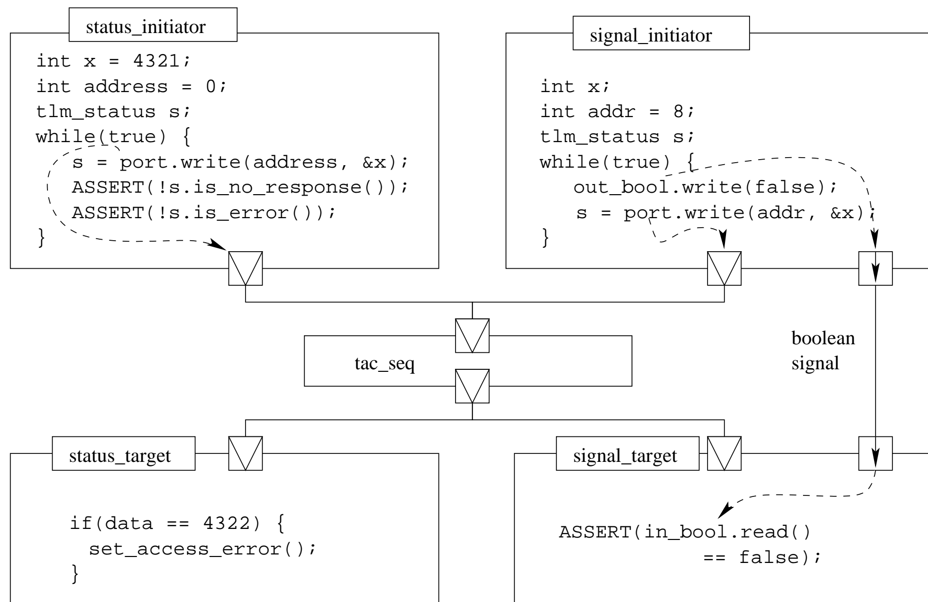


Figure 2.9: An example transactional system

```
; Module name                       start address       size
status_target.target_port           0x0000              0x0008
signal_target.target_port           0x0008              0x0001
```

Since we are writting at the address 0, the transaction will be routed to the module `status_target`. The module `status_target` does not overload its `transport()` method, so, the default one will be used. For a transaction of mode `WRITE`, it will call the member function `WriteAccess()` of the module (the body of this function is the code shown on the picture).

If a module writes at an unmapped address, then the transport method of the channel returns, and the status returned is such that `status.is_no_response()` is true. If a module is mapped at this address, the status returned is such that `status.is_ok()` is true, unless the method `set_access_error()` has been called during the `WriteAccess` call.

2.4.3.6.4 Desynchronize. After the function `transport()` of the target module returns, the channel executes a `wait` statement to allow other processes to execute. This is necessary since the SystemC scheduler is not preemptive, and we want to let other processes execute.

2.5 A Larger Example: The EASY Platform

2.5.1 Description of the Platform

We present here a platform we used to evaluate our approach and tools. The platform, created by ARM, is called EASY, for **E**xample of **A**MBA **Sy**stem. It is intermediate between a real-life large platform and a trivial example: it uses real components and bus protocols. The specification [ARMip] can be downloaded from ARM's website.

The platform is represented graphically in Figure 2.10. The components are connected using two different buses: a high performance bus, the Advanced High-performance Bus (AHB), for the performance-critical part of the chip, and a low-power bus, the Advanced Peripheral Bus (APB) for low bandwidth components. The main ones are:

The core processor is an ARM7 processor, with its AMBA wrapper (ARM7TDMI).

The internal memory which, from the behavior point of view, can be considered as a large array of bytes.

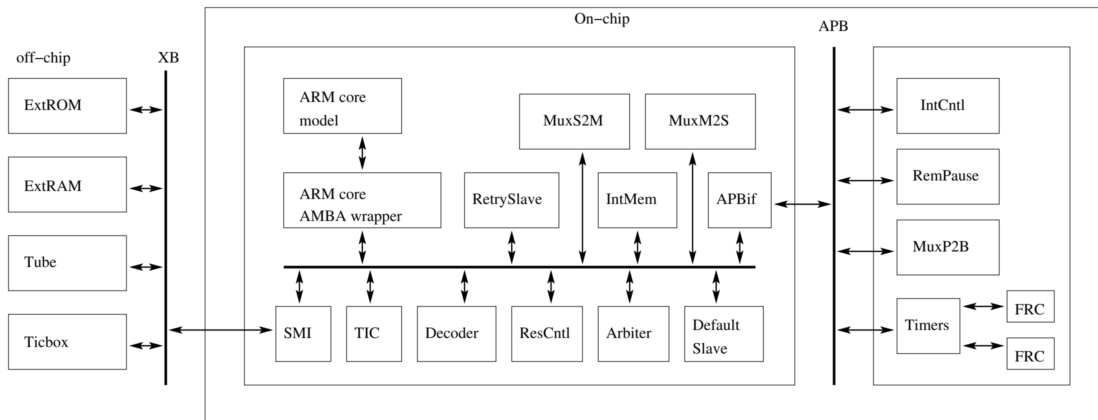


Figure 2.10: Example of AMBA System (EASY) Platform

The interrupt controller This is an APB slave, providing a simple software interface to the interrupt system.

Two timers are gathered in an APB slave. They are Free-Running decrementing Counters: programmed by software, they count and raise an interrupt when their value reaches zero.

The test interface controller (TIC) provides a state machine stimulating the rest of the chip for testing. Be warned that the meaning of "testing" here is the one used in the hardware domain (check the presence of defect in the physical chip), not "simulation". This component is used only once for each chip, at the end of the manufacturing.

The retry slave is a minimalist example of a slave module.

Some components are dedicated to communication within the chip, or with the external world. They are either standalone components or part of the bus.

The static memory interface (SMI) is an interface to the external bus. It is a slave for the internal bus and a master on the external one.

The decoder is actually a part of the bus. Given an address, it generates the signals to route the accesses to the corresponding slaves. It can be programmed to change the mapping of the first segment of the address space, which must be different during the boot phase and during normal behavior.

The reset controller (ResCntl) is used to generate a reset signal from an external input.

The arbiter manages concurrent accesses on the bus.

The default slave is used to respond to transfers that are made to unmapped addresses.

The multiplexor slave to master is used to connect all of the system bus masters to the bus slaves.

The multiplexor master to slave is used to connect the read data and response signals of the bus slaves to the bus masters.

The AHB to APB bridge receives transfers from the AHB and transmit it to the APB.

The remap and pause controller is programmed through the bus (by software), and can enable the remap in the decoder, and a low-power "wait for interrupt" mode.

The Multiplexor Peripheral to Bridge is used to connect the read data outputs of the bus slaves to the bridge module.

2.5.2 A TLM model for EASY

ARM provides the RTL description of the EASY platform, written in VHDL. The SPG Team of STMicroelectronics wrote a TLM model for this platform in SystemC. It currently uses the PV level of abstraction (no timing information), and uses the TAC protocol for the transport mechanism. The specifications can be found in [WS04] (written by STMicroelectronics, but unpublished as of now).

The following components appear in the TLM model of the EASY platform:

One or Several Traffic Generator(s) replace the core processor. Since we would run compiled C code on the processor anyway, compiling it for an ISS or compiling it as part of the platform doesn't change

much.

One TAC Channel that models all the communication in the chip.

One Memory That models both the external and the internal memory.

One Timer Component modeling the two timer sub-components of the actual platform.

One Interrupt Controller similar in behavior and interface to its RTL version.

One Universal Asynchronous Receiver Transmitter (UART) that doesn't exist in ARM's version, but has been added to give an example of I/O component. This makes the TLM EASY a variant and not exactly a model of the reference version.

In addition to the TAC modeling the communication through the buses, interrupt signals are modeled using `sc_signal`. Future versions will include a dedicated TLM protocol for interrupts, which is more efficient than `sc_signal` in terms of simulation speed.

The following components have been omitted from the TLM version, because they have not been considered relevant at this level of abstraction (some of them can be subject to discussion).

The static memory interface: At the TLM level, we don't make any differences between internal and external memory. The SMI is completely transparent from the software point of view.

The decoder, the arbiter, the various multiplexors, the AHB to APB bridge are actually part of the implementation of the bus protocol, and will be modeled by one single TAC channel.

The reset controller is not modeled in the version of EASY we're using. It would have to be modeled to be able to test completely the embedded software, though.

The default slave and the test interface controller have not been considered as being part of the platform's behavior.

The remap and pause controller are not modeled in the version we are using, but more recent versions of the TAC channel include remapping capabilities, built in the channel.

It contains 7 modules, 2 `SC_THREAD` processes, 6 `SC_METHOD` processes. It counts 3,500 lines of C++ code (including comments and blank lines). It is illustrated in Figure 2.11.

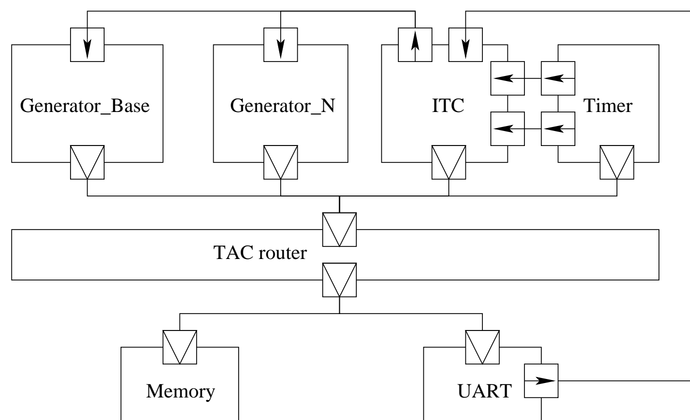


Figure 2.11: TLM model of the EASY platform

Chapter 3

Overview of LusSy: A Toolbox for the Analysis of Systems-on-a-Chip at the Transaction Level

Contents

- **3.1 Introduction** **37**
- **3.2 Techniques and Tools for the Verification of SoCs** **38**
  - 3.2.1 Algorithms for Formal Verification 38
  - 3.2.2 Candidate Tools for the Verification of SoCs 41
- **3.3 Our Verification Approach** **42**
  - 3.3.1 Expressing Properties 42
  - 3.3.2 Synchronization Code Vs. Complex Algorithms 43
  - 3.3.3 The LusSy Tool Chain 43

3.1 Introduction

Section 2.3.3 presented the importance of TLM in the design flow of Systems-on-a-Chip. Verification methods are well established for RTL. Bugs in hardware are known to be extremely costly, and various techniques are applied to find them as efficiently and as soon as possible.

The introduction of a new abstraction level implies the creation of a complete development environment, including a programming language, an editor, some debugging, visualization, and verification tools. The question of verification tools is a key point for the wide adoption of TLM models in the industry, and is being addressed by a joint project between Verimag and the SPG team of STMicroelectronics.

Several problems appear:

- What does it mean to validate properties at the TLM level? Validation can be either simulation or formal verification. One of the problems is that the languages used for TLM modeling, such as SystemC, do not have a formal semantics. Validation of TLM models is essential since they become the reference model for further verification.

- Since automatic synthesis from TLM to RTL does not exist (and will not exist soon), how can we *compare* a TLM reference design and a RTL design that is supposed to implement it, and is partly written by hand?

- How can we express and validate non-functional properties of SoCs at the TLM level? Non-functional properties include timing properties, power consumption, and silicon surface.

In this document, we report on the work done for addressing the first item: how to give a formal semantics to SystemC and the additional TLM constructs, and then express and verify properties of a TLM design written in SystemC. We will insist on properties related to the synchronization of the system, but will not try to prove properties depending on complex algorithms implemented by the platform. We describe both the principles and the implementation, LUSSY.

This chapter is an overview of the approach we followed to solve this issue. We start by a general introduction about the algorithms used for formal verification in general in section 3.2.1, and in the particular case of Systems-on-a-Chip in section 3.2.2. Then, we introduce our own approach in section 3.3. The tool chain is presented component by component in section 3.3.3.

3.2 Techniques and Tools for the Verification of SoCs

3.2.1 Algorithms for Formal Verification

When applicable, formal verification is the ideal case of verification method. It gives either a formal proof that the program is correct or some hints about the way to falsify the property to prove (usually, a counter-example, or a superset of possible counter-examples).

Formal verification can be classified into several categories. We present briefly here four of them: Theorem proving, model-checking, abstract interpretation and SAT solving.

3.2.1.1 Theorem proving

It is well known that the problem of equivalence between two programs or between a program and its specification is not decidable in the general case. Two solutions can be considered: restrict the languages to decidable cases, or accept the fact that the problem to solve is not decidable, and design an interactive tool.

The latter is the approach followed by theorem proving tools like COQ [BC04], ISABELLE [NPW02], PVS [ORS92], and ATELIER B [AN96]. The last one has successfully been applied on real-life case studies such as the "Météor" subway in Paris.

The main idea behind the B method is the refinement: Designers start by writing a formal specification and refine it (in one or several steps) to an implementation. The original ATELIER B uses subsets of C and Ada as target languages for implementation. Research are carried out to apply the B method to the development of Systems-on-a-Chip: Instead of refining towards a software-oriented imperative language, the refinement flow splits between software and hardware, that will use VHDL as a target language. This is implemented in the tool BHDL [Phi03].

The drawback of the theorem proving approach is that the tool being interactive, it requires an expert user, and a lot of manpower to prove a reasonable amount of code. This is probably the reason why only very few industrial successes have been achieved with this class of methods.

3.2.1.2 Model-Checking

Another approach is to restrict the model to study to a finite set. This is the case of any bounded-memory program, and in particular, the case of finite automata. In this case, many state-space exploration techniques can be used, and will either prove a property or give a counter-example in finite (but often exponentially long) time. This set of techniques is called *model-checking*.

The kind of property that can be proved on a program can be classified in two main categories: safety and liveness [Lam77] (a theorem says that any property P can be written as $P \Leftrightarrow P_{liveness} \wedge P_{safety}$ where $P_{liveness}$ is a liveness property and P_{safety} a safety property). A *safety property* is a property that can be falsified by a finite trace ("something wrong won't happen"). A *liveness property* is a property that can not be falsified by a finite trace ("something good will eventually happen"). Liveness properties are both less useful in practice and harder to prove than safety properties. It is important also to note that the notion of conservative approximation is not the same for safety and for liveness properties: a conservative abstraction for safety is an abstraction that only adds behaviors, while a conservative abstraction for liveness can only

remove behaviors. Proving safety and liveness properties are therefore two different tasks, and we can hardly do both efficiently at the same time. We will therefore focus on safety properties, which can always be expressed as a state or transition reachability in an automaton.

The system on which we are trying to prove a property is defined by a set of states, and a transition function. We note $\text{succ}(x)$ the image of x under the transition function. We extend this notation to finite sets: $\text{succ}(\{x_1, x_2, \ldots, x_n\}) = \{\text{succ}(x_1), \text{succ}(x_2), \ldots, \text{succ}(x_n)\}$.

The "simplest" solution to explore the state-space is the enumerative approach. The idea of the algorithm is as follows (we compute the set E of reachable states, also called fixed point of the system being proved). We're looking for the smallest set E such that $E = E \cup \text{succ}(E)$, that we can compute iteratively:

E: The set of states to explore

S: The set of states already explored

E := $\{$ initial states $\}$

while E $\neq \emptyset$

  choose $x \in$ E

  S $=$ S $\cup \{x\}$

  E $=$ E - $\{x\} \cup \left(\text{succ}(x) \cap \overline{\text{S}}\right)$

end loop

The biggest revolution in the domain of model checking has probably been the introduction of symbolic model checking [DCB+90]. The idea is to find a way to represent a Boolean function in a compact and canonical form. Sets of states can then be encoded by the characteristic function of the set. We also need an efficient way to compute the image of a set of states by a function. *Binary Decision Diagrams* (BDD) [Bry92] satisfy all those requirements for states represented by valuations of Boolean variables. The resulting algorithm is as follows:

E: A formula describing the set of states reachable for this step

NE: A formula describing the set of states reachable for next step

E := $\{$ initial states $\}$

NE := $\emptyset$

while E $\neq$ NE

  NE $=$ succ(E)

  E $=$ E $\cup$ NE

end loop

In this case, the complexity of the algorithm does not directly depend on the number of states anymore. It is still exponential in the worst-case, but can give better results than enumerative methods in practice.

Those methods apply well to small and medium Boolean programs. In particular, it is used successfully to verify RTL designs, since an RTL design is actually an encoding of an automaton, in which any safety property is decidable. Tools widely used in the industry embedding a symbolic model-checker include RuleBase™ [BBDEL96] by IBM, Synopsys Magellan™ [HSH+] and FormalCheck™ [Cad99] by Cadence. In practice, state explosion does not necessarily allow any property to be proved, but considering the design part by part allows to prove local properties very efficiently.

Exact model-checking quickly becomes hopeless for real-life models. In particular, for a program containing integers, modeling an n-bits integer with n Boolean variables leads to a state explosion and irregular BDDs, whose size can be an exponential of the number of variable (this is the behavior of SMV). If we consider integers to be infinite, then the problem is even undecidable. To be applicable on real

programs, a model-checker needs therefore to perform some abstractions on the program. Abstractions must be conservative for safety properties. This means that the abstract program's set of reachable states must be a superset of the one of the concrete program. In this case, the answer "true property" of the model checker is still exact, but the answer "false property" actually means "I didn't manage to prove the property, so I don't know".

A simple abstraction is to abstract integer values completely. This is a very rough approximation, that computes a large super-set of the actually reachable states. When the property to verify highly depends on the data, it is not sufficient. This is, for example, the default behavior of LESAR [HLR92]: it replaces any condition depending on an integer value by a non-deterministic Boolean variable.

A less brutal abstraction relies on *local satisfiability* of transition guards, abstracted into convex polyhedra. For example, in the automaton of Figure 3.1, the final control point is unreachable, but we could not have proved it with Boolean analysis only. This is the behavior of LESAR when used with the `-poly` option.

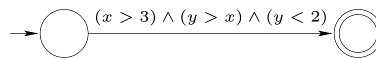


Figure 3.1: Property Provable With Local Satisfiability

3.2.1.3 Abstract Interpretation

Abstract Interpretation [CC77] goes further and has a global view of the possible values of variables at each step of the program's execution. For example, in the automaton of Figure 3.2, the final control point is not reachable, and abstract interpreters like NBAC [Jea03] are able to prove it: in the leftmost state, any value may be possible for x, but in the middle state, the only incomming transition has an assignment, so the only incomming value for x is 42. The intersection of $\{42\}$ with $]-\infty, 0[$ is empty, so, we can not take the next transition. At this point, we have reached the fixed point: no action can provide another value for x that could allow the automaton to take the last transition.

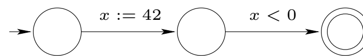


Figure 3.2: Property Provable by Abstract Interpretation

3.2.1.4 SAT engines

Another approach for the problem of verification is to encode the behavior of the system and the property using Boolean formulas, and check for their satisfiability [BCC+99]. The satisfiability problem for a Boolean formula in conjunctive normal form (also known as clause form) is well known, and called *SAT*. Although SAT is NP-complete, many optimization techniques can be applied to make it very efficient in many cases.

One limitation of this approach is that only finite properties can be encoded: it is not possible to express "something wrong will never happen", but we can say "something wrong will not happen in less than n steps". We can apply several strategies based on this remark:

Bounded model checking and bug searching: A "naive" approach is to encode the property "something wrong will not happen in less than n steps" with n sufficiently large, and conclude "If the system is safe for n steps, it is very unlikely to be unsafe for more than n steps". Of course, we can not give a formal meaning to "unlikely" here, which is not very satisfying from the theoretical point of view. Still, this can be better than testing, which is only able to say "something wrong did not happen for some simulation of less than n steps". A less naive approach is to iterate over n, and stop the proof either when the machine goes out of memory because n is too large or when a bug is found.

Exact model checking: Another approach is to prove that the system must have reached its fixed point after n steps. This gives a termination condition to the iterative approach given above. The proof

engine can then either stop saying "the property can not be falsified in less than n steps, and after n steps, the system will loop anyway, so, the property is true" or "I found a counter example of size n. Here it is". In practice, of course, the computation may also be too long or too costly in terms of memory to finish.

SAT can be extended to manage Presburger arithmetics, and treat numerical data. All of this is implemented in PROVER PLUG-IN™ for SCADE™ [Tec] (unfortunately, this is a commercial tool and the details of implementation are not published as of now).

3.2.1.5 The Best Strategy...

The verification problem is known to be exponential in the worst case. Although some algorithms are known to be "usually better" than others, any algorithm will therefore have some weaknesses, and may need millions of years or terabytes of RAM to complete in some cases. The consequence is that the best strategy is to have several strategies. A clever user may be able to choose the best strategy depending on the problem he has to solve, but a much simpler approach is to launch several algorithms in parallel, and to take the result of the first one to complete.

One LUSSY is an open tool: it can already connect to several proof engines, and lets the user choose the one he wants to use. Other back-ends can be added in the future without breaking the internal architecture of the tool.

3.2.2 Candidate Tools for the Verification of SoCs

SystemC designs being *circuit* designs, we could think of using one of the verification tools (model-checkers, SAT-solvers, etc.) developed for hardware verification, for instance SMV. However, these tools are tailored for the RTL, exhibiting a clear notion of clock, while we need to deal with heterogeneous designs. Heterogeneity comes from several places: determinism and non-determinism, synchronous and asynchronous systems, hardware and software components. Moreover, these tools cannot take general SystemC as input.

As far as we know, all the work on verification techniques and tools for SystemC designs are limited to the subset of SystemC that allows to write RTL designs. It cannot be used for real TLM designs (see [DG03] for instance). [SF02] treats the SpecC language (similar to SystemC): a date of execution is associated with each instruction of the program, which can then be considered as a dependency graph, on which synchronization properties can be proved. The approach is very limited: since only one date can be associated with an instruction, this does not allow to consider general loops (inside which each statement is executed several times, at different dates).

Now, since SystemC is mainly a C++ library, one could think that we have to face the same problems as those addressed by general-purpose software model-checking tools. This is not the case: Verifying SystemC designs is, on the one hand *simpler*, because we do not have to deal with general dynamic data structures and general algorithms; on the other hand *harder*, because we have to take parallelism into account, and to know about the scheduler specification. General software model-checking techniques concentrate on dynamic data structures and general algorithms. They provide sophisticated tools like invariant extraction, loop unrolling, etc., but are not directly usable to exploit the particularities of the SystemC constructs provided as a C++ library. Using these tools for SystemC would need to include the non-deterministic scheduler specification in the tool. Moreover, they usually do not take parallelism into account. For instance, CBMC [CK03, CKL04] can apply bounded model-checking techniques on pure C models, but does not deal with parallelism, or with infinite loops. SLAM [BR00] uses clever abstractions and refinement techniques, but also focuses on sequential programs. VeriSoft [God97] can handle parallel processes written in any language. They are executed as black boxes, communicating via calls to operating system primitives. These calls are intercepted to build a model of their parallel behavior. We cannot exploit such a black-box approach, because we need to extract the transaction-level specific constructs of SystemC, and aim at treating addresses in a specific way (see below). Other works propose improvements for runtime verification: some by providing tools to check properties during a simulation (for example [RHKR01]), others by generating sets of test cases with a better code coverage ([BCDM03], [FRS02]). Usually, those

simulation-based techniques scale up better than formal verification, but give weaker guarantees on the reliability of the platform.

A close related work is to be found in Java model-checking, since they take a scheduler specification into account. The first version of the Java Path Finder model-checker [Hav99] used an approach similar to ours, translating Java into the intermediate representation Promela, and using the model checker SPIN to prove the properties. Version 2 [HPV00] checks the byte-code directly, using a dedicated JVM with backtracking capabilities, and lots of other model-checking techniques. However, the techniques dedicated to Java are not directly applicable, neither to SystemC and its scheduler, nor to the modeling of synchronous and asynchronous mechanisms.

More recently, in [KS05], an approach very similar to ours was published. Starting from a SystemC program, the tool also extracts the semantics in terms of automata. Their intermediate formalism is a bit different. It's a flattened view of the automata (all variables are shared by default), and each state contains a set of predicates that hold in this state. This allows some proofs by predicate abstraction [SH97]. This work focuses on proof techniques, but manages only a very strict subset of SystemC, and in particular, nothing about the TLM level. They describe abstraction heuristics for combinational threads and clocked threads, which proved to be very efficient for their case studies, but would not apply on transaction level models, where we have no clock, and very few combinational threads.

An approach similar to ours is described in [Bak95]. This work is prior to the existence of SystemC, so it's obviously not applicable directly here, but provides an interesting overview of existing theories to express the semantics of a program. It presents the execution models in which this semantics can be expressed, and how they compare to each other based on several well defined criteria.

3.3 Our Verification Approach

We advocate an approach able to exploit all the particularities of a TLM design written in general SystemC. The idea is not to express the TLM concepts manually in yet-another-formalism that can be exploited by verification tools, but to be able to take real SystemC designs into account. We describe a method implemented in a new dedicated tool called LUSSY: based on compiler front-end techniques, it is able to extract architecture and synchronization information from a TLM design written in SystemC with very few abstractions, by exploiting carefully the constructs provided by the library. It builds its own intermediate representation called HPIOM (for *Heterogeneous Parallel Input/Output Machines*) made of communicating parallel machines, able to represent both deterministic and non-deterministic components, synchronous and asynchronous communication protocols, Boolean and numerical data. This is very much in the spirit of the *action language* [Bul00]. For the moment LUSSY connects this intermediate representation to model checkers, abstract interpreters, and a SAT engine. These tools provide conservative automatic verification results for safety properties, and may perform their own abstractions on the HPIOM representation, when needed. The current state of the LUSSY implementation is being applied to case-studies provided by STMicroelectronics; it accepts a large subset of SystemC.

3.3.1 Expressing Properties

Generic properties do not require the use of a specification language. In LUSSY we can express and check the following:

- Check that a global dead-lock never occurs. We consider that a global dead-lock occurs when SystemC scheduler enters the “time elapse” phase while no process is waiting for time. In the example of figure 2.3 page 23, this might happen if the processor is waiting for an interrupt, if the other processes are waiting for transactions to process. The particular notion of scheduling of SystemC makes global dead-lock equivalent to the reachability of a state in an automaton.

- Check that a process never finishes: this should always be the case except for test benches. This corresponds to the case when a process goes to the “sleeping” state without any wake-up condition, that is: no `next_trigger` have been called in `SC_METHOD`, and the last statement of the function is never reached for `SC_THREAD`.

- Check that a synchronous signal is never written on twice during the same δ-cycle. This is a dangerous situation since the final value on the signal will depend on the order of execution, which is most probably dependent on the scheduling policy. In the example, if one process of the interrupt controller raises the interrupt signal while another cancels it, there is a data race.

- Additionally, it would be interesting and relatively easy to check that a TLM master port is not accessed by two processes at the same time. This is known to produce undefined behavior with at least TAC ports in their current implementation.

In order to specify and prove user-defined properties of SystemC designs, we need a specification formalism. The idea in LUSSY is that the user should not have to learn a timed logic language. The property should be written in the same language as the implementation. We may check that some portions of code are mutually exclusive. This is slightly intrusive in the source code since the beginnings and ends of the critical sections have to be specified. Finally the most general safety properties are expressed by assertions in the source code: `ASSERT(condition)`.

3.3.2 Synchronization Code Vs. Complex Algorithms

A typical TLM design exhibits a clear distinction between the potentially complex algorithmics of some components, and the code dedicated to synchronization. For instance, a processor may be included in the design, with SystemC code describing the interpreter of its machine language. In this case, the code intented to be run on the processor is provided separately.

If a processor is present in the design, this means treating it in a very abstract way. The program it runs might be checked by other techniques (software model-checking or theorem-proving); the processor component may then be replaced by a very simple SystemC component describing how it is connected to the other components, and abstracting all its behavior. The properties that can be checked on such an abstracted TLM design cannot depend on the details of the algorithms run by the processor, but this is good design practise, anyway.

3.3.3 The LUSSY Tool Chain

The tool chain is presented in figure 3.3. Starting from a SystemC program's source code, PINAPA, the front-end, extracts an abstract representation comprising both the architecture and the syntax related information. BISE use the output of PINAPA to generate a representation of the program using the intermediate representation HPIOM (this is an interpretation of the semantics of SystemC). BIRTH performs some HPIOM to HPIOM transformations. The translation from HPIOM to any synchronous language is then rather straightforward. We currently have a LUSTRE [BCH+85] and an SMV [McM01, McM93] back-end that allows us to use SMV, LESAR [HLR92], NBAC [Jea03, Jea00] and PROVER PLUG-IN™ for SCADE™ to carry the actual proof.

LUSSY is the composition of PINAPA, BISE, BIRTH and the different back-ends.

The following gives a short presentation of each element of the tool chain. They will be detailed in the next chapters.

3.3.3.1 PINAPA: Pinapa Is Not A PArser

PINAPA [MMMC05b] is a SystemC front-end based on GCC and on the SystemC library. Its role is similar to the one of a compiler front-end for a traditional programming language, but the way it works is very different since we are not dealing with a real programming language, but with a library built on top of C++. Chapter 4.3 will detail the role, principle, and implementation of PINAPA.

3.3.3.2 BISE: Back-end Independent Semantic Extractor

BISE [MMMC05a] takes the output of PINAPA as input. It defines the data structure HPIOM (for Heterogeneous Parallel Input/Output Machines), an intermediate formalism of communicating, synchronous

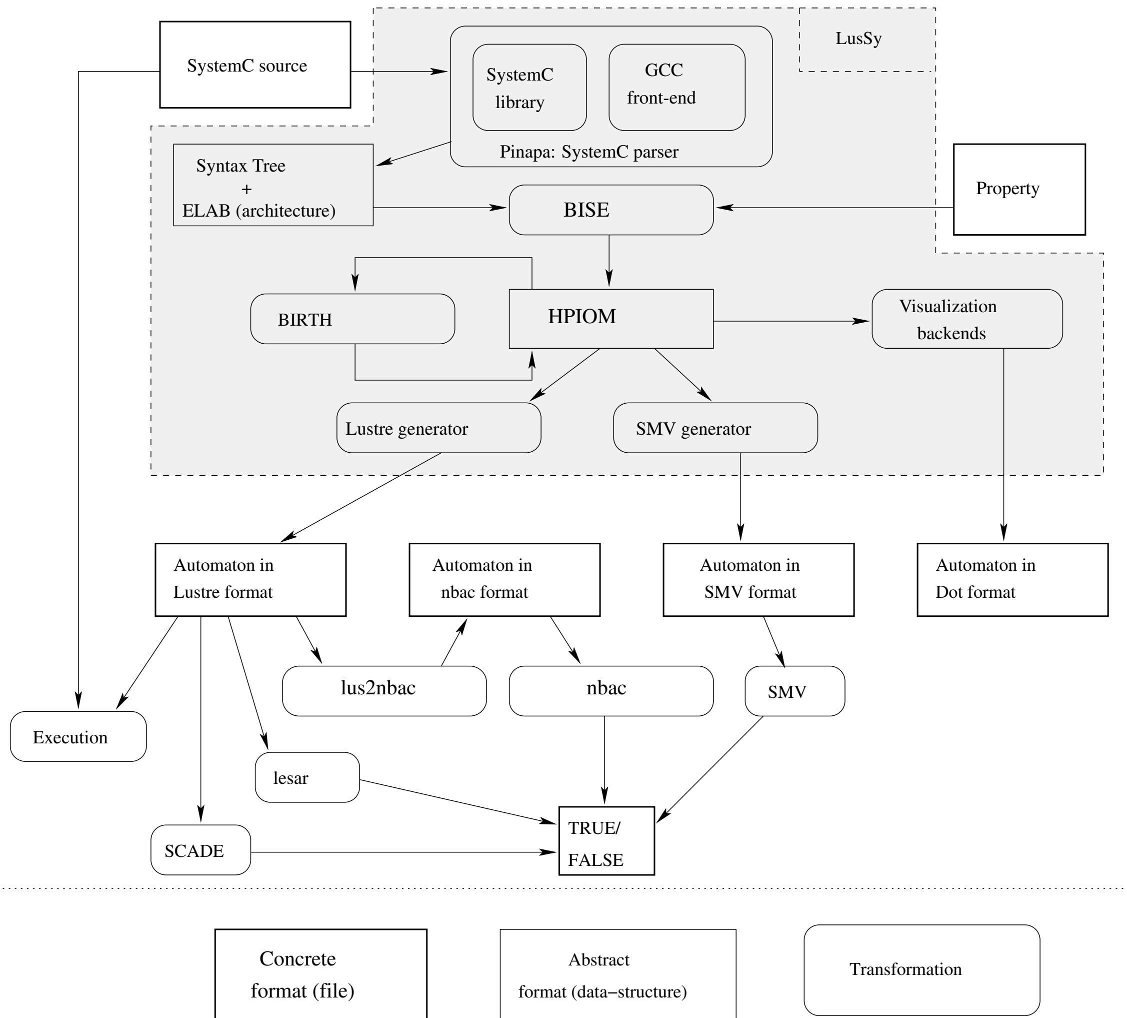


Figure 3.3: The LUSSY Tool Chain

automata. It generates a system of HPIOM automata whose semantics is a conservative abstraction of the input program, with as few abstractions as possible.

3.3.3.3 BIRTH: Back-end Independent Reductions and Transformations of HPIOM

BIRTH implements some HPIOM to HPIOM transformations: some of them are necessary transformations, like expression of high-level HPIOM constructs using lower-level constructs to ease the task of the back-ends. Others are optional abstractions and optimizations, that allow proving larger platforms.

3.3.3.4 LUSTRE and SMV back-ends

The next components in the tool-chain are the back-ends, or code generators. They will transform our intermediate representation into the input format of the external tools we want to use. We can currently generate LUSTRE and SMV code to use the related formal verification tools.

3.3.3.5 Visualization back-end

This back-end was mainly written for debugging purpose. From the HPIOM automata, it generates one `dot` file for each automaton, and one to represent the connection between automata in the system. The tool DOTTY from the graphviz package [EGK+03] reads those files, applies a state placement heuristic and provides an interactive visualization tool. We also implemented an interface to display an automaton at runtime, from the debugger GDB.

This has been very useful not only for debugging, but also for the manual verification of the correctness of the translation: the generated SMV or LUSTRE code is not very readable, and it is very hard to make sure the "shapes" (states and transitions) of the automata are correct, whereas this is almost immediate with a visualization tool.

Part II

Automatic Extraction of Formal Semantics of SystemC Models

Chapter 4

PINAPA: Extracting Architecture and Behavior Information From SystemC Models

Contents

4.1 Introduction **49**
4.1.1 Static and dynamic information in SystemC 50
4.1.2 PINAPA: Requirements 50
4.1.3 Contributions 50
4.2 Related Work **51**
4.2.1 Existing SystemC Tools 51
4.2.2 Other combinations of static and dynamic analyzers 52
4.3 PINAPA Principles, Limitations and Uses **52**
4.3.1 Specifications of PINAPA 52
4.3.2 Limitations 58
4.3.3 Other possible approaches 59
4.3.4 Pinapa: Current and Future Uses 59
4.4 Implementation of PINAPA **60**
4.4.1 Links from ELAB to AST 60
4.4.2 Links from AST to ELAB 60
4.4.3 Architecture of PINAPA 62
4.4.4 Modifications of GCC and SystemC 65
4.4.5 Practical Considerations 66
4.5 Parsing the EASY Platform With PINAPA **67**
4.6 Conclusion **68**

4.1 Introduction

A difference between SystemC and other Hardware Description Language is that SystemC, although often referred to as a *language* (including in the name of its reference manual), is not actually a language, but a library for C++. Execution of a SystemC model is "trivial", since it can be compiled with any supported C++ compiler. But simulation is not the only thing one may want to do with a language.

Static extraction of information is useful for example to synthesize a lower-level view of the model (gate-level synthesis, for example, is possible for a strict subset of SystemC), to visualize it graphically, to generate some documentation automatically, etc . . .

A tool like PINAPA is compulsory for anybody who wants to work on realistic SoC designs: it is able to extract both architecture and behavior information from a piece of SystemC code, with very few limitations. It is open source and available to public.

In the context of formal verification, we will need to get all the static information on the model. Our input is a SystemC model in a set of text files. The first step before being able to do anything else is to get an abstract representation of it.

This part will explain how parsing a SystemC model is different from parsing a program written in a traditional programming language, and how we solved the different problems it raised.

The first section of this chapter (4.2) will present the related works: some tools with the same goal, and some tools using similar approach in other domains. Section 4.3 will present the principles of PINAPA, while section 4.4 presents its implementation. We present our results on the EASY platform in section 4.5, and conclude in section 4.6.

4.1.1 Static and dynamic information in SystemC

SystemC, like several programming languages or runtime environments, is used for describing: 1) the architecture of a system and then 2) the activity of the elements in this system. The architecture, although it is built by the execution of some piece of code (the so called "elaboration" phase), is not really *dynamic*, and will not change during the simulation of the program activity. It is described in a general-purpose programming language because of the expressivity of such languages, compared to the dedicated pseudo-languages of "configuration files".

4.1.2 PINAPA: Requirements

We presents PINAPA (for **P**inapa **I**s **N**ot **A** **PA**rser), our implementation of a SystemC front-end.

PINAPA was developed as the first component of the tool LUSSY, dedicated to formal verification of SoCs described in SystemC. Our main requirements were the following:

1. As few *a priori* limitations as possible. We cannot make any assumption about a well-defined subset of SystemC used in the models we want to analyze. In particular, we don't want the tool to require any manual annotation of the source code to be analyzed.

2. The tool must give precise information on all parts of the model: architecture, software parts, hardware parts.Abstractions may be done in the back-end if necessary, but the front-end must not lose information.

3. Since writing PINAPA was only the first (necessary) step to write a more complex tool, we did not have sufficient manpower to write a big piece of software from scratch, and needed therefore a solution maximizing the code reuse, and minimizing the manpower. Code reuse in PINAPA is also a way to get closer to standard: the C++ front-end of GCC is better than anything we could have written in reasonable time, and reusing the SystemC library for architecture extraction also helps in complying with the SystemC specifications.

4. The models we want to manipulate use some high level Transaction Level Modeling constructs, that are not yet standardized by the SystemC consortium. The tool must be able to manage those constructs.

4.1.3 Contributions

PINAPA satisfies all the abovementioned requirements. The contributions are the following: 1) a general principle for building front-ends of "simulation" languages in which part of the system architecture that has to be extracted statically is actually built by the execution of some piece of code; 2) an open source implementation of this principle for full SystemC (it has been tested on the TLM model of the EASY platform in SystemC as described in section 2.5.2, whose complexity is representative of the designs written in SystemC – although it's relatively small in terms of size); 3) working connections to analysis tools.

When fed with a SystemC model, PINAPA executes its elaboration phase, parses it with GCC, and outputs a data structure useable through GCC and SystemC API, plus some additional PINAPA-specific functions.

4.2 Related Work

4.2.1 Existing SystemC Tools

Several other tools manipulate SystemC models. Some present themselves as SystemC front-ends, but none of them meet our requirements.

4.2.1.1 SystemPerl

SystemPerl [Sny] is a perl library containing, among other tools, a netlist extractor for SystemC (a *netlist* is a description of the connections between modules). It uses a simple grammar-based parser and will therefore not be able to deal with complex code in the constructors of the model, and does not extract any information from the body of the processes. This does not satisfy requirement 2 above.

4.2.1.2 Synopsys™ front-end, SystemCXML

Synopsys™ developed a SystemC front-end that has successfully been included in products like CoCentric SystemC Compiler and CoCentric System Studio [Syn, syn03]. It parses the constructors and the main function, as well as the body of the modules with the EDG [Edi] C++ front-end, and infers the structure of the model from the syntax tree of the constructors. SystemCXML [MBPS] seems to use the same approach, using doxygen's [vH] C++ front-end, but the implementation details are not published as of now. Using this technique, to be able to parse any SystemC model (requirement 1), one must be able to compute the state of any program at the end of the execution of the constructors knowing their bodies. In other words, the tool must contain a re-implementation of a C++ interpreter (which does not satisfy requirement 3).

4.2.1.3 SLEC™

SLEC™ is a formal equivalence checking tool, developed by Calypto Design Systems. The details of implementation haven't been published, but it has requirements similar to ours. It seems they are using EDG as a front-end.

4.2.1.4 ParSyC, sc2v, KaSCPar

The University of Bremen developed a SystemC front-end called ParSyC [FGC+04]. The approach is similar to the one of Synopsys™, except that the grammar is written from scratch (including both SystemC and C++ constructs) instead of reusing an existing C++ front-end. sc2v [Vil] is also a SystemC synthesizer, built with the same approach. KaSCPar recently came into the picture, with a grammar-based parser (using JavaCC), and an XML output. To be complete, this approach needs to include all the C++ syntax (to parse the model) and semantics (to interpret the constructors).

4.2.1.5 Lint tools

Some lint tools such as AccurateC [Act] also manipulate SystemC code. AccurateC can check rules both in the code (this is an extension of a C++ lint tool) and in the netlist. However, it does not need the link between the behavior and the netlist (unfortunately, the internal structure of AccurateC has not been published at time of writing).

4.2.1.6 Simulators

Some simulation tools provide an alternative to the reference simulator, with additional features like VHDL or Verilog cosimulation. These tools do not need information about the body of the processes in uncompiled form, so, their requirements are different from ours. One particular case is NC-SystemC [Cad] from Cadence: it also provides source-level debugging, using the EDG C++ front-end. The approach is therefore similar to ours, since the tool has to deal with both syntax and architecture information. However, this tool is focused on debugging, and the front-end is anyway not available to the public.

4.2.2 Other combinations of static and dynamic analyzers

4.2.2.1 Reverse engineering

The combination of static and dynamic information extraction is used in other domains. In particular, several reverse engineering techniques use a comparable approach: in [HHL02], the dynamic analysis is used to refine the result of the static analysis and eliminate false positive in design pattern recognition, and in [RR02], the static and dynamic information are combined to generate UML diagrams. In both cases, the difference is that the dynamic information extracted relates to the behavior of the model, and not to an elaboration phase as we are doing in SystemC.

4.2.2.2 Graphical User Interfaces

The most similar works are to be found in the domain of Graphical User Interfaces. Most GUI toolkits have this notion of elaboration phase where graphical elements are built and displayed, followed by the behavior of the model which consists in waiting for an event and executing the corresponding action. The difference with hardware modeling is that GUI elements can be created dynamically. Many tools and Integrated Development Environments need to deal with the static part of the interface (in particular, to provide a graphical editor for it). The approach followed by most of them is the one presented in section 4.3.3.2, defining a dedicated language to describe the interface, and providing a code generator or a dynamic loader.

4.3 PINAPA Principles, Limitations and Uses

The methodology for writing or generating front-ends for various kinds of languages has been studied extensively (see for example [ASU86]). Such general techniques are used indirectly in our tool since we are using a general C++ front-end, but are not sufficient to get all the necessary information from a SystemC model. Typically, they cannot extract the information about the SoC architecture, which is built by executing the first phase of the SystemC model.

At first, it may appear meaningless to write a front-end for a library, but the case of SystemC is particular. To understand what we mean by "SystemC front-end", we need to examine the notion of *static* and *dynamic* aspects of a SystemC model.

4.3.1 Specifications of PINAPA

4.3.1.1 Informal Definition of the Static Information in a SystemC Model

Observe Figure 4.1. On the left are the kinds of information present in a SystemC model. From the point of view of a C++ front-end, lexicography and syntax are static and used to build the Abstract Syntax Tree (AST), while the architecture and the behavior are visible during the execution. From PINAPA's point of view, the *static* information extends to include the architecture. The architecture will be present in the memory at the end of the elaboration phase. The dynamic part is reduced to the simulation phase. The static part is made of: the AST obtained by reusing a standard C++ front-end on the SystemC model; the architecture-related information (ELAB) that stays in memory at the end of the elaboration phase.

Figure 4.2 describes the dataflow of PINAPA. The AST is obtained by parsing the program with a traditional C++ front-end (right hand side of the figure), and ELAB is obtained by compiling and executing the elaboration phase (left hand side of the figure).

Type of information	Traditional C++ front-end	Pinapa
lexicography	static: AST	static: AST, ELAB
Syntax		
Architecture	dynamic: elaboration	
Behavior	simulation	dynamic

Figure 4.1: Static and Dynamic information in a SystemC model

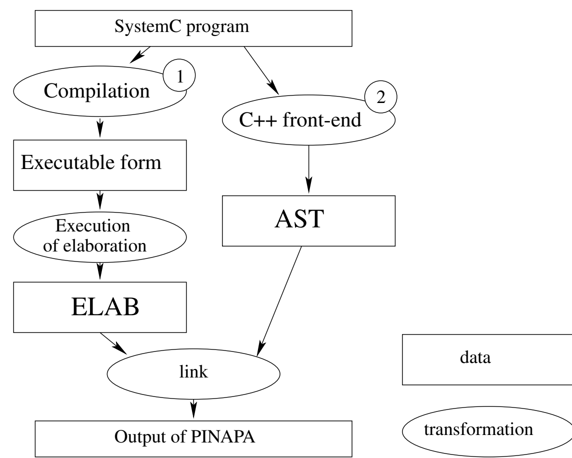


Figure 4.2: Data-flow of PINAPA

Note that the source code is parsed by a C++ front-end twice: first, to compile the elaboration phase (step (1) in Figure 4.2), and then to get the AST,

which, in our case, contains some address offset information (offset of a data-member of a class compared to the address of the class), dependent on the Application Binary Interface (*ABI*) (step (2)). The address offsets obtained in both cases must be consistent, so the C++ front-ends of (1) and (2) must be ABI-compatible, that is, the layout of the data-members of a class must be calculated exactly in the same way (in particular, they can be the same compiler). We currently use GCC (GNU Compiler Collection) version 3.4.1 for both. This means that the model will not be parseable by PINAPA if it does not compile with this precise version of GCC (indeed, *any* parser has the limitation that a program that does not parse with it can not be used. The only difference here is that we reuse an existing work).

4.3.1.2 Overview of the Algorithm

The main task of PINAPA is to establish links between the AST and ELAB (last step in Figure 4.2). The idea behind these links is illustrated by the following: the SystemC processes perform actions of the form `port.write (...)`, and these instructions are present in the AST. The elaboration phase creates instances of modules and connects ports, building an architecture that is present in ELAB. The relationship between an instruction `port.write (...)` in the AST, and the actual data structure describing this port in ELAB, has to be established by PINAPA. In practice, PINAPA installs pointers in both directions between the AST and ELAB.

4.3.1.3 Definition of the Output of PINAPA

The information extracted by PINAPA is mainly GCC's AST, and SystemC's ELAB. We have chosen to keep those data-structures as they are: we add information where needed, but do not perform any transformation.

4.3.1.3.1 Decoration Mechanism. A set of template classes are provided in PINAPA to allow a non-intrusive decoration of GCC and SystemC data structure. The difficulty is to avoid introducing dependencies from the decorated object to the decoration (to avoid dependencies from GCC and SystemC to PINAPA, and from PINAPA to the back-end). In practice, we add a `void *` pointer to the structure to decorate, and wrap it in well-typed decoration primitives. In the case of GCC's AST, some processing of the code during the compilation of GCC forbids such addition. Our workaround is to use a hashtable *tree* → *decoration* instead of a pointer in the data structure representing the node of the tree. Actually, the most important decorations on the AST are not attached to the AST itself: Since a module may be instantiated more than once, the same element in the AST may refer to several objects in ELAB. However, for a given process, an element in the AST only corresponds to one object in SystemC. The link is therefore actually a hash table:

$$(\text{AST}, \text{process handler}) \longrightarrow \text{SystemC object}$$

We call *AST instance* the pair $(\text{AST}, \text{process handler})$. The decoration will therefore be associated to an AST instance.

For example, the port referred to line 10 in the example page 26, and in the AST of Figure 4.4 below has two instances, but the pair (AST of the port, process handler for `instance1_2->code1`) uniquely identifies the port `instance1_2->port`.

The decoration can itself be decorated (so that the back-end can add informations to the output of PINAPA). The result is represented in Figure 4.3.

4.3.1.3.2 SystemC Data-Structure* ELAB *and link to AST. The state of memory at the end of the elaboration phase give the architectural informations about the model to parse. In Figure 2.6, each graphical element corresponds to an object in ELAB, which contains:

Process handlers. The process handler gives the following information: name of the function, name of the class containing it, type of process (`SC_THREAD` or `SC_METHOD`), and pointer to the executable code of the function. It also contains the list of events the process may be waiting for by default after

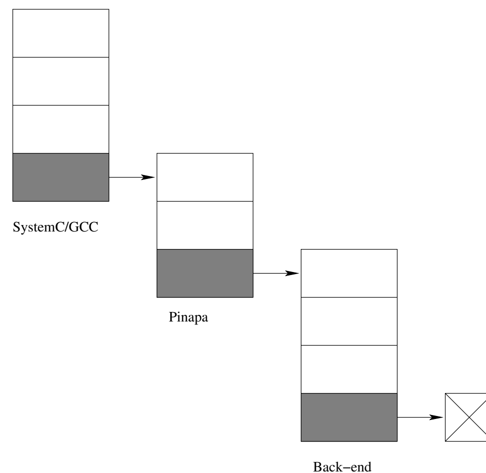


Figure 4.3: Decoration Mechanism

suspending itself. This list is called the *static sensitivity list*. The static sensitivity list just provides a default argument for `wait` and `next_trigger` statements, so a process with a static sensitivity can use `wait()` (with no argument) or omit the `next_trigger` statement, but this is mainly a syntactic sugar.

SystemC Objects. Each SystemC object (ports, modules, ...) contain the necessary information about the binding. In the example above, the port `port` of `instance1_1` contains a pointer to the signal `signal1`, which itself gives the list of connected ports (`ports[0]`).

Since SystemC is optimized for simulation speed, it does not always record all the informations a back-end would expect:

- On each `sc_port`, we add the list of ports bound to it as a decoration. This is useful in the case of port-to-port binding (say binding of port A to port B, itself bound to the interface I), because by default, SystemC "propagates" the binding to the interface without recording anything in the intermediate port (in our case, A is bound to I, but B is completely bypassed.)
- On each slave module, we add the list of connected interfaces (normally, SystemC keeks the list of slave modules in the channel, but not the list of channels in the interface).

For each element in ELAB representing a C++ function, we add a pointer to the AST of this function:

- Process handlers are linked to the corresponding function.
- For each `sc_module`, we keep the AST of all the methods `read`, `write` (for the `BASIC` protocol), `ReadAccess` and `WriteAccess` (for the TAC protocol).

4.3.1.3.3 C++ AST. The AST represents the bodies of the processes. For example, the `if` statement line 9 in Figure 2.4 would be represented as in Figure 4.4.

We want to link AST and ELAB to each other, as shown in Figure 4.5: each process handler will be linked to the corresponding AST, and each mention of a SystemC object in the AST will be linked to its instances (one instance per instance of the process, see below).

The abstract syntax tree of GCC is documented in GCC internal manual [St]. Some additional documentation can be found in the code itself (files `tree.def`, `cp/cp-tree.def` and `c-common.def`).

The AST is implemented by a pointer to a node. A node contains the type of node (a value in an enumerated type), an array of pointers to the children, and additional data, depending on the type of the node. Decorations are added on some node of the tree.

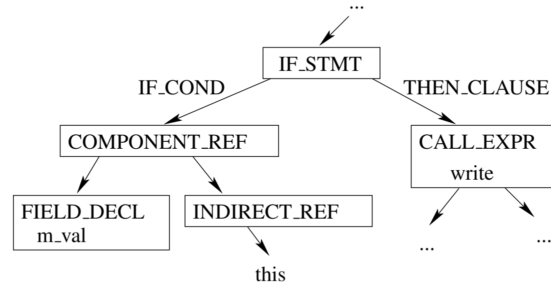


Figure 4.4: Abstract Syntax Tree for an `if` statement

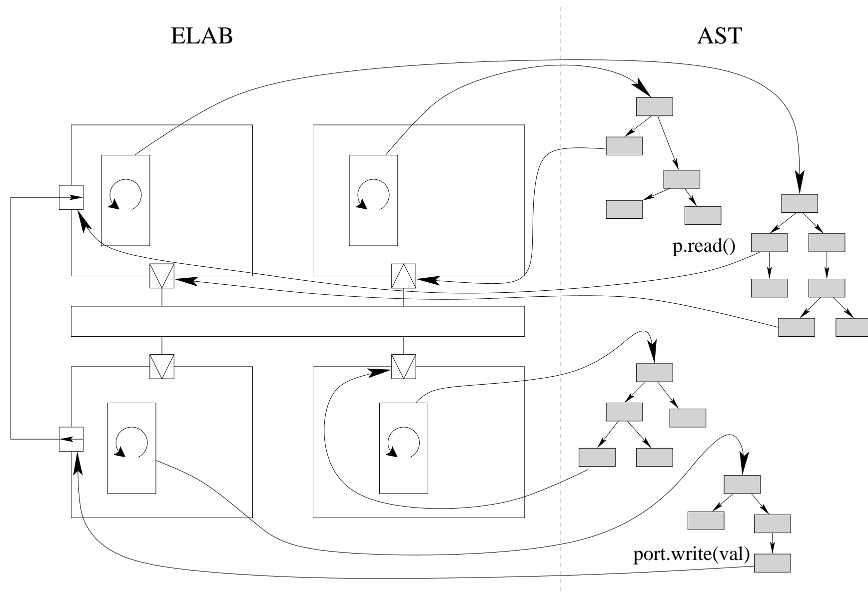


Figure 4.5: Link between AST and ELAB

Type of instruction. The first decoration is the type of instruction. The list of currently supported instructions is the following:

CPP_STANDARD: C++ code unrelated to SystemC
CPP_FUNC_DECL: C++ function declaration
CPP_DATA_MEMBER: A variable which is a data-member of a class
SC_VARIABLE_DECL: Declaration of a variable which is of a SC or TLM primitive time
SCP_TYPE: Type. (decoration of TREE_TYPE field in the tree)
SC_PORT: An object of type `sc_in`, `sc_out` or `sc_inout`
SC_WRITE: A function call `port.write(value)`
SC_READ: A function call `port.read(value)`
SC_MODULE: A module in SystemC
SC_WAIT: A function call `wait(...)`
SCP_BASIC_SEQ_WRITE: A function call `port.write(value)` on a port connected to a `basic_seq`
SCP_BASIC_ARBITER_WRITE: A function call `port.write(value)` on a TLM port connected to a `basic_arbiter`
SCP_BASIC_SEQ_READ: A function call `port.read(value)` on a TLM port connected to a `basic_seq`
SCP_BASIC_ARBITER_READ: A function call `port.read(value)` on a TLM port connected to a basic_arbiter
SCP_TAC_ROUTER_READ: A function call `port.read(value)` on a TAC port connected to a TAC router
SCP_TAC_ARBITER_READ: A function call `port.read(value)` on a TAC port connected to a TAC arbiter
SCP_TAC_SEQ_READ: A function call `port.read(value)` on a TAC port connected to a TAC seq
SCP_TAC_ROUTER_WRITE: A function call `port.write(value)` on a TAC port connected to a TAC router
SCP_TAC_ARBITER_WRITE: A function call `port.write(value)` on a TAC port connected to a TAC arbiter
SCP_TAC_SEQ_WRITE: A function call `port.write(value)` on a TAC port connected to a TAC seq
SCP_TAC_SET_ACCESS_ERROR: `set_access_error()` in a TAC slave method
TLM_IS_ERROR: The function `tlm_status::is_error()`
TLM_IS_OK: The function `tlm_status::is_ok()`
TLM_IS_OVERLAP: The function `tlm_status::is_overlap()`
TLM_IS_NO_RESPONSE: The function `tlm_status::is_no_response()`
TLM_SET_OK: The function `tlm_status::set_ok()`
TLM_SET_ERROR: The function `tlm_status::set_error()`
SC_NOTIFY: Event notification in SystemC
SC_NEXT_TRIGGER: Call to a next_trigger() function in SystemC

Specific types. For AST representing special types, a decoration is also added. The possible values are:

SCP_BASIC_ADDRESS_TYPE: Type of an address in the basic protocol
SCP_BASIC_DATA_TYPE: Type of a data in the basic protocol
SCP_TAC_ADDRESS_TYPE: Type of an address on a TAC channel
SCP_TAC_DATA_TYPE: Type of a data in the TAC protocol
SCP_STANDARD_TYPE: C++ standard type

This is usefull for our back-end LUSSY, who does a special treatment on addresses. Note that C++ typing system does not allow the identification of address types in 100% of the cases. In pieces of code like

```
GLOBAL_DATA_TYPE x = 42;
int y = 56;
```

```
x = x + y;
```

it is easy to identify `x` as being of type address, but `y` will be of type “plain integer”.

Additional decoration on the AST instance. Additionally, for some instructions, an additional decoration is added:

- For all AST representing an `sc_object`, the pointer to the corresponding object is added,
- For `SC_EVENT`, a pointer to the actual `sc_event` is added (we consider this special case because `sc_event` does not inherit from `sc_object`),
- For `CPP_DATA_MEMBER`, the value of the variable at the end of elaboration is added,
- For `SC_WAIT`, we also add a representation of the list of sensitivity (information saying when the process will wake up) for this statement, either based on the arguments of the `wait`, or on the static sensitivity list for a `wait` with no argument. In the example of Figure 2.4, the `wait` statement line 34 has no argument. The list of sensitivity used is therefore the static list built during elaboration (lines 46 and 47): PINAPA will attach the list `{*ports[0], *ports[1]}` to this statement.

4.3.2 Limitations

4.3.2.1 Possible Limitations and Consequences

Unlike other existing approaches, PINAPA has no limitation regarding the complexity of the code of the constructors used to build the architecture, because it does not interpret them; it compiles and executes them. For example, a model reading a configuration file or the command line arguments to determine the number of modules to instantiate can be parsed correctly by PINAPA. In the example of Figure 2.4 page 26, for instance, the initial values of some data-members depend on command-line arguments, but they will be extracted correctly by PINAPA (the command line arguments have to be provided to PINAPA).

Moreover PINAPA (like the front-end of Synopsys™) uses a real C++ front-end and will therefore correctly parse any code that would have been parsed successfully by the C++ front-end. The limitations regarding the C++ language itself are therefore minor (limited to “details” such as the `export` keyword not managed by GCC). The use of macros in the source code is not a problem: the macros will be expanded by the C++ preprocessor. Whether the code uses the macro or its expanded version doesn’t have any influence on PINAPA. For example, whether the user writes

```
SC_MODULE(name) {...}
```

or

```
struct name : public sc_module {...}
```

does not have any influence on PINAPA (the second form may indeed have to be used in the case of multiple inheritance). PINAPA would deal as well with user-defined macros. Any tool using a dedicated grammar for SystemC would have to include all the grammar and typing rules of the C++ standard in the tool to have a correct parser.

While PINAPA has no limitations (except the ones of GCC) regarding the AST (we use a C++ front-end) or ELAB (we let a C++ compiled code *execute* the constructors), it does have limitations due to the way we establish the links between the AST and ELAB.

4.3.2.2 Dynamic references to SystemC objects

It is not always possible to establish these links. For example, if a process uses a pointer to a SystemC port or an array of ports, then, the actual object pointed to by this pointer cannot be known statically. This is the case of `port`, declared line 29 in Figure 2.4. In some cases, advanced static analysis techniques like abstract interpretation would allow to get more information statically, but the subset of SystemC managed

by the tool would be very hard to define. In practice, those constructs are usually not considered as good programming practice and did not appear in the programs used as input for PINAPA up to now.

PINAPA does simply not manage references and pointers to SystemC objects (the pointers to ports will appear in the output of PINAPA, both in AST and in ELAB, but the objects in the AST will not be linked to the corresponding ones in ELAB). For arrays of SystemC objects, if the index is a constant, then, the actual object is known statically, and PINAPA decorates the AST referring to the port with a pointer to this object (this is the case in the instruction `ports[1]` line 27 of the example). Otherwise PINAPA decorates the AST with the index in the array (which is itself an AST) and a pointer to the first element of the array. In any case, we could reduce the case of arbitrary array indexes to the case of constant index by transforming the code to eliminate non-constant indexes, while preserving the semantics of the model. In the example, the transformation would unroll the `for` loop or transform `ports[i]` into

```
i == 0 ? ports[0] : (i == 1 ? ports[1]
                            : (abort(),ports[1]))
```

PINAPA being open-source, such transformation can easily be added if needed.

4.3.3 Other possible approaches

4.3.3.1 Using a C++ Interpreter

An interesting option would be to modify an existing C++ interpreter like UnderC [Don]. A C++ interpreter contains a C++ front-end, and the environment to execute the elaboration phase. Ideally, the C++ interpreter should be 100% compliant with the C++ standard, and do the interpretation at the AST level (not on an intermediate byte-code representation, which is unfortunately the case of UnderC) to ease the link between the AST and the runtime information. We are not aware of any such interpreter.

4.3.3.2 Avoiding the need for a SystemC front-end

The problem solved by our approach is the expressivity of the language used to describe the model's architecture. Another approach would be to eliminate the problem instead of solving it, by using a less expressive language.

In particular, the SPIRIT [SPI] XML Schema can be used to describe the architecture of the model. The version 2.0 with a support for TLM constructs is expected for the end of the year 2005. Extracting the structure of the model would then consist in parsing an XML file, and extracting the body of the processes would still have to be done with a C++ front-end. Simulation of the model would also be possible, by generating C++ from XML and compiling it as usual.

This approach is not applicable today since we need to deal with existing SystemC models.

4.3.4 Pinapa: Current and Future Uses

We currently use PINAPA as a front-end for our formal verification tool LUSSY. Starting from the abstract representation of the model provided by PINAPA, we generate an intermediate representation (a set of communicating automata) which is itself dumped in a text format used as input for a traditional model-checker. This will be detailed in the following chapters.

STMicroelectronics is currently developing a visualization tool for SystemC using PINAPA: reading a SystemC program, it generates another representation usable by a visualization tool (using the `dot` SPIRIT format). This is a very simple use of PINAPA since it only use the ELAB part of the information extracted, but it is being extended to provide more advanced visualization including static information about process communication and synchronization. Our medium-term plans include the development of a lint tool for SystemC and our TLM methodology. The tool has to be able to identify both the architectural and the language constructs, which is exactly the scope of PINAPA.

PINAPA has also successfully been used by a research project for compositional verification of transactional models of Systems-on-a-Chip, led by the POP ART team of INRIA Rhône-Alpes (France) in co-operation with STMicroelectronics. The tool, called (temporarily) SC2PROM, has been started by Nico-

las Palix (see [Pal04]), and continued by Yvan Roux (see [Rou05]). Its internal structure is similar to the one of LUSSY, described in the following chapters.

From the feedback we get on the public mailing list of PINAPA, we can say that PINAPA is at least used for one project of formal proof in Infineon, and a project of tool to support the platform based design methodology by Humberto Rocha for which he needs to extract behavioral and structural information from a SystemC design. I also got a person to person email discussion with someone interested in writing a simple synthesizer based on PINAPA.

4.4 Implementation of PINAPA

The previous section presented the output, and the limitations on the input of PINAPA. We will present here the internals of the tool.

The execution of PINAPA can be divided into three main tasks:

1. Get the ELAB information by executing the elaboration phase,
2. Get the AST of the process bodies using GCC,
3. Make the link between the results of phases 1 and 2

Phases 1 and 2 are just software reuse. For phase 1, fortunately, SystemC keeps a list of most objects in a global variable, it is easy to examine them.

Concretely, PINAPA first launches the elaboration of the model. We use a slightly modified version of SystemC, in which we redefined in particular the function `sc_start()` called by the program at the end of elaboration. Instead of launching the simulation, our version of SystemC launches a C++ front-end. A few other minor modifications have been necessary. They will be detailed in section 4.4.4.

In phase 2, GCC parses the functions one by one. We actually ignore many of them, since we are only interested in the body of processes. We get an abstract representation of the source code of the processes in the form of an Abstract Syntax Tree (AST).

Then, the actual job of PINAPA begins: we have to make the link between this AST and ELAB. Sections 4.4.1, 4.4.2 and 4.4.2.3 below detail some interesting problems raised by this phase. Section 4.4.3.3 summarizes the architecture of the tool.

4.4.1 Links from ELAB to AST

The first step is to make the link from ELAB to the AST. There is not much to do: for each process handler, look for the AST of a method with no argument whose class name and function name match the ones in the process handler, and add a pointer to this AST in it.

In the example page 26, there are two process handlers for `module1::code1()` (one for each instance of `module1()`), and each of them points to the AST of function `module1::code1()` declared at line 8.

4.4.2 Links from AST to ELAB

The link from the AST to ELAB is a bit more complex. Each instruction in the AST corresponding to a function or object of the SystemC library must be considered as a SystemC primitive and requires a special treatment.

4.4.2.1 SystemC Functions

SystemC function calls (in the process bodies) are recognized by their name and list of arguments. We add a decoration to the tree saying that this function is a SystemC function (and which one it is).

For `wait()` statements, PINAPA provides the list of sensitivity used for the statement. We defined a unified data-structure for the possible `wait()` arguments and static sensitivity list. From the back-end point of view, there's no difference between a `wait()` with no argument (thus, using the static sensitivity list) and the `wait(...)` with the equivalent arguments. In practice, we added a callback in SystemC that

builds the static list of sensitivity during elaboration, and we decorate the `wait()` AST either with this list or a list computed from the arguments.

4.4.2.2 SystemC objects

SystemC objects require much more work. In the AST, we get an abstract representation of the classes, but in ELAB, we have *instances* of these classes. These instances are built once and for all during elaboration. Unless the model uses pointers to SystemC objects, a variable containing such object will therefore always contain the same object.

We describe two methods to get a pointer to an object in ELAB from its AST and process handler, and how we applied them in the case of GCC. Depending on the information present in the AST, either one method, the other, or both can be applicable using another C++ front-end, depending on the information provided by this front-end.

4.4.2.2.1 An Example: SystemC Communication Ports. In the case of GCC, SystemC communication ports correspond to a situation where the name of the object does not appear in the AST. This is due to the way GCC represents a member function call in the AST: for example, when the user writes
```
port.write(x);
```
in a process body, if `port` is a member of the current class, this is equivalent to
```
this->port.write(x);
```
which is itself converted to
```
write(this->port, x);
```
by GCC's front-end. Now, here is the bad joke: this code is converted to
```
write(*(this + offset_of_port), x);
```
where *offset_of_port* is a literal numerical constant. At this point, we are still in GCC's front-end, but we have lost an important information: the name of the port.

So, we only have the offset of the port being examined, and we want to get its instance in ELAB. Since the compiler used for the C++ front-end and the one used to compile the model are ABI-compatible, the solution is the following: For each instance of the process, we can get a pointer to the instance of the class containing the process (this information was already in the original SystemC's process handlers). If we add the offset we got from the AST to the value of this pointer, we get a pointer to the instance of the port.

4.4.2.2.2 Other objects. The same approach is used for other SystemC objects like `sc_event`.

4.4.2.3 C++ Classes Data Members

It is often the case that a data member of a class is initialized during elaboration, and we would like PINAPA to be able to extract this information from the model. PINAPA provides an option to read the value of these data members at the end of elaboration.

The problem is that in this case, the address offset does not appear in the AST in the output of the front-end of GCC. The approach of section 4.4.2.2.1 is therefore not applicable. We could compute the offset from the AST (GCC does this anyway, later in the compilation flow), but we chose a different approach, that does not require this computation.

Since we have here both the name of the data member and the name of the class it is a member of, we can write a piece of C++ code that would read the value of this data member. The C++ language is not flexible enough to execute dynamically this piece of code, but never mind: we can write it in a file, compile it (run `g++` as an external program), load it dynamically (`dlopen`, `dlsym`, ...), and execute it. It will be executed in the environment ELAB. Figure 4.6 shows an example of such generated code.

In the current implementation, the return value is converted into an AST representing the value of the constant, which is attached as a decoration to the AST of the model.

There is a limitation here because the return value of the generated function has the same type as the data-member that we are examining, which can be any type. To be able to call this function from PINAPA, we have to know the return type statically. Concretely, this means we need to write a piece of code in PINAPA for each data-type we want to manage.

```
#include "preprocessed_sc_source.cpp"

namespace pinapa {
struct get_value {
  static bool
  function_to_get_value_0(sc_module * arg) {
    return (static_cast<module1 *>(arg))->m_val;
  }
  [...]
};
[...]
} // namespace pinapa
```

Figure 4.6: Example of generated code getting the values of data members

In a future version, it would be interesting to implement the conversion from a concrete value to an AST in the generated code itself. This way, the return value of the function would always be an AST, and this would remove the above limitation. In other words, code generation can be generic on the type of the variable, whereas function calling can not.

4.4.3 Architecture of PINAPA

4.4.3.1 Modules, Dependencies, and Link

PINAPA reuses code from GCC and the SystemC library (both of them in a slightly modified version), and dynamically loads the model to parse.

Some constraints must be satisfied:

- The user's model must not depend on PINAPA's code. We do not want to limit ourselves to models written explicitly for PINAPA.
- The modified versions of GCC and SystemC should not depend on PINAPA itself. This is not a strong requirement, but simplifies greatly the build system of PINAPA. This allows to compile and install the modified GCC and SystemC once and for all, and recompile only the heart of PINAPA when some modifications are done on its code. Furthermore, adding dependencies from GCC to PINAPA would imply to modify GCC's build system, which is extremely complex (a "bootstrap" system more than a simple "build" system actually).
- The back-end must not depend on the front end. We want PINAPA to be a library callable in the standard way: the writer of the front-end writes a program with a main function, and can call the main function of the back-end at any time.

The dependency graph must therefore be the one of Figure 4.7.

To achieve this goal, function calls from the modified GCC or from the modified SystemC are done through callbacks (function pointers in C, functors in C++). The callbacks are declared and used in GCC and SystemC, and PINAPA is in charge of initializing them.

Another constraint is that we want to let the back-end execute some code before loading the model to parse (in particular, in LUSSY, the model to parse can be specified from the command line, so, command-line parsing must be done before being able to load the model). For this reason, loading the model is done at runtime using the `dl` library (`dlopen` to load the library, and `dlsym` to fetch its `sc_main` symbol). The link mechanism is presented in Figure 4.8.

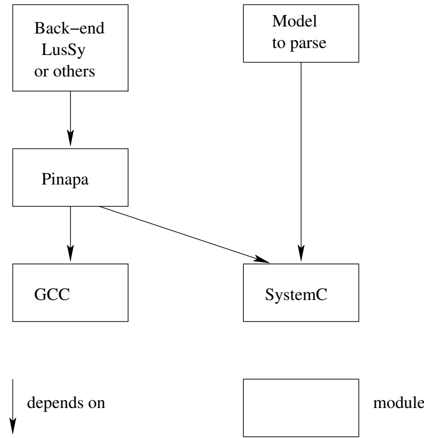


Figure 4.7: Dependencies in PINAPA

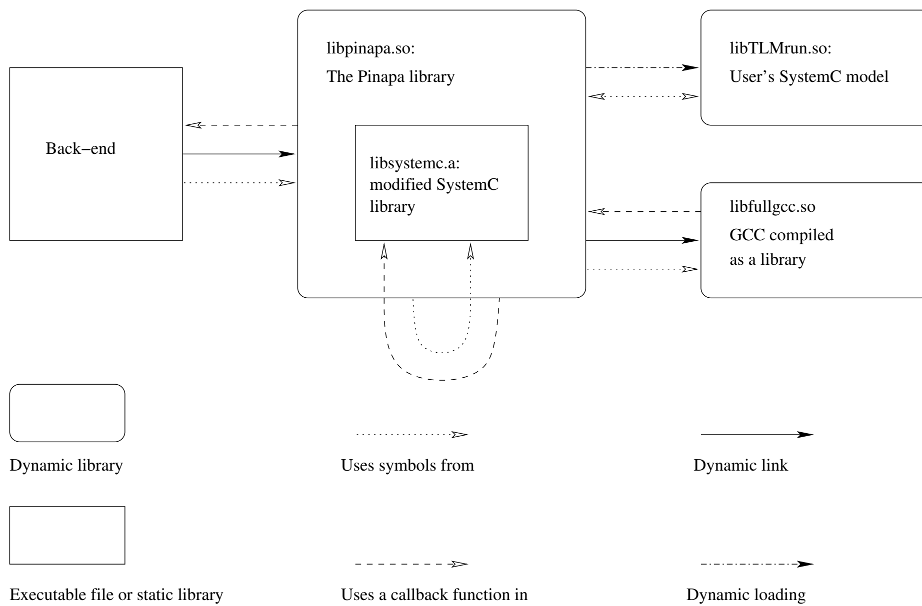


Figure 4.8: Link Mechanism in PINAPA

4.4.3.2 A note on module hierarchy

SystemC provides a notion of hierarchical component. This means that an object (module, port, ...) can be instantiated as a sub-component of another one. While this is essential to organize the design, it has no influence on the semantics. For completeness, PINAPA keeps a pointer `parent` in each object, pointing to the containing object, but other components of LUSSY will not use it.

4.4.3.3 Function Call Graph

The resulting function call graph in PINAPA is somewhat complex (Figure 4.9), but will be made clearer by the end of this section.

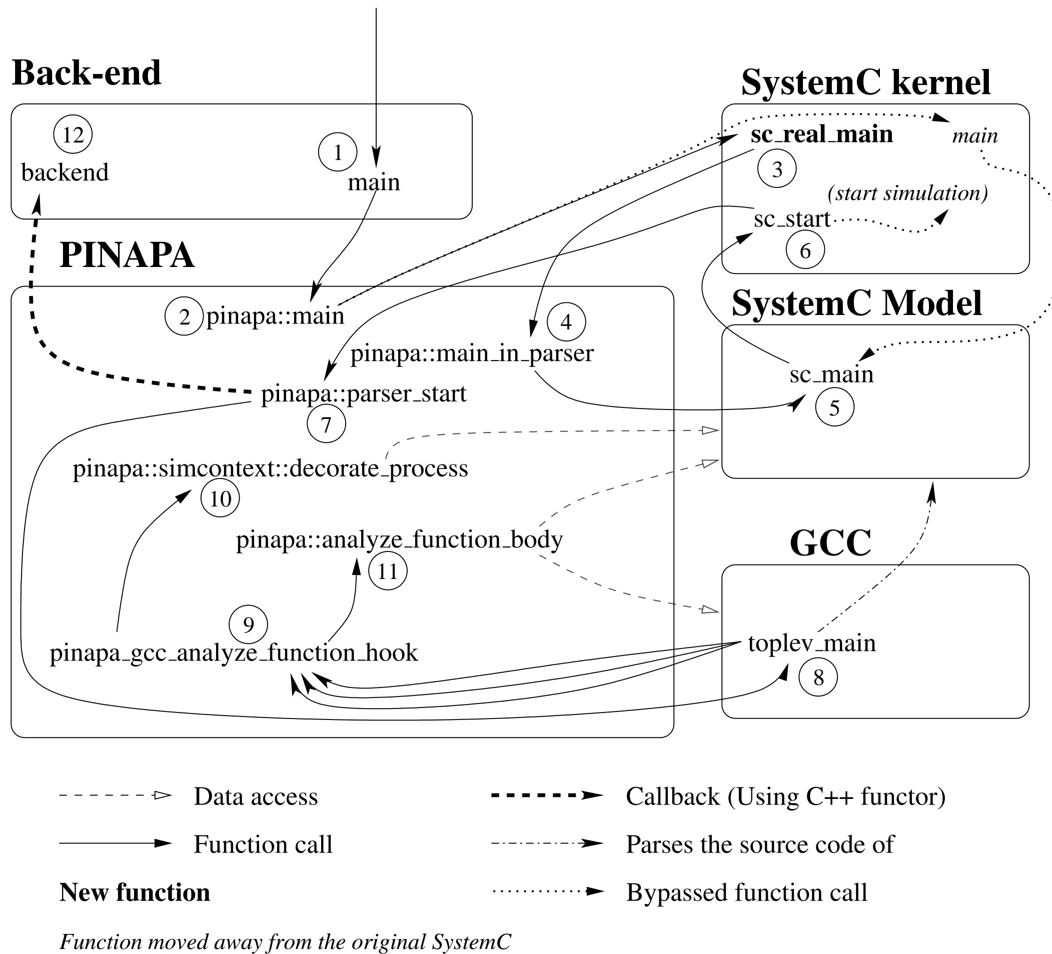


Figure 4.9: Architecture of PINAPA

In the original version of SystemC, the `main` function is in the SystemC library itself (it actually does not do much more than displaying a copyright message and calling the `sc_main` function). We have removed it from the library, considering that the main function should be written by the user (i.e., the programmer of the back-end). This is the item (1) of Figure 4.9. Having the `main` function in a library can be acceptable in the case of SystemC, since it is deliberately a library which is very intrusive in the code using it (which is the reason why there is sometimes a confusion between a language and a library), but this is not acceptable in the context of a more traditional library, where the user wants to *use* the library, not to be constrained by it (in particular, it is not possible to use two libraries containing a `main` function at the same time).

The call to the `sc_main` function (and therefore the call to the PINAPA's main function) must not return, because the elaboration phase may have allocated objects on the stack. We use therefore a callback mechanism (using a C++ functor), so the main function of the back-end should look like:

```
int main(int argc, char ** argv) {
  // The backend is in my_callback::operator()
  pinapa::st_backend *callback = new my_callback();
  // ...
  pinapa::main(..., callback);
}
```

From the `pinapa::main` function (2), we call the function that was originally the `main` function in the SystemC kernel (3), which in turn calls the `pinapa::main_in_parser` function (4), which dynamically loads and executes the user's code (5) to elaborate the model. The call to `sc_start` (6) that originally started the simulation is bypassed and calls `pinapa::parser_start` (7).

The elaboration has now been executed. We call the main function of the GCC compiler (8). We have modified GCC to call the function `pinapa_gcc_analyze_function_hook` (9) in PINAPA for each function it parses (passing the AST of this function as an argument). For each function parsed, `pinapa::simcontext_decorate_process` (10) searches for the corresponding process handler in ELAB and `pinapa::analyze_function_body` (11) runs over the AST to link SystemC primitives to their corresponding object.

4.4.4 Modifications of GCC and SystemC

4.4.4.1 Modifications of GCC

4.4.4.1.1 C/C++ compatibility. The first modification performed on GCC's source code was to make the header files defining the tree data structure compilable in C++.

4.4.4.1.2 Callback to let* PINAPA *Examine the AST. The most important modification was simply to add a function call from the code of GCC to PINAPA, taking the AST of the function being processed as argument. Ideally, we should have extracted the C++ front-end from GCC, and the result would have been a library with a function to parse the C++ code, returning a list of the AST of the functions of the program. Unfortunately, in GCC's architecture, the front-end and the back-end are tightly coupled, and separating the front-end would represent a huge amount of work. Our solution is to "inhibit" the back-end by setting the flag `syntax-only` (the equivalent of the command-line argument `-fsyntax-only` for the GCC executable).

4.4.4.1.3 Avoid Garbage Collecting. Although written in C, GCC implements a garbage-collector on its internal structure. During compilation, some objects were not referenced anymore, except by the annotation we added in ELAB, which are not taken in account by the garbage-collector. We had therefore to add a dummy reference to those objects to avoid garbage-collection.

4.4.4.1.4 Inhibit Some Optimizations. Some of GCC's optimizations lead to loss of information in the AST (even without enabling the optimizations):

- In a function in which all `return` statements were returning the same variable, the "named value optimization" was loosing the AST representing this variable (it was replaced by a `NULL` pointer in the AST). We keep a reference to this value before it is deleted.
- During constant propagation, GCC does implicit conversion from `boolean` to `integer`. Here also, we keep a reference on the AST of the boolean type before it is replaced by the `integer`.

4.4.4.2 Modifications of SystemC

4.4.4.2.1 Make internal data available. The SystemC library was clearly written for simulation, not to let the user explore the internal data structure. PINAPA abuses the library to build a SystemC front-end on it, but we had to make a few modifications to make this possible. Many classes had private members, accessible only to the simulation kernel. We added `friend` and `public` keywords to those classes to allow PINAPA to access those private data. In addition, some data-structure (like process tables) were considered internal by SystemC and were declared in `.cpp` files. We had to move those definitions to the appropriate header file.

4.4.4.2.2 Code added. During elaboration, PINAPA collects a few more information than the original SystemC does. This is the case for example for the construction of sensitivity list described in section 4.3.1.3.3, or for port binding described in section 4.3.1.3.2. This is done by adding function calls to PINAPA function in SystemC.

The macros `SC_THREAD` and `SC_METHOD` have been modified also, to keep the name of the class in addition to the name of the process and the name of the module's instance in the process handler.

4.4.5 Practical Considerations

4.4.5.1 Tools used to build PINAPA

PINAPA is written in C++. The choice of the language was rather straightforward: the SystemC library is written in C++, and makes heavy use of the language's advanced features (class, multiple-inheritance, templates, ...). Since PINAPA use SystemC, and accesses the internals of the library, the benefit of using a better language would not compensate the effort needed to write a binding for another language.

We use the autotools[1] for the build system. AUTOCONF allows an easy configuration management, while AUTOMAKE takes care of generating a `Makefile` with an efficient dependencies management. We also use LIBTOOL to generate libraries. The project was also used as a pilot project to evaluate the autotools, and the potential benefits of using them in other projects of the team.

The documentation of PINAPA is generated with doxygen (a C++ equivalent for JavaDoc). We used both per-functions or per-class documentation and separate pages for more global documentation.

4.4.5.2 Open Source Release

After developing PINAPA internally, for the needs of the back-end LUSSY, we decided to distribute it publicly under a Free Software [Fre] (mostly synonymous of Open Source Software [Ope]) license.

4.4.5.2.1 Motivations. Our main motivation for this choice was to improve the quality of PINAPA with a reasonable effort. Making a program open source is a way to increase the number of potential users (and therefore testers), and the number of potential contributors. Up to now, we received as external contributions a guide on the way to compile PINAPA on a Windows platform. "Unfortunately", the author decided to switch to a Linux platform before completing the windows installation, so, the guide is incomplete, and we actually can't estimate the remaining effort for a windows port of PINAPA.

A counter-argument would be that making PINAPA open source makes it harder for STMicroelectronics to sell it (it's possible to sell free software, but to sell something that anyone could get for free otherwise, we would have to provide some added value). The position of STMicroelectronics regarding this is that selling a SystemC front-end is anyway not the mission of the company, and we estimated that we would anyway not make money with PINAPA.

4.4.5.2.2 Licensing. Since PINAPA links against GCC, which is licensed under the terms of the GNU General Public License, and SystemC, whose license is not GPL-compatible, the software as a whole can not be distributed. However, distribution of separate packages as source code is allowed.

[1] http://sources.redhat.com/autobook/

The GPL does not satisfy us in the context of PINAPA: we would like to be able to distribute some components of LUSSY under Non-Disclosure Agreement for example, and we have no hope to have PINAPA adopted by major CAD companies if we do not allow them to link PINAPA against their commercial and proprietary tools. We can't change the license of GCC, but we may have a workaround for it in the future (using an intermediate file to transmit the information from GCC to PINAPA). We don't want to get locked by the license of PINAPA itself (If external contributions are merged in PINAPA's mainline, we won't be able to change the license without the agreement of all contributors). On the other hand, distributing PINAPA under a non-copyleft license represents a risk for us: someone could fork PINAPA and make proprietary software out of it without contributing back to STMicroelectronics.

The GNU Lesser General Public License is a good compromise: it allows proprietary or confidential software to be dynamically linked to PINAPA, but the core of PINAPA remains protected by the license: Anyone distributing a modified version of it must distribute the source code under the GNU Lesser General Public License too.

The license scheme chosen is therefore the following: the patches for GCC and SystemC are distributed under the terms of the same license as the software they modify (respectively the GNU GPL and the OSCI license), and the core of PINAPA is distributed under the terms of the GNU Lesser General Public License (also known as the LGPL).

PINAPA is now part of GreenSoCs (`http://www.greensocs.com/`), a project to build an open source infrastructure for SystemC. This gives us more visibility than being a separate project, and didn't impose unreasonable constraints. PINAPA is therefore available on `http://greensocs.sourceforge.net/pinapa/`. The site provides a download area with a mirror of the main GNU Arch archive, nightly snapshot as compressed `tar` archives, the generated doxygen documentation (which is more detailed than this document regarding the implementation details). We also use sourceforge's services for the bug tracker and the mailing list.

4.4.5.2.3 Multi-site infrastructure. A practical constraint was the management of different development sites. The site of ST being protected by a very restrictive firewall. The solution adopted is the revision control system BAZAAR (patched to manage SSL and authenticated HTTP proxy), and a WebDAV server located in Verimag. This was also used for code sharing between ST and INRIA. The master archive is hosted on Verimag's web server, and mirrored periodically on sourceforge by a cron job running in Verimag.

We use sourceforge's compilation farm to generate the nightly snapshot and the online documentation. A cron job runs on the compilation farm every day, gets the latest version from the archive, creates the `tar` archives, generates the documentation, and upload them all on the web site.

With this architecture set up, to change the online documentation on sourceforge from either ST or Verimag, one just has to modify the doxygen comments, commit, and wait for a few hours (even behind ST's firewall !).

4.4.5.2.4 Validation. We developed PINAPA incrementally, following our needs for the formal verification back-end LUSSY. Each feature added to PINAPA was validated by at least one example, stimulating both the front-end and the back-end. The correctness of the translation can be ensured by the examination of the model-checker's diagnosis compared to the simulation behavior, and by the visualization tools connected to LUSSY.

4.5 Parsing the EASY Platform With PINAPA

The EASY platform served as a test-case for PINAPA. Since PINAPA isn't able to parse several files separately, we moved the code from the `.cpp` files (C++ bodies) to the corresponding `.h` files (C++ header). We could also have created one file including the others, or used the `-include` option of GCC. No other modification of the code were necessary regarding PINAPA.

As a result, the time spent in PINAPA to parse the EASY platform (less than 20 seconds on a Intel(R) Pentium(R) 4 CPU 2.80GHz, with GCC compiled in debug mode with all optional type-checking enabled)

was even smaller than the time to compile it with GCC (more than one minute on the same machine), because we avoided the overhead due to separate compilation (but also lost its benefits). Note that since the input of PINAPA includes the compiled form of the platform, the total time to use PINAPA is the parsing time plus the normal compilation time.

Since the main goal of the Ph.D was to perform formal verification, we didn't take time to test PINAPA on larger platforms (remember that PINAPA, without a back-end, is irrelevant, so testing other case-studies would need to have a working back-end and the way to check the correctness of its output).

4.6 Conclusion

We presented PINAPA, a front-end for SystemC. Unlike traditional compiler front-ends, it executes a part of the program before parsing it, and the main work presented in this paper is the way to make the link between the source code representation and the runtime information.

This technique allowed us to write a SystemC front-end with very few limitations, with a minimal effort. It reuses megabytes of source code from GCC and SystemC, but counts itself less than 4,000 lines of code. The performances are reasonable: most of the time is spent in GCC, so parsing a model with PINAPA takes almost the same time as compiling it with GCC. It already manages the TLM TAC and TLM BASIC extensions of SystemC, and other could be added in the future depending on our needs.

The parser is already operational and used in two formal verification back-ends and a prototype of visualization tool.

Chapter 5

HPIOM: Heterogeneous Parallel Input/Output Machines

Contents

5.1 Introduction **69**
- 5.1.1 The Need For an Intermediate Formalism 69
- 5.1.2 Design Choices 70
- 5.1.3 Contents of the chapter 70

5.2 HPIOM Basic Concepts **71**
- 5.2.1 Syntax of HPIOM 71
- 5.2.2 Semantics of HPIOM 74
- 5.2.3 Non-determinism 76

5.3 Additional constructs **76**
- 5.3.1 Convenience constructs 76
- 5.3.2 Abstract Addresses 77

5.4 Expression of Properties in HPIOM **81**

5.5 Implementation of and Manipulation of HPIOM **82**
- 5.5.1 HPIOM Expressions and the Composite Design Pattern 82
- 5.5.2 HPIOM Expressions and the Visitor Design Pattern 83
- 5.5.3 On-the-fly Optimizations 88
- 5.5.4 Future improvements 90

5.1 Introduction

The previous section presented PINAPA. From the source code of a TLM model, it is able to provide an abstract representation of the syntax and architecture of the model. At this point, no transformation involving the semantics of the program has been applied to the model.

5.1.1 The Need For an Intermediate Formalism

To be able to apply any kind of formal methods to the model, we need to transform the output of PINAPA into a structure with a well-defined, formal semantics.

Starting from the abstract representation of the model provided by PINAPA, our final goal is the connection to a formal verification tool. A direct connection would be possible, but not desirable: first, mixing the interpretation of the semantics of SystemC and the generation of the target language in the same piece

of code would lead to complex and unmaintainable code, and most of all, this would require a complete rewrite for each back-end tool to manage.

We need therefore an intermediate formalism. The requirements are the following:

Well defined formal semantics: The transformation of SystemC into this formalism is a way to give a semantics to SystemC. It only makes sense if the semantics of the destination language is itself clear;

Powerful enough: The formalism should of course be expressive enough to represent the semantics of the model;

Simple to manipulate: The idea is to perform as much as possible of the work before the intermediate formalism in the transformation flow. Any transformation of a complex structure into a simpler one done before the intermediate representation is done once and for all. Any transformation coming after it in the flow may have to be re-written for each back-end;

Executable: To get confidence in our algorithms, a good solution is to test the generated code against the official SystemC execution engine. The formalism being executable is crucial to allow validation and easy debugging of the generated code;

Provable: Our final goal is to perform formal verification. The proposed formalism should therefore allow it. This requirement is not very different from the "Well defined formal semantics" one.

For a more general discussion about the expected qualities of a formal language, see appendix B.

5.1.2 Design Choices

5.1.2.1 Reusing an Existing Format

We experimented the IF toolkit [BGO+04], but quickly realized that the effort to learn, and adapt the toolkit to our needs, was much higher than redeveloping an automata manipulation library from scratch.

We did not find another flexible enough automata library under reasonable license conditions, and decided to develop our own formalism and manipulation library. This formalism is called HPIOM, for **H**eterogeneous **P**arallel **I**nput/**O**utput **M**achines. It is inspired from synchronous languages like Esterel [Ber00] and Argos [MR01].

5.1.2.2 Expressiveness Choices

The above-mentioned requirements are sometimes contradictory, and some choices have to be made. In particular, "Provable" is not always compatible with "Powerful enough". The more expressive a language is, the less provable it is. For example, proving properties like code reachability and termination on a Turing-equivalent language is not decidable, while it can be decidable, or even trivial on less expressive languages. In our context, it would not really make sense to privilege expressiveness: Having a more expressive intermediate formalism would allow us to translate more source programs with a complete preservation of the semantics, but would not allow us to prove more properties if the proof engine does not support those constructs. We have chosen to design HPIOM as a finite automata formalism, without any dynamic process creation and dynamic data-structures.

5.1.2.3 Limitations of LusSy due to HPIOM Design

The choice of a limited expressiveness for HPIOM has some consequences on the possibilities of LusSy. Since we have no dynamic data-structure in HPIOM, and no easy way to simulate it (we have unbounded integers on which we can encode anything in theory, but that's not reasonable in practice), any SystemC program using dynamic data structure or non-terminal recursive calls will have to be approximated during the conversion. We'll manage to perform the approximation in a way that preserves the properties we want to verify.

5.1.3 Contents of the chapter

This chapter will define an intermediate formalism called HPIOM. It is a simple model of communicating automata, simple enough to be manipulated easily, and powerful enough to express the semantics of a

TLM model. We start with a presentation of expressions (section 5.2.1.1), and use it to build automata (section 5.2.1.2). Section 5.2.2.3 presents the semantics of a system containing several automata. Section 5.3 adds some constructs that can be reduced to the primitive ones that we've added to HPIOM, including the encoding of addresses into Boolean (section 5.3.2). We describe a simple way to express properties on an HPIOM system in section 5.4 and section 5.5 presents the implementation.

5.2 HPIOM Basic Concepts

5.2.1 Syntax of HPIOM

HPIOM is a formalism of communicating, parallel automata with a synchronous semantics. An automaton is a set of control points linked by edges. An edge has a guard, a set of parallel assignments, and a set of signals emissions. We detail the notion of expression and the notion of condition first, and build the syntax and semantics of edges and automata based upon it.

One particularity of HPIOM is that it has no concrete textual syntax, and is only available as an abstract data-structure. We will however use a graphical syntax to make the explanations clearer. The current implementation of HPIOM can not import such graphics, and is only able to export an incomplete graphical representation (used for debugging purposes).

5.2.1.1 Expressions and Conditions

HPIOM defines two kinds of "expressions": HPIOM expressions that can be of any type, and evaluate to a value of this type, and HPIOM conditions, that can be true or false. Only HPIOM conditions can be used as guards or condition for a conditional expression, but conversion operators between expressions of type Boolean and condition are provided. Both expressions and conditions are purely functional (no side effect is allowed).

5.2.1.1.1 Types. The following types are defined in HPIOM:

Boolean: Boolean expressions can have the value `true` or `false`.
Integer: Integer expressions can have any positive or negative value.
Enumerated types: The set of values for an enumerated type is defined statically and finite.

Arrays can be implemented by using n variables of the same type. A type "array" is also provided for convenience.

5.2.1.1.2 Abstract Grammar. HPIOM expression are defined by a simple grammar. Terminals of the grammar are variables, unknown values, signals and constants, and are composed using unary and binary, purely functional operators. An abstract grammar follows. E represents an expression, C represents a condition, **`this font`** is used for terminal symbols while *this one* is used for non-terminal symbols.

Remark: The C++ implementation uses prefixes to avoid name clashes in identifiers. `st_`, for "`st`ructure" means "a class designed to be used through a pointer". `scp_`, for "`S`ystem`C` `p`arser" is the prefix used for all identifiers in LUSSY (this should have been `lussy_`, but the code was here before the name of the tool…). Those prefixes are usually omitted in this document, for clarity.

E	⟶	C	*Conditions*
	⟶	**array_index**(E, **int**)	*Access to an element of an array like* `t[i]`.
	⟶	**cond_expr**(C, E, E)	*Equivalent of* `if` *in lisp, or* `(x ? y : z)` *in C.*
	⟶	*binary_operator*	***plus**, **minus**, **mult**, etc...*
	⟶	*lvalue*	*Left operand of an assignment.*
	⟶	**signal**	*Value of a signal*
	⟶	**unknown**	*Non-deterministic value.*
	⟶	*unary_map*	*Extension of unary operator to arrays.*
	⟶	*binary_map*	*Extension of binary operator to arrays.*
	⟶	*constant*	

constant	⟶	**array_value**	*Constant of a type array, like* `[0, 42, 3]`.
	⟶	**int_constant**	*Numerical constants*
	⟶	**enum_value**	*Constant of an enumerated type.*
lvalue	⟶	**variable**	

C	⟶	*binary_bool_operator*	
	⟶	*unary_operator*	
	⟶	**bool_constant**	
	⟶	**bool_expression**(C)	*An expression of type boolean.*
	⟶	**bool_unknown**	*Non-deterministic boolean value.*
	⟶	**comparison**	$<, <=, >, =>, =, \neq$ *operators*
	⟶	**default_cond**(state)	*Condition true if and only if all conditions on other outgoing edges are false.*
	⟶	**non_deterministic_choice**(state)	*Choice between several outgoing edges of a control point.*
	⟶	*signal_present*(*signal*)	*True if the signal is present.*
	⟶	**in_state**(A, state)	*True if automaton A is in state "state".*

binary_bool_op.	$\longrightarrow$	**and**(C, C) ....
	$\longrightarrow$	**or**(C, C) ....
	$\longrightarrow$	**xor**(C, C) ....
unary_operator	$\longrightarrow$	**not**(C) ....

Most HPIOM constructs (and, or, not, ...) have obvious semantics and will not be detailed here. The constructs `non_deterministic_choice`, `in_state`, arrays, ... can be reduced to other ones. They are therefore not essential to the semantics of HPIOM and will be detailed later, in section 5.3.

5.2.1.2 Automata

An automaton $\mathcal{A}$ is reactive machine formally defined by a tuple $(Q, V, U, q_0, \mathcal{V}_0, q_f, T, M)$ where

- Q is a set of *control points*,
- V is a set of variables (variables can have an initial value which is an expression, or be uninititialized. In the later case, the initial value of the variable will be non-deterministic),
- U is a set of unknown values,
- $q_0 \in Q$ is the initial control point,
- $\mathcal{V}_0$ is the initial valuation of variables (a possibly incomplete valuation, since some variables may be uninitialized),
- $q_f \in Q$ is the final control point (it is used to define the append operator, but doesn't have any influence on the execution semantics),
- T is a set of labeled edges between control points,
- M is a set of signals, either pure (present or not) or valued (absent, or present with a value).

HPIOM automata have no "official" textual syntax (in LUSSY, automata are manipulated as an abstract data structure using C++ code). We provide a graphical representation to simplify the explanations.

For example, an automaton with control points s_1, s_2 and s_3, with edges $t_1 = (s_1, s_2, \ldots)$ and $t_2 = (s_2, s_3, \ldots)$, and with variables v_1 and v_2 is represented in Figure 5.1.

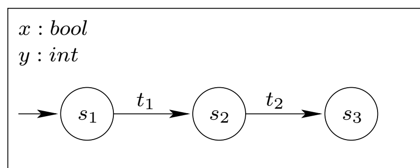


Figure 5.1: Simple HPIOM Automaton

5.2.1.2.1 Edges.

An *edge* $t \in T$ is a tuple (q_s, q_t, G, A, E) where

- q_s and q_t are the source and target control points.
- G is a condition made of elementary tests on the variables of V and the unknown values of U and tests on the presence and value of a signal.
- A is a set of parallel assignments in the form $v := e$.
- E is a list of signals emissions.

Graphically, a guard is denoted by brackets like $[x < 42]$, assignments are denoted by ":=" ("=" is used for the equality condition).

5.2.1.2.2 Communication.

Automata communicate through pure and valued signals. A signal emission is either a pure signal or a couple (*valued signal*, *value*), where *value* is an expression. A signal can be emitted only by one automaton, but can be received by any number of automata. A signal can be emitted on one or several edges. The automaton receiving the signal can either use its value in an expression, or test its presence in a condition (with the construct `signal_present`).

In practice, it is sometimes useful to have signals emitted by several automata merged together (the logical "or" for a pure signal, for example). This can be done with what we call *combinational automata*:

automata with only one state, and no variable. To perform the "or" of several pure signal, we need two edges: one emitting a signal, with a guard doing the "or" of all the signals to merge, and the default one, not emitting any signal. A generic version of this automaton is provided in the HPIOM API. For valued signals, we defined a variant of this automaton to merge the signals: the output signal will be present when one of the input signals is present, and will have the value of the present signals. The input signals should be in mutual exclusion (otherwise, the output value will be chosen non-deterministically among the present input signals). The implementation has one control point, but has one edge per input signal, plus the default edge. The guards of the non-default edges are the presence of the corresponding signal, and each edge emits the corresponding value on the output signal.

Graphically, a pure signal emission is denoted by $!$*signal*, and a valued signal emission is written $!$*signal*$($*value*$)$. For instance, the automaton of Figure 5.2, the automaton has guard $y > 5$ on edge t_2 on which it performs an assignment. The pure signal m_1 and the valued signal m_2 (with value x) are emitted on edge t_1. A condition on the presence of a signal is denoted by $?$*signal* (true if and only if *signal* is present).

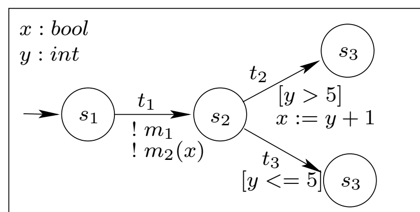


Figure 5.2: HPIOM Automaton with Signal Emission, Guard, and Assignment

5.2.2 Semantics of HPIOM

5.2.2.1 States

A *state* $s = (q, \mathcal{V})$ of such an automaton is made of a control point q and a valuation of the variables $\mathcal{V}$. The set of initial states of $\mathcal{A}$ is $\{(q_0, \mathcal{V})$ where $\mathcal{V}$ is a valuation of the variables of $\mathcal{A}$ where all initialized variables are associated with their initial value$\}$.

5.2.2.2 Transitions

The set of edges between control points implicitly define the set of transitions between states of the automaton: given a edge (q_s, q_t, G, A, E), a valuation $\mathcal{U}$ of the unknown values, the corresponding set of transitions is the set of $((q_s, \mathcal{V}_s), (q_t, \mathcal{V}_t), G', E', \mathcal{U})$ where $\mathcal{V}_s$ is a valuation of the variables of the automaton, $\mathcal{U}$ is a valuation of the unknown values of $\mathcal{A}$, and $\mathcal{V}_t$ is the valuation obtained by applying the substitution A to $\mathcal{V}_s$. G' is the condition obtained by replacing variables (resp. unknown values) of G by their value in $\mathcal{V}_s$ (resp. $\mathcal{U}$). It is therefore a Boolean condition on input signals, and E' is the set of signal emissions obtained by replacing variables by their value in $\mathcal{V}_s$ in signal values (pure signals are left unchanged). Section 5.2.2.3.3 will use this notion to define transitions for a system of automata where those conditions will be replaced by actual values of signals emitted by other automata.

Remark:	The difference between states and control points, and between transitions and edges can be somewhat confusing because the literature sometimes uses "state", or "explicit state" to refer to what we call "control point" in HPIOM, but we preferred using different words to make it clearer: intuitively, a "control point" q is what we represent by a circle when we draw the automaton. It is a compact representation of a potentially large or infinite number of states. A "state" $(q, \mathcal{V})$ contains all the information about the state of the automaton: the control point and the valuation of the variables. Likewise, "edges" link control points, are represented by arrows on the diagrams, and represent sets of "transitions", whereas "transitions" link "states". "state" and "transition" are the vocabulary of explicit state machines, while "control point" and "edges" are the vocabulary of interpreted automata.

5.2.2.3 Semantics of a System of Automata

5.2.2.3.1 HPIOM ***systems.*** A set of automata $(\mathcal{A}_1, \mathcal{A}_2, \ldots, \mathcal{A}_n)$ is called an HPIOM *system*. The product of automata in HPIOM has a synchronous semantics. This means that a transition of the global system is made of one and only one step of each component.

5.2.2.3.2 Global state. A state of the HPIOM system (called *global state*) is a tuple $(s_0, s_1, \ldots, s_n)$ = $((q_1, \mathcal{V}_1), (q_2, \mathcal{V}_2), \ldots, (q_n, \mathcal{V}_n))$ where $\forall i \in [1..n], s_i$ is a state of $\mathcal{A}_i$ (therefore, q_i is a control point and $\mathcal{V}_i$ a valuation of the variables of $\mathcal{A}_i$). The set of initial states of the system is the product of the sets of initial states of its components.

5.2.2.3.3 Global transition. A transition of the HPIOM system (called *global transition*) is a tuple $((s_0, s_1, \ldots, s_n), (s'_0, s'_1, \ldots, s'_n), (E_0, E_1, \ldots, E_n), (\mathcal{U}_0, \mathcal{U}_1, \ldots, \mathcal{U}_n))$. The set of transitions for each individual automata define the set of transition for the global system: $((s_0, s_1, \ldots, s_n), (s'_0, s'_1, \ldots, s'_n),$ $(E_0, E_1, \ldots, E_n), (\mathcal{U}_0, \mathcal{U}_1, \ldots, \mathcal{U}_n))$ (s_i and s'_i being states of $\mathcal{A}_i$, E_i sets of signals emissions, and $\mathcal{U}_i$ a valuation of the unknown values for $\mathcal{A}_i$), is a transition of the HPIOM system if and only if a tuple of conditions $(G_0, G_1, \ldots, G_n)$ exist, satisfying the following conditions for all i in $[1..n]$:

- $(s_i, s'_i, G_i, E_i, \mathcal{U}_i)$ is a transition of $\mathcal{A}_i$,
- G_i is satisfied given the signal emissions E_j from the other automata and the valuation of unknown values $\mathcal{U}_i$.

5.2.2.3.4 Validity Condition. HPIOM imposes one condition for the product of automata to be valid: If ($\exists i, j | G_i$ is a function of a signal emitted by A_j), then G_j is not a function of a signal emitted by A_i. In other words, there is no instantaneous dialog in HPIOM. Since instantaneous dialog is managed differently in the languages we want to use as back-end (in particular, it is not allowed at all in LUSTRE), this can considerably simplify the code generation. This is often referred as the "combinational loop", or the "causality" problem [Ber00].

We generally avoid designing an intermediate formalism based on particularities of the back-end, and prefer a back-end independent format, but it would not be reasonable to allow a construct that can not be translated into the potential target languages.

5.2.2.3.5 Global execution. An execution trace of an HPIOM system is a sequence of global transitions:

$$((\mathbf{S}_0, \mathbf{S}_1, \mathbf{E}_1, \mathbf{U}_1), (\mathbf{S}_1, \mathbf{S}_2, \mathbf{E}_2, \mathbf{U}_2), \ldots, (\mathbf{S}_{n-1}, \mathbf{S}_n, \mathbf{E}_n, \mathbf{U}_n))$$

where

- $\forall i \in [1..n], \mathbf{E}_i = (E_0, E_1, \ldots, E_n)$ and $\mathbf{U}_i = (\mathcal{U}_0, \mathcal{U}_1, \ldots, \mathcal{U}_n)$

- $\forall i \in [0..n], \mathbf{S}_i = (s_0, s_1, \ldots, s_n)$,
- $\mathbf{S}_0$ is an initial global state of the system,
- $\forall i \in [1..n], (\mathbf{S}_{n-1}, \mathbf{S}_n, \mathbf{E}_n, \mathbf{U}_n)$ is a valid global transition for the system.

Remark: An HPIOM system is a "flat" structure, without any notion of hierarchy (automaton inside a state of another automaton) or encapsulation (also known as "hide operator": make some signals local to a set of automata). The reason for this choice is that we never felt the need for such constructs. Since we use HPIOM only through an automatic generator, we can easily achieve the same result as hierarchy using loops in the generator. Encapsulation is used to allow an incremental product of automata and to avoid name clashes. Since we never build the product, and since the HPIOM API automatically makes names unique, encapsulation would not be of any use for us.
However, we plan to reuse HPIOM in other tools, as a common intermediate formalism. In this context, the encapsulation operator can be useful to optimize the construction of the product. Adding the encapsulation to HPIOM is therefore part of Verimag's plans, but will not be implemented during this Ph.D.

5.2.3 Non-determinism

Non-determinism in HPIOM can be expressed in two ways:

Implicitly, if the guard of several edges evaluate to true at the same time, then one of the enabled edge is chosen, non-deterministically.

Explicitly, by using the specific HPIOM constructs `unknown_value` and `non_deterministic_choice` (presented in section 5.3.1.2).

Note that implicit non-determinism can be reduced to explicit non-determinism. LUSSY implements this transformation to ease the task of the back-ends: LUSTRE and SMV back-ends can safely assume that implicit non-determinism is never used. Since LUSTRE is a deterministic language, the non-determinism would anyway have to be made explicit in the LUSTRE back-end.

Unknown values are used like any other HPIOM expression. They can have any type. Their semantics is that they can take any value at runtime. To simulate this behavior in a deterministic language, unknown values have to be converted into inputs of the program.

When the writer of HPIOM can ensure the set of outgoing transitions from a state is fully specified (one and only one transition enabled at a time), he can use `set_choice_safe(true)` on this state. Code generators can use this information to disable the implicit to explicit non-determinism conversion as an optimization. Note that it is safe not to call `set_choice_safe(true)` on a deterministic state, but unsafe to use it on a state with implicit non-determinism.

5.3 Additional constructs

5.3.1 Convenience constructs

Those constructs do not add any expressive power to HPIOM, but proved to be usefull to avoid systematic coding of some higher-level constructs.

They are useful for the programmer constructing HPIOM, but can also be used by back-ends to use an optimized encoding for those high-level constructs (such optimized encoding would otherwise require a pattern recognition that would be much harder to implement).

5.3.1.1 Abstracted Type

HPIOM defines the notion of *abstracted type*. An expression of type "abstracted type" can have no value. This type is used in the translation from SystemC when our tool encounters a type that could not be encoded easily in HPIOM (typically, a pointer), and that has to be abstracted away.

The rules to manipulate this type are the following:

- for an expression $E = \text{operator}(op_1, op_2, \ldots, op_n)$, if the type of E can be deduced from the types of the operands op_i whose type is known, or from the context (for example, if E is the right hand side of an assignment) and if one of the operands is of type "abstracted type", then, E has the semantics of an unknown value (its value is chosen non-deterministically). The same rule applies for right values of assignments.
- for an assignment $L := E$, if L is of type "abstracted type", then the assignment itself has to be abstracted away. Currently, only variables are supported as left values for assignment. This means the variable has to be removed from the automaton, and replaced by unknown value wherever it appeared.

5.3.1.2 Non-deterministic choice

It is possible to specify explicitly a non-deterministic choice between several edges, using the construct `non_deterministic_choice`. The construct is parametrized by a state. The semantics is that one and only one of the `non_deterministic_choice` s attached to a state is true at each instant.

The writer of HPIOM usually wants to ensure that using this construct, one and only one of the outgoing edges is enabled at a time (this can be easy to do using a default condition), to avoid mixing implicit and explicit non-determinism.

5.3.1.3 States as condition

It is sometimes useful to express conditions on the current state of another automaton. Intuitively, this could be described by "take this edge if automaton A was in control point s at the beginning of the transition". We added a specific condition `in_state(`s`)` in HPIOM to specify this case more easily.

This can be reduced to a condition on a pure signal: the automaton A would emit this pure signal on each outgoing edge from s.

5.3.1.4 Continuous Signal

As a shortcut for a valued signal emitted on each edge with the same expression as a value, HPIOM defines the construct `continuous_signal` . This is, for example, a way to export a variable to the external world.

5.3.1.5 Arrays

HPIOM allows the type "array of T of size n" for any type T and any positive integer n. Arrays are simply sets of values numbered from 0.

5.3.2 Abstract Addresses

In the SystemC models we are dealing with, the addresses are simply `int` values. If nothing special is done in the translation, addresses become ordinary variables in HPIOM, and any property related to addresses has to be transmitted to a verification tool able to deal with `int`s. However, in the source of the model, we can distinguish the addresses (with high influence over the control) and the data (with lower influence over the control). For addresses, we propose an encoding based upon the existence of *address maps*. Indeed, in TLM, the significant values of the address variables are given by the address maps used to describe the connection between components.

The address space is split into several relevant intervals (the address map of the bus, typically). We first define the set of relevant values $(x_j)_{j\in 1..n}$ (we use the notation $(X_i)_{j\in 1..n}$ to represent the tuple $(X_1, X_2, \ldots, X_n)$). The set of relevant intervals is then $\mathit{II} = (R_1, \ldots, R_n) = ([x_j, x_{j+1}[)_{j\in 1..n-1}$, as illustrated by figure 5.3.

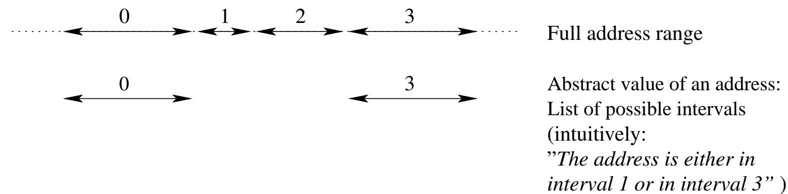


Figure 5.3: Abstract Addresses as Sets of Relevant Intervals

To represent the value of an address, we associate a Boolean variable b_i to each R_i. An address variable x is then encoded by a valuation of the vector $b_1 \ldots b_n$. A constant value $k \in R_i$ is encoded into $b_i = 1, b_{j\neq i} = 0$. As soon as we manipulate addresses, we may lose information, resulting in encodings where $\exists i \neq j | b_i = b_j = \mathit{true}$, meaning the value of x is in range R_i *or* in range R_j. The worst case is the case where all the information has been lost about an address: $\forall i, b_i = \texttt{true}$. We note $\top$ (read "top") this value.

This is conservative for safety properties. It simplifies the proofs a lot, and has proved to be sufficient on the examples we tried. Figure 5.4 illustrates the conservativeness of the abstraction. The abstraction consisting in removing completely addresses from the HPIOM system is trivial: one can replace any condition depending on an abstract address by unknown values, and remove any assignment to a variable of type address. This corresponds to the set $A \cup B \cup C \cup D$ on figure 5.4. Another abstraction, would be to compute precisely the addresses, but to replace the addresses by their Boolean encoding when they are used in a condition. In this abstraction, one and only one of the Boolean variables used in the encoding can be true at a time. This abstraction is only useful to help understanding our encoding, but it loses information without simplifying the proof. There is a loss of information only when the abstract addresses are compared to a value that is not the bound of an address range. This corresponds to the set $A \cup B$.

Our implementation of abstract addresses can be seen as an abstraction of the actual behavior, but also as a refinement of the most brutal abstraction (which would be equivalent to our abstraction with the value `true` for each interval). Any information that allows to "cut" an execution from C to D is safe and can allow to prove more properties. Any "loss" of information, moving an execution from D to C is also safe for safety properties, but may prevent the proof of some property. In other words, the value `true` for the variable representing an interval in our representation is always safe, but ensuring a `false` value allows to prove more properties.

Our goal will therefore be to keep as much information as we can while remaining conservative.

All this could be done without additional constructs in HPIOM, using simply `boolean` values. It greatly simplifies the translation to use a specific type and specific operators for the addresses. The expansion into boolean is documented in section 7.3.1.

The HPIOM constructs to manipulate abstract addresses are the following.

5.3.2.1 Extension of the HPIOM Expression Grammar for Abstract Addresses

The grammar for expression given above is extended as follows:

E	$\longrightarrow$	**`AA_const`**	*Abstract Address Literal*
	$\longrightarrow$	**`AA_plus_N`**(AA, E)	*An abstract address plus a constant integer*
	$\longrightarrow$	**`AA_plus_unsigned`**(AA, (+|-))	*An abstract address plus an unsigned value*

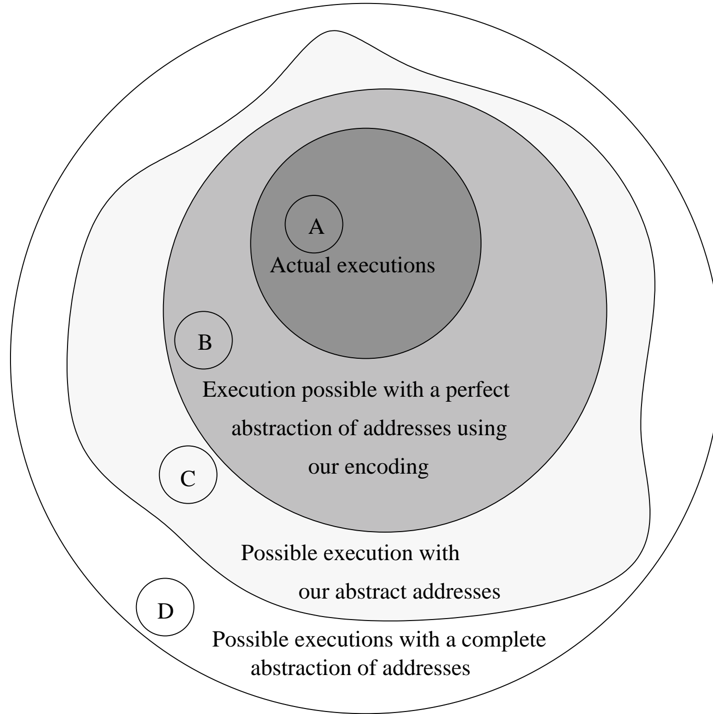


Figure 5.4: Conservative abstraction using abstract addresses

C	$\longrightarrow$	**AA_cmp_AA**(AA, AA, *operator*) .	$<$, $<=$, $>$, $=>$, $=$, $\neq$ *operators. AA means an expression of type abstract address, and the types of the operands must match.*
	$\longrightarrow$	**AA_cmp_int**(AA, N, *operator*) . .	*Comparison of an abstract address with an integer.*

5.3.2.2 Literal

The first thing we need is a representation of abstract addresses literal. They are the equivalent of boolean arrays whose size is the number of ranges in the address map. To represent a constant address in the source code, we will therefore use an encoding with one `true` value, and only one. This corresponds to case B above. We note literal abstract addresses as follows: $AA = (b_i)_{i\in[0,n]} = (b_1, b_2, \ldots, b_n)$. They are defined using the constructor `AA_const`.

5.3.2.3 Binary operators

Addresses can be manipulated, compared to or added with integers. The universal solution would be to completely abstract those constructs (return an unknown value, or $\top$ if the result must be an address). This solution will be used if no other solution is available, but our HPIOM implementation provides several constructs to encode those operations in a more refined way:

Addition/Subtraction of Constant The encoding is the following:

$$\texttt{AA_plus_N}\left((b_i)_{i\in[0,n]}, N\right) = \left(\bigvee \{b_j | \exists n \in [x_j, x_{j+1}[\text{ with } (n+N) \in [x_i, x_{i+1}[\}\right)_{i\in[0,n]}$$

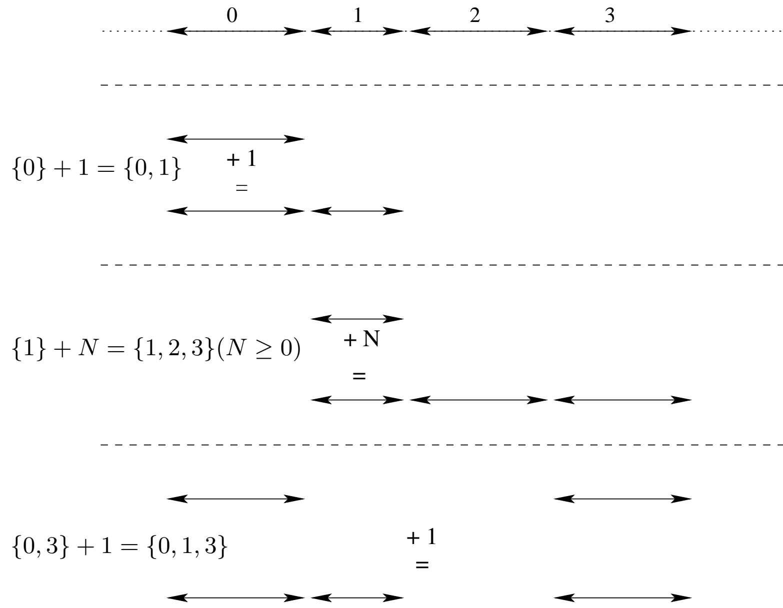


Figure 5.5: Examples of Abstract Address Operations

Intuitively, if AA can be in interval $[x_j, x_{j+1}[$, then AA_plus_N(AA, N) can be in $[x_j + N, x_{j+1} + N[$, and therefore in $[x_i, x_{i+1}[$ if its intersection with the later is non-empty.

Note that this operation can only be applied to constants. Constants means either a literal in the program, or any expression for which the value can be computed statically. This operator is currently applied only in the case of literal constant, but any kind of sophisticated static analysis technique could be added to use this construct in other cases (in Figure 5.4, this would mean cutting some behaviors from C to D).

Addition/Subtraction of Unsigned Value: When a non-constant value is added to an abstract address, and when the sign of this value is known statically, the result is the following:

$$\texttt{AA_plus_unsigned}\left((b_i)_{i\in[0,n]}, > 0\right) = \left(\bigvee_{j \geq i} b_j\right)_{i\in[0,n]}$$

$$\texttt{AA_plus_unsigned}\left((b_i)_{i\in[0,n]}, < 0\right) = \left(\bigvee_{j \leq i} b_j\right)_{i\in[0,n]}$$

Addition/Subtraction of Arbitrary Value: In the case of arbitrary value, we can't do better than returning $\top$.

5.3.2.4 Comparisons

There are two kinds of comparisons involving abstract addresses: Comparison of addresses, and comparison between address and concrete value. We give here the examples of $>$ and $=$. The other traditional comparison operators ($\neq$, $\leq$, $<$ and $\geq$) are documented in the code.

Remark:	In any case, the value of a condition depending on an abstract address will be an abstract condition, which means it can evaluate to `true` when executing the HPIOM system even if the concrete condition (involving concrete addresses) is false. The question answered to by those constructs is: "Is there a value for which the concrete condition is true?".

5.3.2.4.1 Comparison of Addresses. In this section, we use the notations $AA = (b_i)_{i\in[1..n]}$ and $AA_j = ((b_j)_i)_{i\in[1..n]}$.

Equality: Two abstract addresses can be equal if and only if they have at least one interval in common:

$$\texttt{AA_cmp_AA}(AA_1, AA_2, EQ) = \bigvee_i \{(b_1)_i \wedge (b_2)_i\}$$

Greater Than: AA_1 can be greater than AA_2 if one of the interval of AA_1 contains one value greater than a value of an interval of AA_2:

$$\texttt{AA_cmp_AA}(AA_1, AA_2, GT) = \bigvee_i \left\{ \bigvee_j \{(b_{1i} \wedge b_{2j}) \wedge \exists a \in [x_j, x_{j+1}[, b \in [x_i, x_{i+1}[|a > b\} \right\}$$

5.3.2.4.2 Comparison With a Concrete Value.

Equality: An abstract address can be equal to an integer value if the boolean variable associated to the interval containing this value is true:

$$\texttt{AA_cmp_int}(AA, N, EQ) = \bigvee_i \{b_i \wedge N \in [x_i, x_{i+1}[\}$$

Greater Than: An abstract address can be greater than an integer value if this integer value is greater than one of the value of this interval, i.e. if it is smaller than the biggest possible value in this interval (the upper bound minus one since we work with integer values):

$$\texttt{AA_cmp_int}(AA, N, GT) = \bigvee_i \{b_i \wedge N < x_{i+1}\}$$

When the concrete value is a constant, those expressions can be simplified (they involve comparisons of constants that can easily be computed statically). We didn't need to do anything special about those simplifications in the implementation, since a trivial general constant folding algorithm did it, and more.

5.4 Expression of Properties in HPIOM

We have defined the execution semantics for an HPIOM system. Since our goal is to verify properties on it, we also need a way to specify properties for the system. We could have used a temporal logic such as CTL or LTL [Var01] as a specification language, but we have chosen a simpler alternative: since we're only interested in safety properties, and since HPIOM has synchronous semantics, it is well known that any safety property can be encoded as a synchronous observer, and thus reduced to a reachability property. Reachability of an edge and reachability of a control point are clearly equivalent from the expressiveness point of view.

Our choice is to specify a set of pure signals that must never be emitted: An HPIOM *property* is a set of pure signals emitted on some edges of the components of the system.

An HPIOM property $(m_1, m_2, \ldots, m_k)$ (where each m_i is a pure signal) is said to be satisfied for an execution $((\mathbf{S}_0, \mathbf{S}_1, \mathbf{E}_1, \mathbf{U}_1), (\mathbf{S}_1, \mathbf{S}_2, \mathbf{E}_2, \mathbf{U}_2), \ldots, (\mathbf{S}_{n-1}, \mathbf{S}_n, \mathbf{E}_n, \mathbf{U}_n))$ if and only if $\forall i \in [1..n], \forall j \in [1..k], m_j \notin \mathbf{E}_i$.

An HPIOM property is said to be satisfied on an HPIOM system if it is satisfied for all possible executions of the system.

5.5 Implementation of and Manipulation of HPIOM

This section presents the data structures we use to represent an HPIOM system. We also present the tools and methods available to manipulate this data-structure. The most important uses of the HPIOM manipulation API are the optimizations and the back-ends (LUSTRE and SMV code generators). They are introduced and described technically here, but the algorithms will be detailed in the next chapters.

HPIOM expressions are trees, and follow exactly the “Composite Tree” design pattern [GHJV94], and is manipulated with the “Visitor” design pattern. The following chapters recall briefly the definition of those design patterns and how they are applied to HPIOM. HPIOM automata are the natural extension of this pattern, but do not have a tree structure.

5.5.0.5 Other Possible Abstractions for Addresses

The encoding we are using is relatively simple. It is efficient (it uses only Boolean variables) and has the big advantage of being applicable without modifying the proof engine. It is only an intermediate construct, and the back-end will see only the Boolean encoding. It is, on the other hand, not extremely expressive. We have no way to distinguish two addresses that are in the same address range.

We could use abstract interpretation in the prover itself to lose less information. NBAC can already use an abstract domain based on polyhedra, but this is too costly and does not scale up. An interesting work has been carried out by Mathias Peron, master student in Verimag on an abstract domain tailored for address manipulation [Pér05]. It is an extension of the intervals domain, with in addition the notion of equality and inequality.

5.5.1 HPIOM Expressions and the Composite Design Pattern

5.5.1.1 Principle of the Composite Tree Pattern

The composite design pattern allows one to treat composite and primitive objects the same way. A base class is common to composite objects and “leafs” (non-composite objects). In HPIOM expressions, the composite has a tree structure, with one class per kind of tree node. The grammar rules for the tree correspond to the inheritance diagram for the classes. Figure 5.6 shows an example of a class hierarchy following the composite tree pattern.

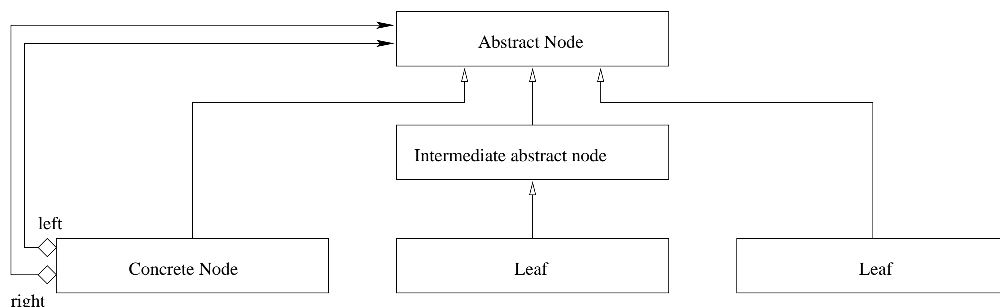


Figure 5.6: Class Hierarchy for a Composite Tree

This pattern is easy to implement and pleasant to use for the programmer: the type of children in function of each node is provided explicitly through the class hierarchy and relies on the host language (C++) typing mechanism (both static and dynamic). The main benefit will come later, with the use of the Visitor's design pattern as an iterator for this structure.

5.5.1.2 HPIOM Expressions Classes

HPIOM expression's class hierarchy defines the abstract base class `st_scp_expression`, and the intermediate abstract class `st_scp_condition`. Figures 5.7, shows the inheritance diagram for expressions. Figures 5.8 and 5.9 show the inheritance diagram for conditions (the hierarchy is truncated for clarity. The full diagram is available in LUSSY's documentation). Those diagrams are automatically generated from the source using doxygen[vH]. They use an UML-like syntax.

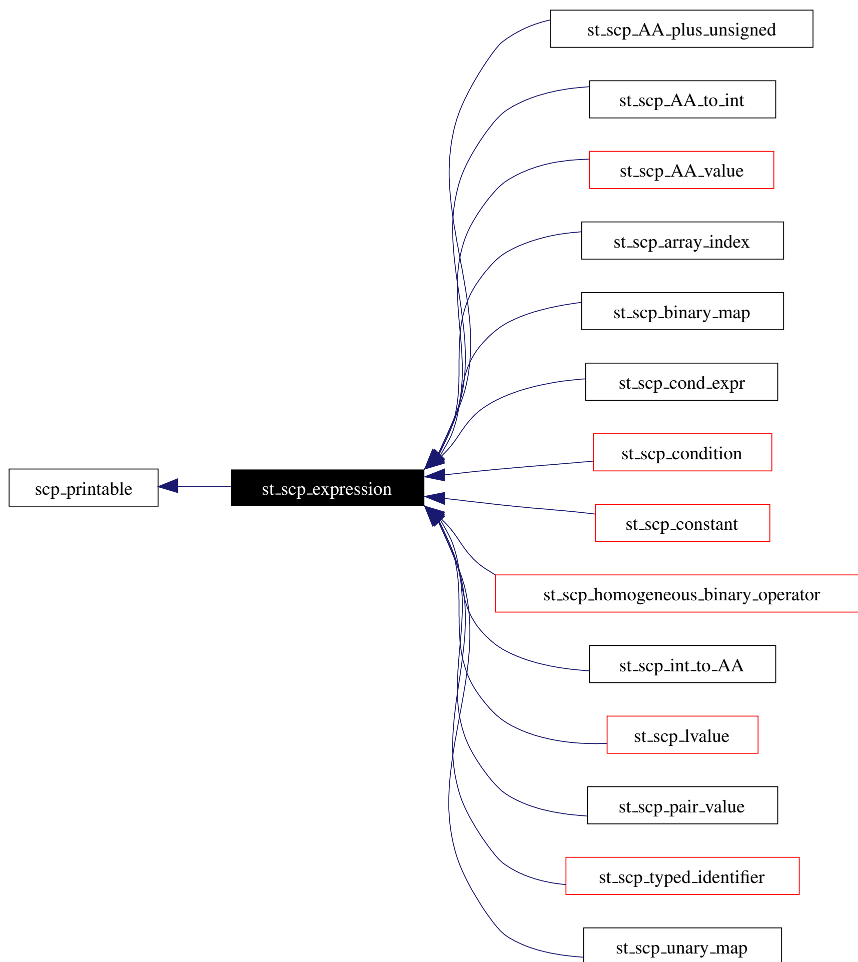


Figure 5.7: Class Hierarchy for Expressions

5.5.2 HPIOM Expressions and the Visitor Design Pattern

5.5.2.1 The "Visitor" Pattern

The "Visitor" is often cited as a typical example of a design pattern. It corresponds exactly to our use-case for expressions, and extends rather naturally to automata.

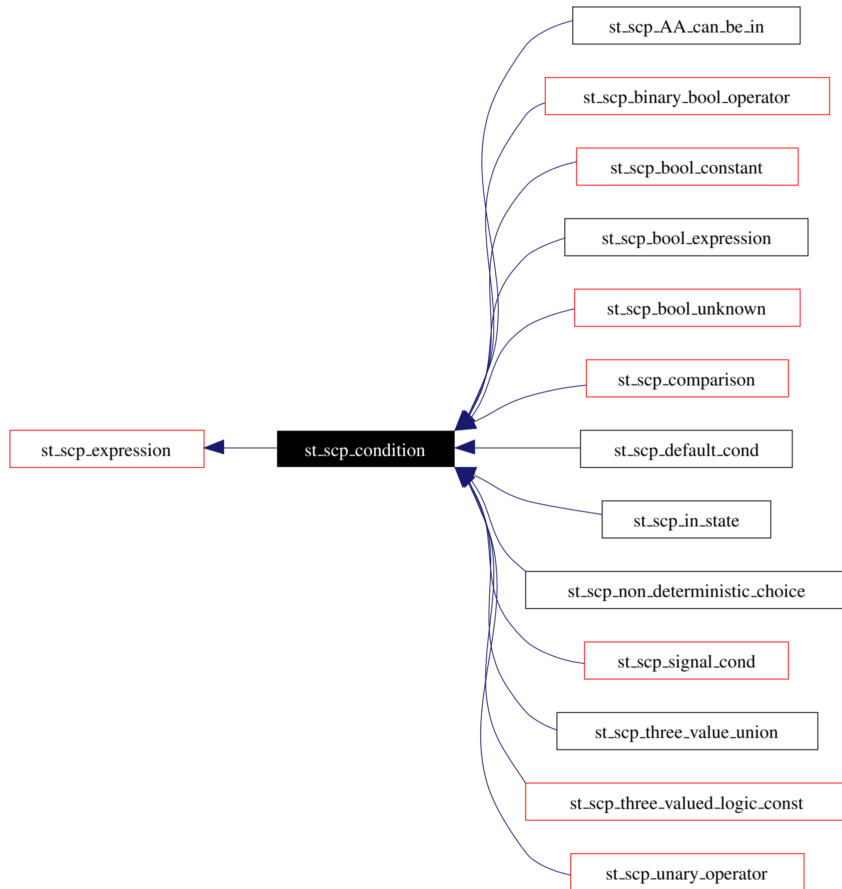


Figure 5.8: Class Hierarchy for Conditions

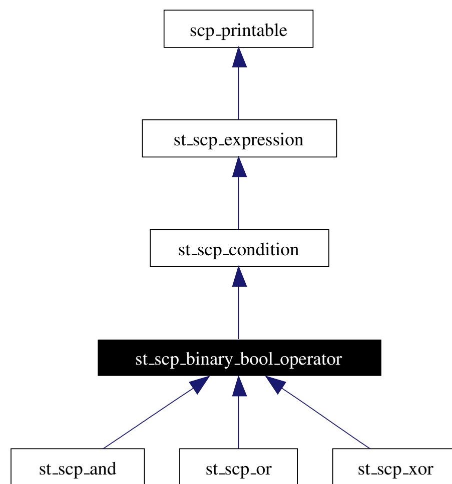


Figure 5.9: Class Hierarchy for Binary Boolean Operators

5.5.2.1.1 *Problem solved.* The problem to solve is to implement a recursive run on the tree structure. A function is called on the top of the tree, an action is chosen depending on the type of node (using the “Composite” pattern, this means depending on the dynamic type of the object), and this action will itself call this function on its children.

The LUSTRE back-end is a typical example of such an algorithm. Printing the HPIOM expression `plus_operator`$(+, 2,$ `mult_operator`$(*, x, 3))$ which represents the expression $2 + (x * 3)$ consists in calling a function on the node $+$, that will execute

return ”(” + *recursive_call(left_operand)* + ” + ” + *recursive_call(right_operand)* + ”)”;

The first recursive call will print the constant 2 while the second will display $(x * 3)$ in a way similar to the algorithm for the $+$ operator. The result will be the string `(2 + (x * 3))`.

5.5.2.1.2 *Solution Without the Visitor Pattern.* The natural solution to invoke a different action depending on the dynamic type of an object in C++ is to use virtual functions. To implement the LUSTRE back-end, we could have a method `to_lustre` in each HPIOM class. The short algorithm presented above would be the body of the `to_lustre` method for the `binary_operator` class. The *recursive_call* would be recursive calls to the `to_lustre` methods for the operands of the operator.

This solution is simple to write and simple to understand, but spreads the code of the operation `to_lustre` in all the HPIOM classes. It also has the drawback of being intrusive in the HPIOM code, which means it’s not applicable for someone having only read access to the HPIOM library.

5.5.2.1.3 *Principle of the Visitor.* The Visitor solves this problem by grouping all the methods in a single class. This class, the “Visitor”, depends on the HPIOM classes, but HPIOM itself has no dependencies on its visitors. A class diagram is provided in Figure 5.10.

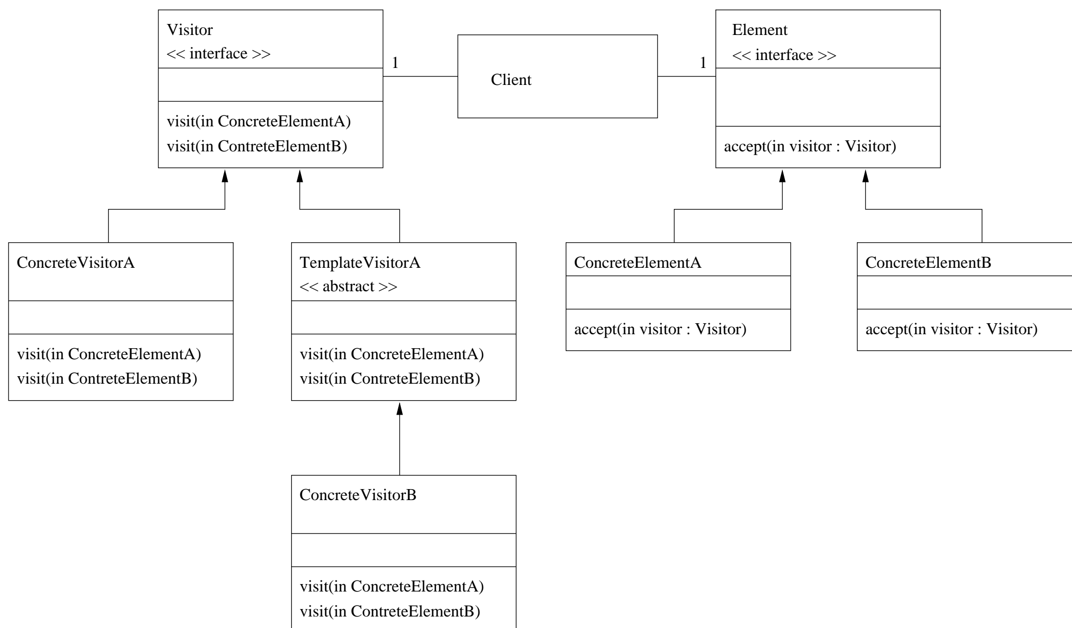


Figure 5.10: “Visitor” Design Pattern

The problem is that the choice of the method to execute can not directly rely on C++ virtual method, since the polymorphism of virtual methods only applies to the object on which the method is called (the visitor, in our case), and not to its arguments. The trick used by the visitor to simulate polymorphism based on the argument is to have a method `accept` in each class to visit, which simply calls the method `visit` on its argument:

```
void accept(visitor v) {
    v->visit(this); // polymorphism takes place here
}
```

So, the code generating LUSTRE code for the binary operators will look like this

```
string to_lustre_visitor::visit(plus_operator p) {
    return "("
        + p->get_left_operand()->accept(this)
        + " " + s + " "
        + p->get_right_operand()->accept(this)
        + ")";
}
```

One big advantage of this solution is that since each operation is implemented in a class, it is possible to use the object-oriented functionalities of C++ to factorize the code: we can define a hierarchy of visitors, with base classes providing default implementations for the concrete visitors. Once the base classes are written, it becomes possible to write a visitor with only a few lines of code.

5.5.2.2 Hierarchy of visitors

We implemented several types of visitors for HPIOM.

5.5.2.2.1 Expression Visitors. `expression_visitor` (Figure 5.11) is limited to expressions. It provides a default implementation for each HPIOM class, which does a recursive run on the expression, doing nothing. A visitor to count the number of times `true` appears in an expression for example, could be written simply by deriving from `expression_visitor` and overloading `expression_visitor::visit(scp_true)` to increment a counter.

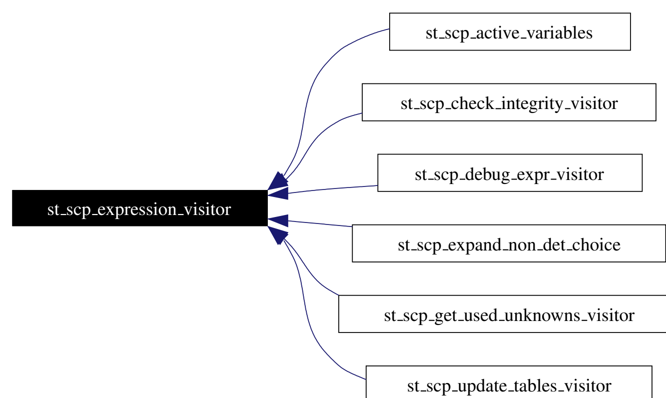


Figure 5.11: Class Hierarchy for Expressions Visitor

We implemented 6 such visitors:

active_variables: Derives from `expression_visitor` but also implements automata-wide algorithms. Performs active variables detection and optimization. This will be detailed in section 7.3.3.1.

check_integrity_visitor: Performs several integrity checks (no null pointers, source and target control points for an edge belong to the correct automaton, no shared variables between automata, ...) on an automaton. It is used for defensive programming, to help finding bugs, but has no influence on the functionality.

debug_expr_visitor: Used to print expressions as a tree. This is used only for debugging and is designed to be called from GDB.

expand_non_det_choice: Transforms non-deterministic choices between several edges into a choice depending on unknown values. This will be detailed in section 7.3.2.

get_used_unknowns_visitor: Counts the number of unknown values used on each edge, to further optimize this number. This will be detailed in section 7.3.3.2.

update_tables_visitor: Updates the variables and unknown tables for an automaton: the list of unknown and variables used by an automaton is kept in the data-structure representing this automaton. Most operations keep this list consistent, but in some cases, the operations will not. This visitor restores consistent variables and unknown tables.

5.5.2.2.2 Transformation Visitors. `transform_visitor` allows the easy implementation of recursive functional-style transformations for HPIOM. Each `visit` method returns an object of the same type as its operand, or a base class of it. The default implementation for each node applies the following idea:

```
visit(type object) {
    o1, o2, ... := childs of object;
    o'1, o'2, ... := recursive call of the visitor for o1, o2, ...
    if (o1, o2, ... == o'1, o'2, ...) {
        // nothing changed
        return o;
    } else {
        return new type(o'1, o'2, ...);
    }
}
```

A visitor inheriting from `transform_visitor` and not overloading any methods will therefore perform a recursive run of the structure, without changing anything. A visitor to change all the `foo` into `bar` in the automata can be written simply by overloading the `visit` method for the `foo` object to:

```
visit(foo f) {
    return new bar();
}
```

Figure 5.12 presents the transformation visitors we have implemented up to now.

process_AA: Adds some required conversion nodes for assignments between abstract addresses and integer values. It only overrides the visit method for the `assignment` object.

expand_AA: Transforms abstract addresses into arrays of boolean, as explained in section 7.3.1.

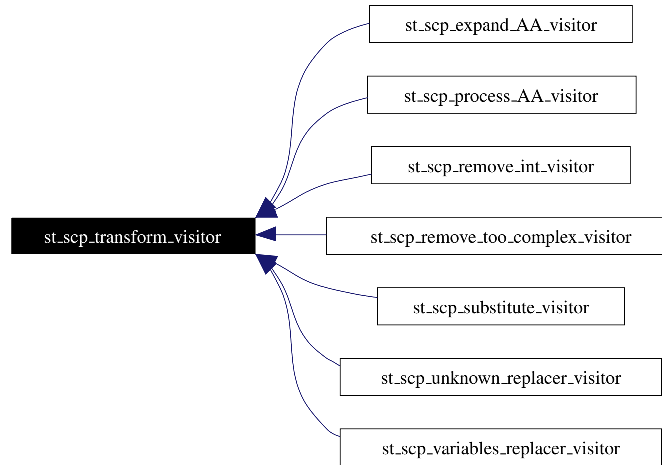


Figure 5.12: Class Hierarchy for Transform Visitor

remove_int_visitor: Abstracts away integer values. The resulting program is a pure boolean program, where conditions depending on integer values are replaced by non-determinism.

remove_too_complex_visitor: Transforms expressions depending on "abstracted type" into non-deterministic values.

substitute_visitor: Replaces all occurrences of a variable by an expression.

unknown_replacer: Used to minimize the number of unknown values in an automaton.

variable_replacer: Replaces each variable by its optimized value during live variable detection and optimization algorithm.

5.5.2.2.3 Printable Visitors. The LUSTRE and SMV back-ends are implemented using visitors (Figure 5.13). This will be detailed in sections 8.3 and 8.4. Each `visit` method returns a string. For historical reasons, `to_lustre_visitor` is a base class of `to_smv_visitor`, but the common code should have been factored in a separate base class.

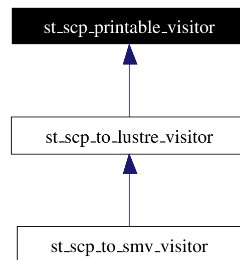


Figure 5.13: Class Hierarchy for Printable Visitor

5.5.3 On-the-fly Optimizations

Some optimizations on HPIOM are performed after building the complete system (typically, with the visitors presented above). Other are simple enough to be performed on-the-fly: instead of optimizing the structure, we avoid generating the unoptimized version.

They are implemented using factory functions instead of constructors for some HPIOM nodes. Instead of creating the new object blindly, the factory function can reuse an existing node, or generate a semantically equivalent construct, and call the constructor only when needed.

5.5.3.1 Constant Folding

It is often the case, in particular in systematically generated code, to have expression that could be evaluated statically. For example, $x * (40 + 2)$ could be written $x * 42$.

In HPIOM, the constructors of most operators are replaced by factory functions that can optionally check if the operands are constant, and return a constant instead of calling the constructor if it is the case. For example, the constructor of the `or` node is replaced by

```
scp_condition st_scp_or::build_or(scp_condition left, scp_condition right){
    if (!scp_flags::optimize_bool) {
        return new st_scp_or(left, right);
    } else if (left->is_true()) {
        return left;
    } else if (left->is_false()) {
        return right;
    } else if (right->is_true()) {
        return right;
    } else if (right->is_false()) {
        return left;
    } else {
        return new st_scp_or(left, right);
    }
}
```

5.5.3.2 Node Sharing

Allocating and initializing many nodes with completely equivalent semantics would be costly in terms of time and memory. LUSSY is able to share some nodes, such as the boolean constants `true` and `false`, which use the singleton pattern to ensure the uniqueness of the node. The constructor is replaced by the following factory function:

```
static scp_true build_true() {
    if (m_instance == NULL) m_instance = new st_scp_true();
    return m_instance;
}
```

5.5.4 Future improvements

5.5.4.1 Memory management

The current HPIOM implementation use both heap allocated objects (using `new` operator) and node sharing. A reasonable `delete` operator would therefore need reference counting. Currently, the tree nodes are never deallocated. This is not a very serious issue since LUSSY is a transformational program, not a reactive program, so, the memory will anyway be freed at the end of execution of the program, but it's not good programming practice anyway[1]. A better implementation should use smart reference-counted pointer such as `boost::shared_ptr<>`.

5.5.4.2 Namespace, modularity

LUSSY clearly distinguishes the front-end (PINAPA) from the rest of the code. However, the components of the back-end (HPIOM structure, BISE, BIRTH, LUSTRE and SMV back-ends) are not separated clearly enough. Each component should be implemented in its own namespace, with clear (and no unneeded) dependencies between the components.

In the future, we would like to make a better separation between the HPIOM API and the rest of LUSSY, to allow other tools to use it either as input or output format. We would like to give it a textual syntax (probably a semi user-readable like XML will be the right choice).

[1]The reader may guess I'm an ex Java programmer by examining my coding style . . .

Chapter 6

BISE: Semantics of SystemC and TLM Constructs in Terms of Automata

Contents

6.1 Introduction **91**
- 6.1.1 Principles 92
- 6.1.2 Expressiveness of HPIOM, Abstractions, and Limitations of BISE 93

6.2 Semantics of process code into HPIOM **93**
- 6.2.1 Simple Operations and Sequence 93
- 6.2.2 Control Structures 94

6.3 Semantics of the Synchronization Primitives and the Scheduler **95**
- 6.3.1 The SystemC Scheduling Mechanism 95
- 6.3.2 Model for the `sc_event` 98

6.4 Communication Mechanisms **98**
- 6.4.1 SystemC Signal 98
- 6.4.2 Direct Semantics of TLM Constructs 99
- 6.4.3 Unmanaged SystemC Constructs 105

6.5 Encoding Properties **105**
- 6.5.1 Assertions 105
- 6.5.2 Multiple `write` on a Signal During the Same δ-cycle 105
- 6.5.3 Process Termination 106
- 6.5.4 Mutual exclusion 106
- 6.5.5 Concurrent TLM Port Accesses 107

6.6 Related Work **107**
- 6.6.1 Other Formal Semantics for SystemC 107
- 6.6.2 SC2PROM 107
- 6.6.3 SystemC Translation Into Signal 107

6.7 Conclusion **108**

6.1 Introduction

We have presented the intermediate representation HPIOM, and our implementation of it. This chapter will illustrate the transformation of a SystemC program, parsed by PINAPA, into this intermediate representation. This is implemented in the component BISE of the tool LUSSY.

6.1.1 Principles

Producing HPIOM from SystemC source files is a way of giving SystemC a formal reference semantics. By translating HPIOM into an executable formal language like LUSTRE, this reference semantics becomes executable.

We could give an *abstract* semantics to SystemC designs, and obtain an intermediate representation simpler and easier to analyze than HPIOM, but losing some potentially important information. Our choice for the LUSSY toolbox is the opposite: the translation into HPIOM does not perform more abstractions than those implied by the expressiveness of HPIOM compared to that of SystemC (see section 6.1.2). Since most interesting properties are undecidable on HPIOM, further abstractions will have to be made, but we let them to specific verification tools connected to HPIOM. Some abstraction that could hardly be carried out by the prover (like the encoding with abstract addresses presented in 5.3.2) are implemented in LUSSY itself, but are kept optional.

The translation of SystemC into HPIOM poses the same problems as the one we had for PINAPA (in section 4.3): what is the difference between SystemC and C++? What is the “static” part of the program, and what is the “dynamic” part? Any tool manipulating C++ code statically and formally (like Polyspace [Deu03]) already has a C++ front-end and translator to a formally-defined representation like HPIOM. Such tools solve only part of our needs for BISE: first, they do not take parallelism into account (the SystemC scheduler contains assembly code — or optionally calls to POSIX threads in the version 2.1 — to manage processes, and no static analysis tool will be able to deal with general assembly code manipulating the program counter and stack pointer). Then, they would be very inefficient to model other SystemC kernel primitives.

For those reasons, BISE does more than a standard C++ code analyzer. It has its own model of SystemC, and never parses the SystemC library source code itself. Instead, we describe HPIOM patterns, based on the SystemC library specifications: there is an automaton pattern for the scheduler, one for each signal, etc. To generate instances of these patterns, we need the information extracted by PINAPA from the SystemC design: for instance, the automaton for the scheduler depends on the number of processes in the system, automata for channels depend on the number of connected modules, etc.

We could also translate SystemC processes taking the scheduler and the synchronization primitives into account, but not the TLM constructs, which would then be treated as ordinary C++ code. This would lead us to lose interesting information about the structure and behavior of the design. We have chosen to take TLM constructs into account during the translation, which means giving a direct HPIOM semantics to TLM constructs.

An important question is the one of the accuracy of our semantics regarding SystemC. Since the programs to be analyzed are themselves models of the reality, with some approximations and some abstractions, we could try to provide another model of the same reality, with different modeling choices. For example, SystemC has a non-preemptive scheduling policy, which will not be true anymore in the implementation of the chip. We could have chosen to model a preemptive scheduling in our semantic extraction. This policy would be a give a super-set of the possible behaviors of SystemC, so, proving a non-preemptive model correct for some safety properties gives a proof that the system itself is correct for the same properties. On the other hand, it could exhibit some bugs that can not be found by simulation. For example, if a program does an access to a shared resource, then it will most probably need a mutual exclusion mechanism, and access it only inside a critical section. But with a non-preemptive scheduler, any portion of code separated by `wait` statements is a critical section. So, it is easy to write a program correct with respect to SystemC semantics, but without a synchronization mechanism that would be necessary in the real chip. The question is whether such “bug” can be considered interesting. One of the principles of TLM is to model the architecture and the global behavior of a system, but not its micro-architecture, and the bugs we could find are precisely the ones in the micro-architecture, since the processes are supposed to de-synchronize (use a `wait` statement) on each inter-module communication. The majority of the bugs found with this method would therefore be irrelevant both for the SystemC model (the faulty behavior will not appear with any implementation of SystemC) and for the lower-level implementation (at this granularity, the code would be completely different anyway). Our transformation does therefore take into account the non-preemptiveness of the SystemC scheduler.

Of course, since SystemC has no official formal semantics, a formal proof of the equivalence between

a SystemC source file and the corresponding HPIOM representation built by BISE is impossible. HPIOM being executable means executions can be compared, but it is also of great importance to give a semantics to SystemC into HPIOM in a simple, well structured and clearly decomposed manner, which we describe here. This leaves room for optimizations.

The main idea of the algorithm implemented in BISE is the following:

1. Each process in SystemC will be associated with one automaton in HPIOM representing its control flow, plus one representing its state in the scheduler,
2. The scheduler itself is modeled with one automaton, which communicates with the automata for the control flow and the state in the scheduler of each process to manage processes eligibility and execution.
3. the complete HPIOM description of a SystemC design will be made of all these "process" automata, plus specific automata for SystemC and TLM constructs.

6.1.2 Expressiveness of HPIOM, Abstractions, and Limitations of BISE

Syntactically speaking, LUSSY accepts a very large subset of SystemC, being based on a full C++ front-end. The limitations related to the front-end have been discussed already in section 4.3.2. The other restrictions are of semantic nature and appear in BISE : the SystemC code is accepted by LUSSY, but the semantics is made abstract because the target formalism is less expressive than a general-purpose language. This occurs for all pieces of code that deal with pointers and dynamic data structures.

HPIOM may be used to encode any statically bounded-memory program. In SystemC, static bounds are guaranteed if: 1) the program does not perform dynamic memory allocation; 2) there are no calls to recursive functions (which is acceptable in the context of embedded systems).

The transformation of SystemC into HPIOM abstracts memory allocation primitives and recursive function calls into new input signals with unknown value. We also do that for not-yet-implemented constructs of SystemC, to get a working connection to verification tools before full SystemC has been taken into account by the front-end. These abstractions are clearly conservative for safety properties: the set of behaviours a SystemC code may exhibit when considering two complex expressions is a superset of the set of behaviours it can exhibit when considering the detail of these expressions.

Another abstraction (which is optional) is related to the way addresses are dealt with. The encoding used for this abstraction was described in section 5.3.2. The abstraction consists in using this encoding for any variable of type address.

The last abstraction is related to asynchrony. We are using SystemC to model and simulate *asynchronous* components. Although it provides a construct `wait (t)` where `t` is an amount of time, TLM guidelines specify that this quantitative time `t` should not be used to enforce synchronization (i.e., the designer should not assume that two processes that perform the same `wait (t)` will synchronize when `t` has elapsed). Our translation into HPIOM provides an abstraction of the timing which enforces this guideline, as detailed below (6.3.1.3).

6.2 Semantics of process code into HPIOM

Compiling imperative code into automata is a well known problem and there is no semantic difficulty here. However, the abstract syntax tree for a C++ contains a lot of particular cases, and a lot of them have to be taken into account if we want to apply our tool to real-world SystemC designs. Hence this part of the translation represents a significant part of the work (the translation itself represents around 1,500 lines of C++, and use some operators that have been defined in the HPIOM library for the task).

6.2.1 Simple Operations and Sequence

Simple operations like assignments have a direct equivalent in HPIOM, and can be executed in one transition. We implemented the transformation of the sequence ("`;`" in C++) as follows: the automaton corresponding to the process being transformed always has a final control point. This final control point has no particular semantic for HPIOM, but represents the exit point for the piece of code already translated. Then,

the code to be transformed in sequence will use this final control point as initial control point (initially, the automaton has only one control point which is both initial and final).

Concretely, we defined a macro `CREATE_NEW_FINAL_STATE` that declares and creates a new final control point and an edge from the former initial control point to this control point. The operations that need only sequencing in the automaton (no branch, no loop) use only this macro to create control points and edges.

6.2.2 Control Structures

The translation for the `if` statement and the `while` loop are given in figure 6.2 and 6.1 as an example.

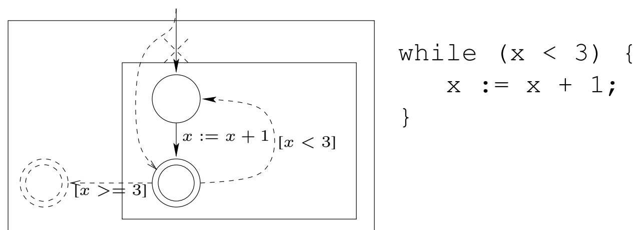


Figure 6.1: Control flow for a while loop

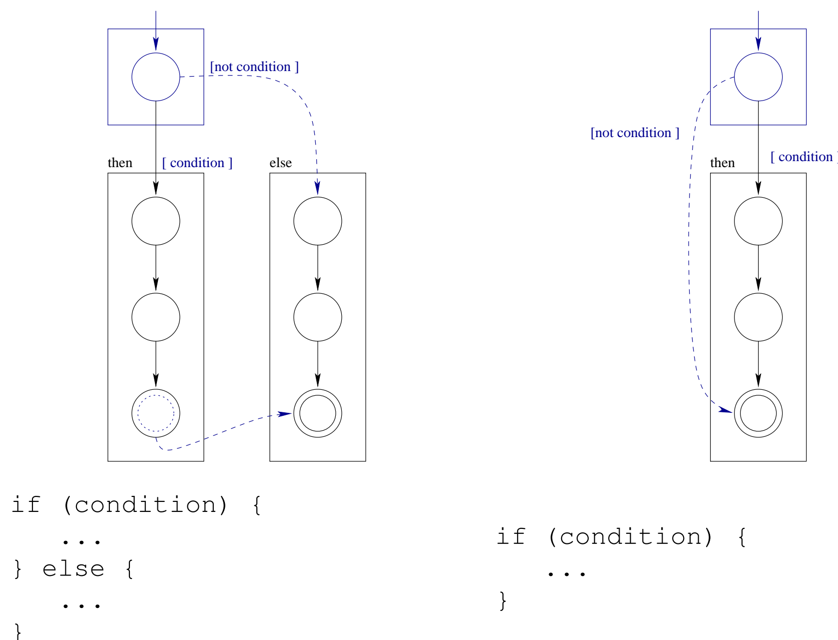


Figure 6.2: Control flow for a while loop

We have a model for the main other C++ constructs: all the operators on primitive types (some of them, like the bitwise operators are abstracted away because they have no equivalent in HPIOM), assignments, `for` and `do while` loops, `switch/case` statements, etc . . . We currently abstract all standard function calls. The best way to manage them would be to inline them at the AST level when possible, and to abstract them otherwise.

In order to accept general SystemC designs before having implemented all the details of C++ code, some constructs are currently abstracted away. For instance, the expressions that are currently too complex to be managed by LUSSY are replaced by inputs of the system. For verification purposes, this is equivalent to considering they have a non-deterministic value, and it is a conservative abstraction. Note that this

approach assumes that the complex expression has no side-effect. This assumption currently has to be verified manually, but this should be automated (in a pessimistic way) in the future. These abstractions are only temporary and we could give exact translations with more work on the tool.

6.3 Semantics of the Synchronization Primitives and the Scheduler

6.3.1 The SystemC Scheduling Mechanism

Expressing the semantics of the scheduler by some synchronizations between the HPIOM automata may be done in several ways. The global communication scheme is shown in figure 6.3 and will be detailed below. To get the highest confidence in the correctness of the translation algorithm, we choose a straightforward translation that may not be the most efficient one and leaves room for optimizations.

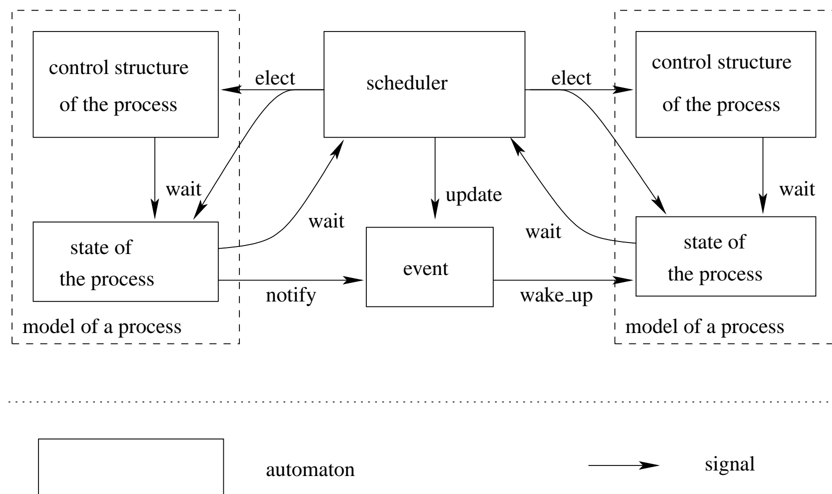


Figure 6.3: Global view of the communication between the automata in HPIOM

The semantics of the SystemC scheduling policy is modeled by one automaton for the scheduler, plus two per process. The fist one represents its control structure (as explained above), and the other one represents its state in the scheduler.

6.3.1.1 State of an `SC_THREAD` in the Scheduler

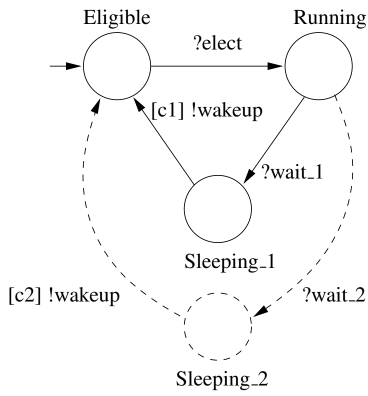
Figure 6.4 shows how the automaton is generated for an `SC_THREAD`. The process may be either running, ready to run (eligible), or sleeping. The synchronization between this automaton and the control structure automaton is such that the last may change control point only if the former is in control point “running”.

6.3.1.2 State of an `SC_METHOD` in the scheduler

The model for an `SC_METHOD` is a slightly different: the wake-up condition is not chosen when leaving the “running” control point (`wait` statement or equivalent), but while the process is running (by a `next_trigger()` statement): by default, the static sensitivity list will be used, and each call to `next_trigger()` will update the condition. We need therefore to have several sub-states for the “running” control point. This is illustrated by Figure 6.5.

6.3.1.3 The SystemC Scheduler

The scheduler itself is represented by an additional automaton (Figure 6.6). It starts in a control point “selecting_process”. At that moment, all the processes are eligible. The SystemC official definition lets the choice between the eligible processes unspecified. In our model, the scheduler chooses one process



Synchronizations:
elect: received from the scheduler when the process is chosen,
wakeup: sent to the control structure,
wait_2: received from the control structure when a `wait` statement is reached,
c1 and **c2** correspond to the conditions the process is waiting for in the corresponding "sleeping" control point.

Figure 6.4: State of an `SC_THREAD`

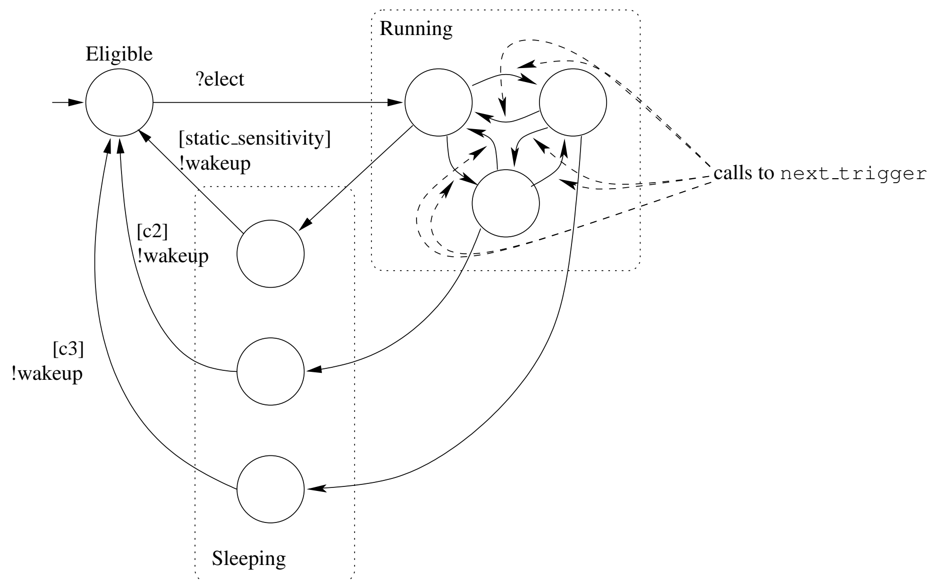
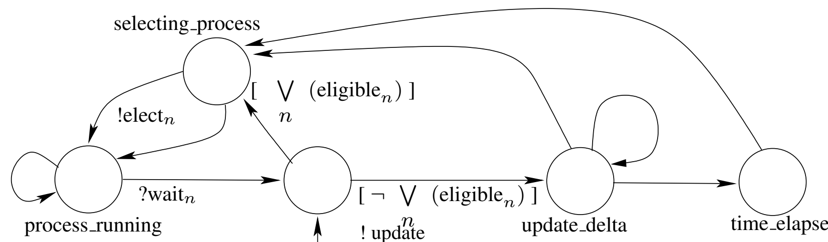


Figure 6.5: State of an `SC_METHOD`

non-deterministically. The election corresponds to the edges emitting `elect_n` on the figure. Then the scheduler runs the elected process: in the automaton representing the state of the process, this means taking the edge from "eligible" to "run". When the process has finished its execution (go to "sleep" control point), the scheduler selects another one, and so on until there is no more process eligible. Then, the scheduler goes to the update phase.



Synchronizations:
elect_n: sent to the corresponding process state automaton and control structure
wait_n: received from the corresponding process state automaton
update: sent to all processes that may have an action to execute during the update phase

Figure 6.6: Pattern of the SystemC scheduler.

Our model of the SystemC scheduler is non-deterministic in two ways. First, one of the transitions emitting `elect_n` is chosen non-deterministically when several processes are eligible. Actually, we had three options when we took this design decision:

1. Model precisely the behavior of the reference implementation,
2. Choose a deterministic behavior, and model it,
3. Implement a non-deterministic choice.

The first solution is the most complex to implement. The OSCI scheduler is optimized for speed, and contains, among others, a heap and two FIFOs. Modeling this in HPIOM would not be easy. The second option is much easier to implement. We can chose any priority policy and apply it to the choice of a transition. The last one is quite obvious since HPIOM supports non-deterministic choices.

Regarding verification, the last possibility is the only one to be correct. The first may miss bugs that would appear with a different simulator, or when a completely unrelated process is added to the platform, and the second one may both miss bugs with the reference scheduler and provide false counter-examples.

The only argument in favor of the two first ones is the fact that they limit the state explosion problem by making the system more deterministic, so it may help the proof engine. However, an exact model of the scheduler would have been complex, so the resulting automaton would have less global states, but its transition function would also have been more complex.

We have therefore chosen the last one. When we prove a property of a SystemC design including this non-deterministic scheduler, we prove it for any possible implementation.

This may seem to be a detail, but it is actually very important for the reliability of the model. A scheduler dependency can occur as soon as two processes share a resource such as an event or a variable, which happens when two processes in the same module access the same data-member, or in the TAC and BASIC channels (from the architecture point of view, TAC and BASIC channels are separate components, but the communications occurs with function calls, so the code is executed in the context of the master module). Such bug did happen in practice in the code developed by the team in STMicroelectronics. A simpler approach to solve this problem is to replace the unspecified scheduler choice with a randomized choice. Claude Helmstetter, Ph.D student is working on a more efficient run-time approach to detect scheduler dependencies.

The second source of non-determinism comes from our management of the time. In the case of `wait`

statement with an amount of time as argument, the "time-elapse" phase of the scheduler algorithm (Figure 2.7) awakes the processes in the order specified by the `wait` parameters. In our translation, they are woken-up non-deterministically (encoding non-determinism with oracles). This means that the HPIOM model has more behaviors than what the SystemC interpreter may exhibit. This conservative abstraction prevents the designer from relying on the amount of time provided as argument to the `wait` statement: if a safety property can be proved on the HPIOM model, then it is true that the `wait` statements have not been used to enforce synchronization. Ideally, this abstraction should be optional, and we should also provide a more accurate mode. However, this would use a numerical variable to model time, and most of the back-ends we are using would abstract it away. An accurate modeling of time plus a removal of numerical values would give the same result as our version.

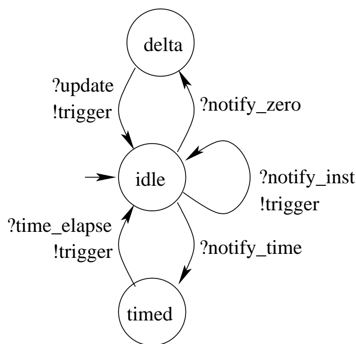
This abstraction is even more important than the non-determinism of the scheduler choice inside a δ-cycle. Being able to give some flexibility to the timed wait statements allows a better testing of the platform and software synchronization. Data-race conditions are problematic but uncommon inside a δ-cycle, but they are all the more common regarding the timed behavior: any shared resource needs to be associated with a synchronization mechanism, regardless of the communication mechanism used to access it (unless, off course, the synchronization is included in the communication itself). Testing the software on a simulator with a fuzzy notion of time can help making the software more reliable and more robust.

6.3.2 Model for the `sc_event`

The low-level synchronization primitive in SystemC is called an `sc_event`. The operations available for an `sc_event` are:

- `notify()` : the event is triggered immediately,
- `notify(SC_ZERO_TIME)` : the event will be triggered at the end of the δ-cycle,
- `notify(`*time*`)` : the event is scheduled to be triggered at some date in the future.

We also build one HPIOM automaton for each `sc_event`, according to the pattern of figure 6.7. It has one initial control point plus one control point for each kind of delayed notification. The immediate notification is modeled by a single edge. In any case, the edge going back to the initial control point is the one triggering the event. It emits a signal that will move processes waiting for that event from the "sleeping" control point to the "eligible" control point.



Synchronizations:
notify_... signals are received from the control structure when a `notify` statement is encountered,
trigger goes to the process state automaton, and will make the condition to the "eligible" control point true,
update and **time_elapse** both come from the scheduler.

Figure 6.7: Pattern for an sc_event

6.4 Communication Mechanisms

6.4.1 SystemC Signal

The simplest communication mean in SystemC is the `sc_signal`. It is possible to read, write, or wait on a port connected to a signal. This is more complex than it seems to be, because a write operation on a signal is not taken into account instantaneously: an internal variable `next_value` is updated inside the signal, and the actual value (the one accessed by the `read` method) is updated only at the end of the

δ-cycle (when the signal receives `update`). At that moment, if the next value is different from the current value, an event is triggered.

This is illustrated by figure 6.8, which represents a Boolean signal. `write(T)` comes from the automaton encoding the body of the process, and `update` comes from the scheduler. The automaton starts in one of the two control points where `next_value` and `current_value` are equal. When receiving a `write`, the automaton changes its control point to reflect the new value of `next_value`, without changing `current_value`. At the end of the delta cycle, the automaton receives the `update` signal. If it is in a control point where `next_value` is different from `current_value`, then a signal is emitted to wake up processes that would be waiting for the event at this moment.

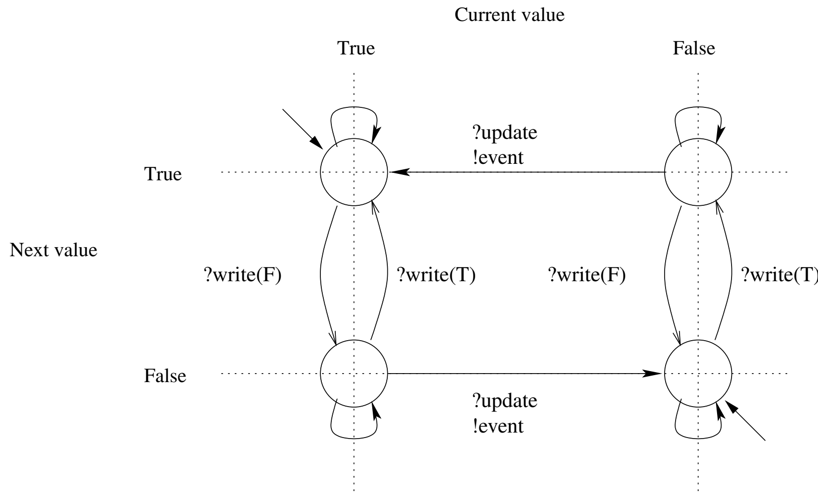


Figure 6.8: Pattern of a SystemC boolean signal

Remark:	The automaton is actually not modeled with four control points, but with one control point and two Boolean variables. HPIOM would not have allowed having two initial control points for one automaton.

6.4.2 Direct Semantics of TLM Constructs

We have seen earlier that TLM constructs needed a special treatment and could not be translated automatically like process code in an efficient way. We currently have a models for one channel with the BASIC protocol, and one with the TAC protocol.

6.4.2.1 Model of the BASIC Protocol

The first protocol we modeled was the BASIC protocol. It is a simple protocol used mainly as an example of what can be done with the TLM interface, but is usually not used in real platforms. It was nevertheless a good exercise to begin with.

Our prototype currently manages the basic protocol in a slightly modified version: the channel used is a variant of the `basic_router` in which we replaced the arbitration policy by a non-deterministic serialization of the transactions. This section presents the idea for the creation of the automata modeling a `basic_seq`, but is not meant to be complete.

The processing of a transaction is divided into several phases, carried out by several well synchronized automata. Most of the time, only one automaton changes its state at a time, and the other ones are blocked in a loop.

Remark:	One of the differences between multi-threaded imperative code like SystemC and automata like HPIOM is that in SystemC, the control flow can move from one component to another with function calls, while HPIOM's control flow is local to an automaton. This strong synchronization simulates a control flow jumping from one automaton to another. This is possible only when two threads never execute the same piece of code at the same time.

6.4.2.1.1 Wait for the channel to be available. For each port, we create an automaton in which the process will wait until the channel becomes available (Figure 6.9). When the transaction starts, a test is done to see if the channel is available. If not, go to a "waiting" control point, to let other processes execute. The process will become eligible again when the channel selects this transaction.

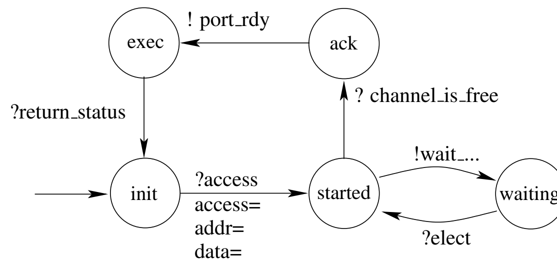


Figure 6.9: Wait for channel availability

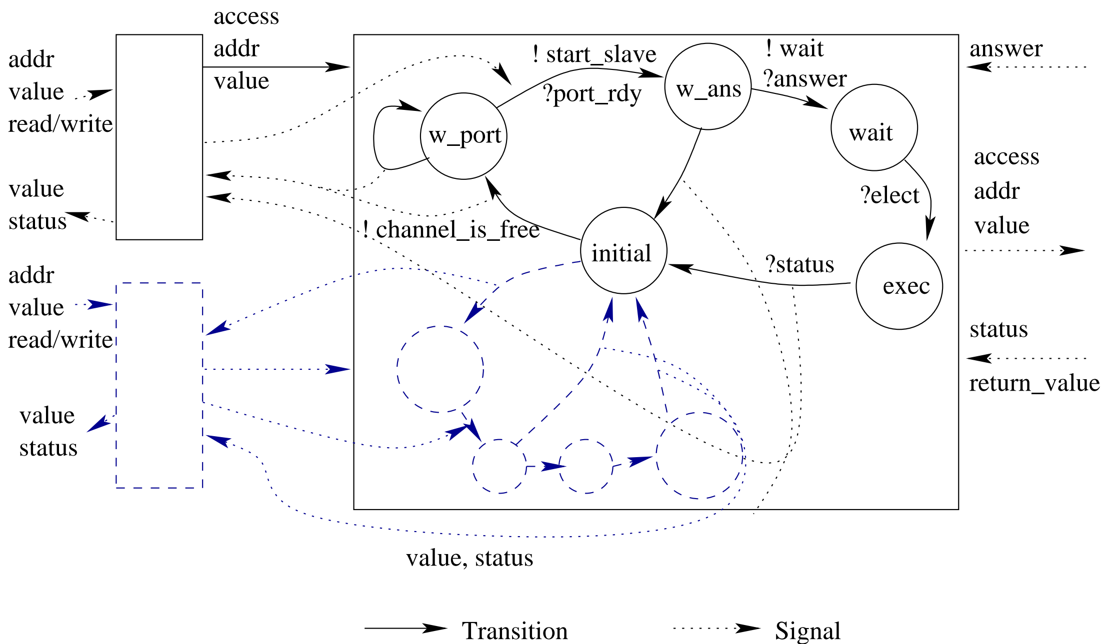


Figure 6.10: Pattern for a basic_seq

6.4.2.1.2 Select transaction and resolve the address. All these automata are connected to another one that loops in the initial control point until it receives a transaction (Figure 6.10 gives some hints on the

structure of this automaton). When some transactions are to be executed, this automaton chooses non-deterministically one of them and go to control point "w_port". A signal is sent to all the slaves, and those whose address map matches answer. If the channel gets no answer, then, it returns immediately, with a status `is_no_response`.

6.4.2.1.3 Simulate a `wait` to allow other processes to execute. If a slave module answered, then it simulates a `wait` statement on time. (control point "wait" on figure 6.10) This is included in the protocol to allow other processes to execute (which is necessary because the scheduler is not preemptive).

6.4.2.1.4 Execute the corresponding method in the slave module. At the end, we have to execute the body of the corresponding method in the slave module, and return the status (control point "exec" on figure 6.10). Note that the schema is a bit simplified here, since the channel has to communicate with several instances of slave modules.

6.4.2.2 Model of the TAC protocol

The TAC protocol, unlike BASIC, was design to be used in production. It is therefore both essential to support it in LUSSY and harder to implement. We implemented a model for the `tac_seq` (unlike the model of the `basic_seq`, we considered the official semantics of the component, and did not modify it to make it easier to model).

The first difference between TAC and BASIC is that TAC channels are templates on the type of addresses and data used on the channel. This has been a source of difficulty in the front-end, PINAPA, but BISE will anyway consider this template arguments to be of type integers (in practice, the TAC components have anyway been tested only with integer types as argument, but the difference between two instantiations is their size: `short int`, `int` or `long int`, mainly). For addresses, we will have to encode them with booleans to be able to prove anything, and for the data, HPIOM will work with mathematical integers ($]-\infty, +\infty[$), which means we will not detect integer overflows (but this is not in the scope of LUSSY anyway), but will get the correct semantics as long as there is no such overflow.

The global mechanism is similar to the one of the BASIC protocol. The following details the differences step by step.

6.4.2.2.1 Wait for the Channel to be Available. The automata for the master ports are very similar to the ones for the BASIC ports. A model of a master port is given in Figure 6.11.

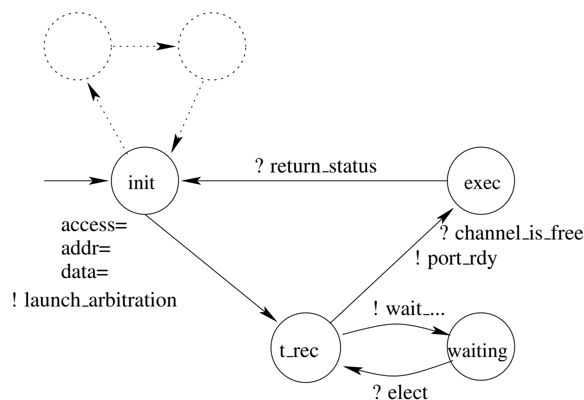


Figure 6.11: Wait Loop for a TAC seq

The difference is that, to be faithful to the semantics, we modeled a real FIFO in HPIOM. It is an n-cells FIFO, where n is the number of processes able to write on the channel. Each cell is a 2-control points automaton, with one control point "busy" and one control point "free". The data itself is stored in variables

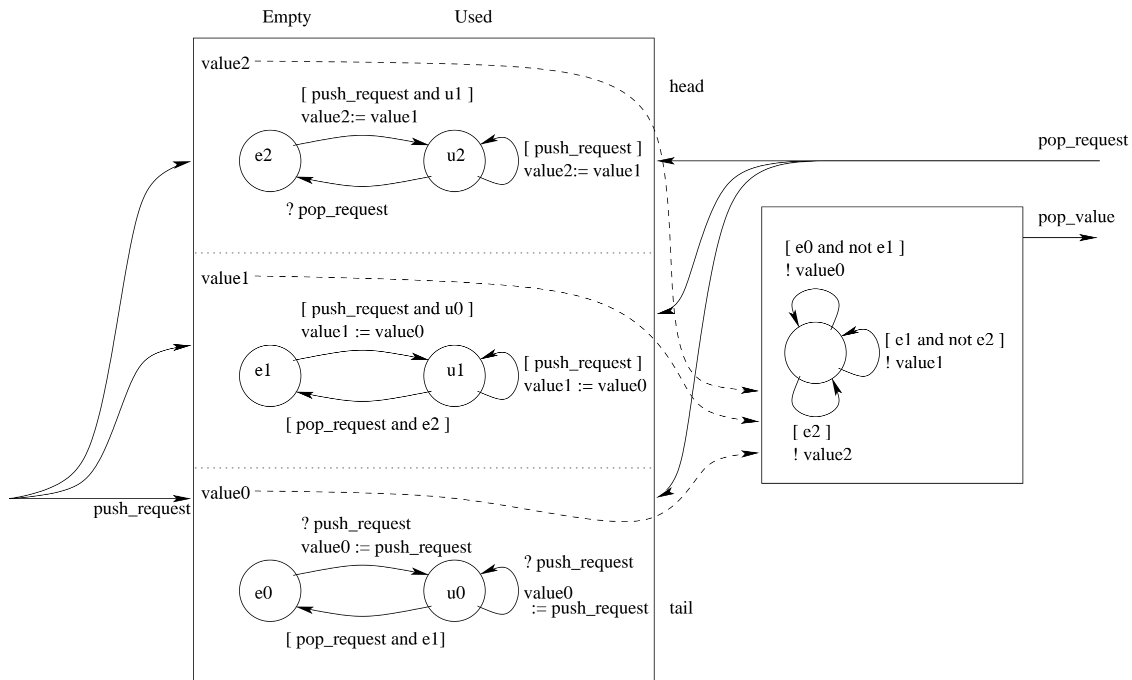


Figure 6.12: Encoding of a FIFO in HPIOM

of the automaton. When a transaction arrives, a signal is sent to all the cells. The empty cell whose previous neighbor is occupied receives the data. When a pop request is relieved (when the channel starts processing the transaction), the occupied cell whose next neighbor is empty is freed. The "head" value is computed by a separate combinational automaton which emits the value of the occupied cell followed by an empty cell. Figure 6.12 illustrates this construction.

In practice, since one signal can not be emitted by several modules in HPIOM, we have to merge the signals coming from the different master ports with another combinational automaton. The resulting data-flow is shown in Figure 6.13.

6.4.2.2.2 Automaton for the Channel. The automaton for the channel itself is again similar to the BASIC seq. It has one loop for each process able to access the channel. The steps in this loop are the same except that the `wait` statement is executed *after* the processing of the transaction and not *before*. See

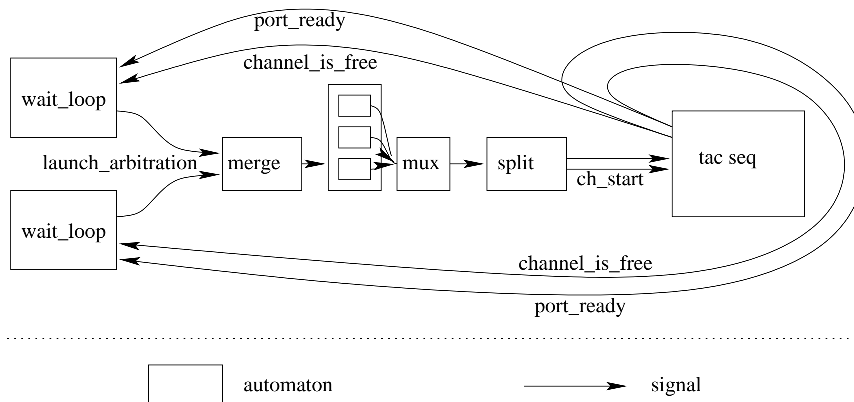


Figure 6.13: Data-flow for the Model of a FIFO in HPIOM

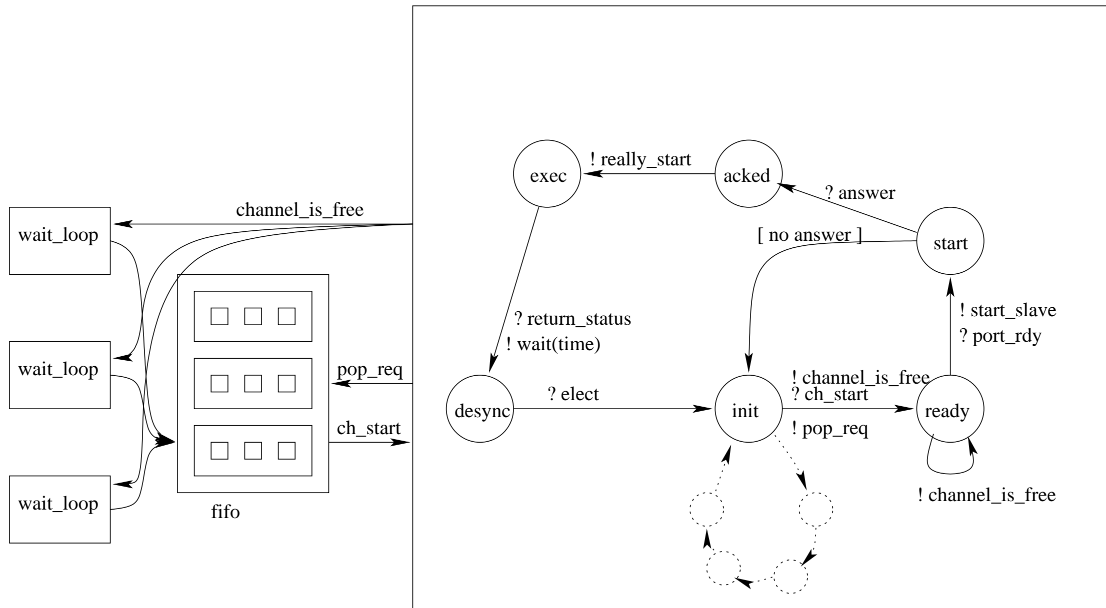


Figure 6.14: Model for the TAC seq

Figure 6.14 for details.

The channel waits for a transaction, pops it from the FIFO, sends a signal to the slave and waits for the answer. If no slave answered, it returns an exit status "no response", otherwise, it starts the processing of the transaction, waits for its completion, and simulates a wait statement to let other processes be executed.

One of the difficulties here is to know which process is running to execute the `wait` statements correctly. In the SystemC implementation, this corresponds to the context of simulation, which is global for the simulation, and independent of the position of the control flow. In HPIOM, an automaton other than a process automaton (for instance, the automaton for the channel) doesn't know which process it is running. The `wait` signal emitted after processing the transaction, for example, is different depending on the process that launched this transaction. In this case, it is easy since we have a different loop for each process able to use the channel. The `wait` signal emitted in each loop will therefore not be the same. In the processing of the slave method, a `wait` statement is forbidden by coding guidelines and would anyway lead to undefined behavior with the current implementation of TAC. We assume that this guideline is verified (it could easily be checked with a lint tool).

6.4.2.2.3 Entry point for the Slave Module. The transaction will be executed in the slave module, but the communication from the channel to the slave is not direct. We create an intermediate automaton that will receive the transaction and call either the automaton of the read method or the one of the write method. The automaton is represented in Figure 6.15.

Remark:	This intermediate automaton has a role similar to what is done in the C++ implementation in the method `tac_prim_slave::transport` : receive a transaction and call the appropriate slave method depending on the type of transaction.

6.4.2.2.4 Processing of the Transaction in the Slave Module. For the TAC slave module, we build one automaton for each "slave method" (`ReadAccess`, `WriteAccess`, ...). The automaton is mainly the automaton for the body of those methods (as explained in 6.2), encapsulated in a loop (Figure 6.16). The

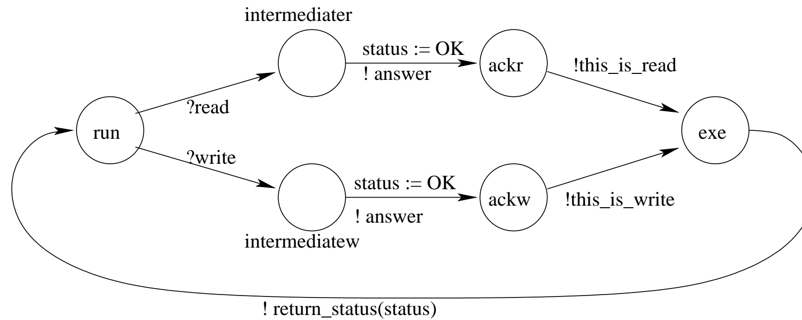


Figure 6.15: Intermediate Between the Channel and the Slave Methods

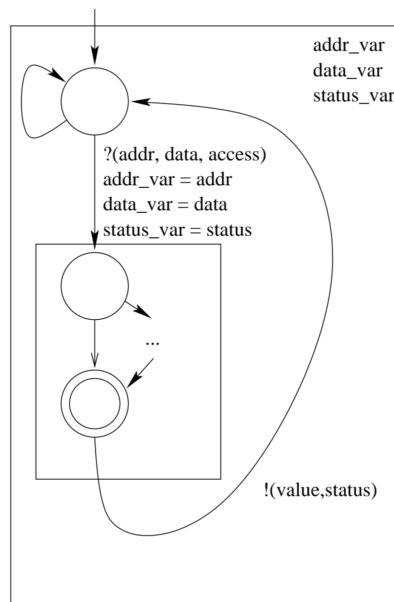


Figure 6.16: HPIOM Model for the Slave Method

first transition is triggered by the intermediate automaton presented above, and the automaton returns to its initial control point at the end of execution. Note that this assumes once again that the method does not contain any `wait` statement.

6.4.2.2.5 Return to the Master. When the slave method finishes executing, each automaton will return the status and data to the previous automaton in the chain from the master to the slave. The master will eventually receive this signal and continue its execution.

6.4.2.3 Applicability of the Method to Other Protocols

The TAC protocol is actually a relatively simple protocol. It contains the necessary to model all the functional aspects of the communication in the platform, but since it is not meant to be time accurate, it doesn't model many of the low-level aspects of a real bus or network-on-a-chip protocol.

For example, the TAC protocol is a blocking protocol in the sense that the master will be blocked in the call to the `read` or `write` statement until the transaction processing is over. The performance of the silicon implementation of this kind of protocol would of course be unacceptable in many cases. This is why protocols like the ST-BUS (a network-on-a-chip protocol developed by STMicroelectronics) have non-blocking communication protocols. Concretely, the transaction is split into a request and a response.

A non-blocking transaction is therefore equivalent to two blocking transactions, and the approach is not so different.

Another limitation of our approach is that is does not allow the presence of `wait` statements in the slave methods, since only one process can be executing the slave method at a time. This could be worked around by using n automata for the slave method, where n would be the maximum number of process being able to execute this portion of code (i.e. the number of processes accessing the channel).

6.4.3 Unmanaged SystemC Constructs

Some SystemC constructs did not appear on the examples on which we tried LUSSY. This is the case for `sc_fifo` and `sc_mutex` for example. We did not implement a model for those constructs, but this would be relatively easy to do so (we already have a model of FIFO queue as part of the TAC seq).

6.5 Encoding Properties

Section 3.3.1 gave a list of the kind of properties that LUSSY is able to verify. We can verify generic or semi-generic properties and C++ assertions in the source code. All of them are safety properties.

We reduce the verification of each of these properties to the reachability of edges in HPIOM (actually, the emission of a pure signal or a list of pure signals, as presented in 5.4). We add the notion of “error” to the semantics of SystemC. The question given to the proof engine will be the formal translation of “Is it possible that an error occurs”, and the translation from SystemC to HPIOM will modify the semantics of SystemC to change legal actions that violate the property into “an error occurs”. The property is therefore involved in the component BISE of the tool chain.

The following sections details the encoding of each property. All of them are optional, the user can select the kind of properties to check with command-line arguments for LUSSY.

6.5.1 Assertions

Assertions are at the same time the simplest and the most expressive way to specify properties. The user can use the macro `ASSERT` to check that a particular expression is true. During simulation, the `ASSERT` macro will raise a runtime error if the argument is not true, so, providing assertions in the source code enables both runtime and static verification (any good programmer should anyway use this kind of defensive programming for runtime checking). Technically, the `ASSERT` macro is defined by:

```
#define ASSERT(X) if(!(X)){is_this_reachable();}
```

so the problem of assertion verification is reduced to the problem of code reachability. The translation into HPIOM of the `if` statement happens as usual, and `is_this_reachable()` is compiled into a two-control points automaton with one edge emitting an error signal.

6.5.2 Multiple `write` on a Signal During the Same δ-cycle

To check that an `sc_signal` is never written on twice in a δ-cycle, we turn the semantics of `sc_signal::write`, which were originally “When two processes or more write on the signal during the same δ-cycle, the last value is taken into account” into “When two processes or more write on the signal during the same δ-cycle, raise an error”.

In practice, we add a variable `already_written` to the automaton modeling the signal. This variable is set to true when the signal is written on, and to false at the beginning of the δ-cycle. If it is true when the signal receives a `write`, then raise an error. Figure 6.17 gives an idea of the resulting automaton (although it is not 100% accurate, since the explicit control points on the picture are actually modeled with variables).

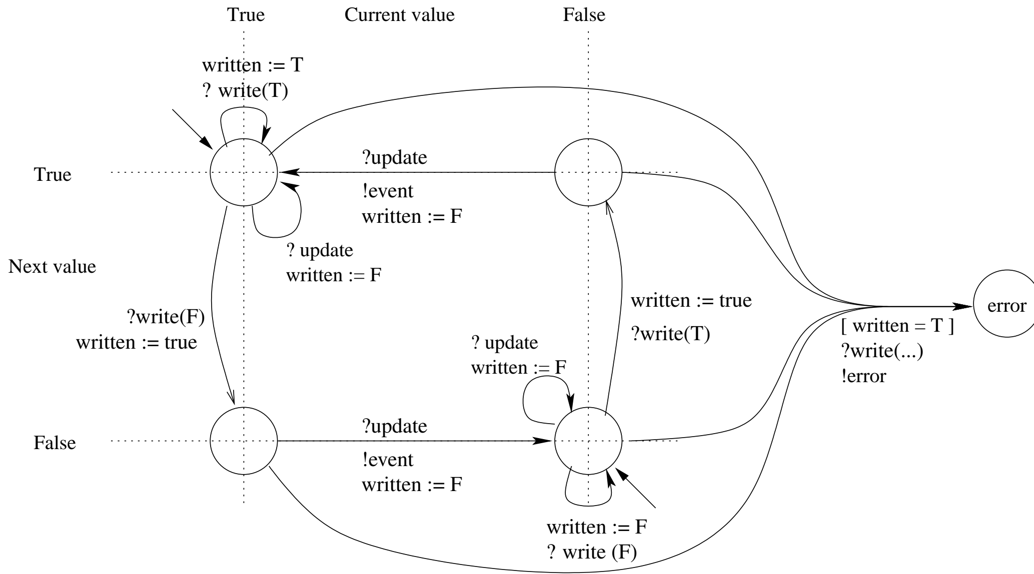


Figure 6.17: Modified Semantics for an `sc_signal`

6.5.3 Process Termination

Checking that an `SC_THREAD` doesn't terminate is relatively easy. We can simply add one edge emitting an error signal in the automaton for the control flow of the process, at the end of the execution of this process. This is equivalent to adding `ASSERT(false)` at the end of the process.

6.5.4 Mutual exclusion

Mutual exclusion is not a completely generic property. The user has to specify the portion of code he wants to be in mutual exclusion. Of course, since SystemC is not preemptive, the property will be trivially true if those portions of code do not contain any `wait` statement. Concretely, we provide the user two functions `scp_begin_critical_section` and `scp_end_critical_section` that can be inserted in the platform's code. LUSSY will check that a call to `scp_begin_critical_section` is always followed by a call to `scp_end_critical_section` in the same process before the next `scp_begin_critical_section`.

BISE adds an automaton to the system that will act as a synchronous observer [HLR93]: it will receive signals from the processes when a `scp_begin_critical_section` or a `scp_end_critical_section` is executed, and raise an error when the signals appear in the wrong order. The automaton for the observer is provided in Figure 6.18.

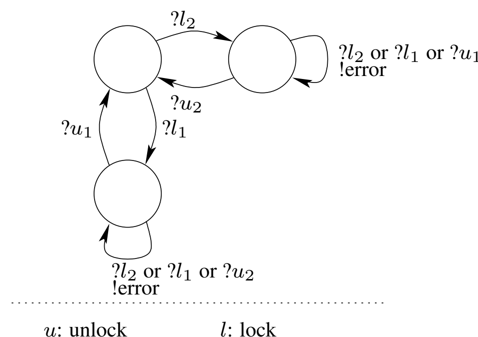


Figure 6.18: Automaton for the Mutex Observer

6.5.5 Concurrent TLM Port Accesses

In the current implementation of the TAC master port does not protect against concurrent accesses, and lead to undefined behavior when this is the case. An interesting generic property that LUSSY could check would be the mutual exclusion of the accesses to this port. This has not yet been implemented in LUSSY, but the implementation would be rather straightforward.

In the automaton of Figure 6.11 page 101, there is one separate loop for each process accessing the port. We would just have to add an error signal whenever the automaton receives an access signal if the automaton is not in its initial control point. This can be implemented either in the automaton itself, or in a separate observer.

6.6 Related Work

6.6.1 Other Formal Semantics for SystemC

We have seen earlier that implementing BISE is a way to specify the formal semantics of SystemC. This is not the first time SystemC is given a semantics: in [HGR+01], the semantics is expressed in terms of Abstract State Machines. It only models SystemC 1.0 which was the only available at that time, and has an inaccurate model of the scheduler: in their model, the processes are executed in parallel whereas the language reference manual explicitly says *"The scheduler is not preemptive. An application can assume that a method process will execute in its entirety without interruption, and a thread or clocked thread process will execute the code between two consecutive calls to function wait without interruption."*. Perhaps this was not the case with SystemC 1.0, but the specifications do not seem to be available anymore today. [Sal03] does the same with another formalism (denotational semantics). It models only a very strict subset of SystemC, with explicit limitation both on the kind of processes and on the content of processes (only `wait` statements and assignments are modeled, without a notion of control-flow). None of these papers provide an actual translation from SystemC to their formalism.

The difference with the work presented here is that BISE is not only a theoretical concept. Is as an actual tool, generating an executable format. This helps getting confident in the semantics we are proposing (we can compare the concrete executions), and allows automatic further treatments such as model-checking, as will be detailed below.

6.6.2 SC2PROM

The approach followed by LUSSY lead to a similar work in INRIA Rhône Alpes, in co-operation with STMicroelectronics: The idea is to apply LUSSY's approach with a compositional verification tool called PROMETHEUS [Goe01]. The tool, SC2PROM, reuses our front-end, PINAPA, and generates its own intermediate format, which is based on an asynchronous formalism with priorities. This approach should allow splitting large proofs into smaller ones, and therefore allow performing larger proofs.

6.6.3 SystemC Translation Into Signal

[TGSG05] describes a general approach to extract a behavioral type from a system using the polychronous model of computation (using Signal [BGJ89] as a support language). This applies to languages and execution models such as Java, SystemC and SpecC. Their starting point is the GNU SSA (static single assignment) [Pro04] form, which is the intermediate representation in GCC, since version 4.0. Compared to the AST structure that we are using, SSA is much simpler, and independent of the source language (complex constructs of the source language have already been broken down into simpler primitives). It has been chosen in GCC because it is a form on which optimizations can be efficiently performed.

We also considered using SSA as a starting point in LUSSY (at that time, SSA was not included in a stable version of GCC, but was already available in a development branch). Using it would have considerably eased the translation of the C++ code into HPIOM (reduced the corresponding code in BISE by a factor 10 probably), but would have made the task much harder, if not impossible, for PINAPA: recognizing a statement such as `port.write(true);` in the SSA form is very difficult, because the simple function call

statement has already been broken down into many small operations. The benefit in BISE would probably not have been worth the trouble since it concerns only the C++ part, whereas most of the effort in BISE is related to SystemC and TLM constructs.

The interesting property of SSA exploited in the translation into signal is the fact that since a variable is assigned once only in each block, the transformation performed by each block can be done in one clock cycle. The translation into a data-flow synchronous language is therefore relatively straightforward. This translation allows checking properties similar to the ones available in LUSSY: termination, dead-locks, concurrent write accesses, ...

[TBS+04] describes an initial version of the application of this method to the case of SystemC. A surprising aspect of this work is that instead of *modeling* the SystemC scheduler (with, in particular, the notion of δ-cycle) using the polychronous framework, it *replaces* it with the synchronous semantics. Both approaches achieve the same goal (simulate the physical behavior of a circuit), so the resulting behaviors are similar, and actually equivalent on a subset of SystemC. However, it is easy to find a counter-example where the δ-cycles of a SystemC program do not model the fixed-point of a combinational system, and therefore where the approach does not apply. One is provided in Figure 6.19 (the process `side_effect::f` may have to be rescheduled several times to compute the fixed-point. If it is executed more than once, then an error is raised instead of recomputing the output based on the input). It is actually the case for most processes with a notion of state and side effects. In practice, in most of the cases, a process with side effect will not be rescheduled several times, and processes relying on δ-cycle to be rescheduled will be purely combinational. However, components like the TAC arbiter implementing an arbitration policy have to rely on δ-cycle.

```
1   SC_MODULE(side_effect) {
2     bool been_there;
3     sc_in<bool> in;
4     sc_out<bool> out;
5     void f(void) {
6       ASSERT(! been_there);
7       been_there = true;
8       out.write(in.read());
9     }
10    SC_CTOR(side_effect) {
11      been_there = false;
12      SC_METHOD(f);
13    }
14  };
```

Figure 6.19: A module abusing the notion of δ-cycle

In the meantime, some work has been carried out to model more precisely the details of the SystemC API. Hopefully, the counter-example mentioned above is no longer a counter-example.

Note however that some process do have the fixed-point semantics, and could be managed in an optimized way. Indeed, [KS05] provides sufficient conditions and optimized treatment for some categories of processes.

6.7 Conclusion

We presented the component BISE of LUSSY that translates SystemC code into HPIOM. BISE also integrates different kind of properties to be verified for the platform into the generated HPIOM. It translates the C++ code, and has a direct semantics of SystemC constructs. Additionally, we also modeled the TAC and BASIC protocols that are not yet standardized in SystemC itself.

Part III

Using HPIOM for Formal Verification

Chapter 7

BIRTH: Back-end Independent Reduction and Transformations of HPIOM

Contents

7.1 Introduction . . . 111
7.2 Results of BIRTH on Some Examples, Without Optimization . . . 112
7.3 Semantic Preserving Transformations . . . 113
7.3.1 Abstract Addresses Expansion . . . 113
7.3.2 Non-Deterministic Choices Expansion . . . 113
7.3.3 Reducing the Number of Variables and Inputs . . . 113
7.3.4 Reduce the Number of States by Parallelizing Transitions . . . 117
7.4 An Approximation Conservative for Safety Properties: Abstracting Away Numerical Values . . . 118
7.5 Non-conservative Approximations . . . 118
7.5.1 Initialize variables deterministically . . . 119
7.5.2 Limit the Depth of the Proof . . . 119
7.5.3 Specify the Initial State Manually . . . 119
7.6 Conclusion . . . 120

7.1 Introduction

While transforming automata into LUSTRE or SMV code, one of the goals of LUSSY is to generate the most optimized code possible for a given semantics. Most optimized means for example as few variables as possible, as few inputs as possible.

Many optimizations can be performed at the time the HPIOM is generated. We have already described some of them in section 5.5.3. But incorporating too many optimizations in the generation itself would make the code more complex. Some optimizations are global and could hardly be performed before the complete system, or at least a complete automaton is generated. We preferred a good separation of the HPIOM generation and the optimizations. This way, BISE can concentrate on the correctness of the semantics, and another component, will transform the generated HPIOM, focusing on the optimizations themselves, independently from the semantics of the source language. In the future, HPIOM may be used outside LUSSY for purposes other than analyzing SystemC code. With a separate software component for the optimizer, any tool generating HPIOM can benefit from the optimizations. The same reasoning applies for the

back-ends: any optimizations performed prior to the back-ends is done once and for all, while any optimization performed in the back-ends has to be duplicated for each of them. The optimizations presented here are therefore both independent from the source language (SystemC) and from the target language (LUSTRE, SMV).

We have therefore implemented optimizations as a separate component, BIRTH, that performs HPIOM to HPIOM transformations.

Some of the optimizations presented here are usual optimization techniques that have simply been adapted to our structure (live variable analysis for example). Others are more original, for example, setting the initial value of an initially inactive variable to a fixed value is exactly the opposite of what a traditional compiler does after live variable analysis.

The symbolic model-checking techniques that we will use to prove properties on HPIOM systems have performances that are directly linked with the size of the system. An optimization that reduces the number of variables and the number of inputs of the system will usually improve the performances (but it is always possible to find corner-cases where a the prover will find a good variable ordering to make small BDDs with the unoptimized system, and where the optimized system will have fewer variables, but bigger BDDs). Our goal will be to reduce the number of variables and the number of inputs. In the results we give, we also provide the performances of the provers, using the back-ends described in the next chapter, to get an idea of the impact of the modifications we made on the global tool-chain.

We will give numerical results for the EASY platform that we described in section 2.5.2. Unfortunately, even with all optimizations enabled, no back-end was able to lead to a successful proof for the generated HPIOM system (SMV starts iterating but hangs after a few hours of computation). We can therefore give results only in terms of size of the HPIOM system, but not in terms of proof performance for this platform. In addition, we give the results for the example platform introduced in section 2.4.3.6, which is much smaller and allows us to complete the proof with both LUSTRE and SMV back-ends, with or without optimization.

Some of the transformations implemented in BIRTH are not optimizations, but reductions of some high-level constructs to lower-level constructs (like abstract addresses transformed into Boolean variables, expansion of non-deterministic choices into unknown values, ...).

This chapter will present several transformations: first, the semantic preserving, in section 7.3, then, the conservative approximations in section 7.4. Section 7.5 will give some examples of non-conservative approximation.

7.2 Results of BIRTH on Some Examples, Without Optimization

The results for the unoptimized HPIOM system are the following for the EASY platform:

- 137 automata
- 1,074 control points
- 1,789 edges
- 1,105 Boolean variables
- 293 Boolean unknown values

For the smaller example, we get the following:

- 29 automata
- 103 control points
- 197 edges
- 51 Boolean variables
- 13 Boolean unknown values

And following are the resulting performances for the proof engine. Time is in seconds on a dual AMD Athlon(TM) MP 2000+ with 512Mb of RAM, running Linux. The number of BDD nodes allocated gives a measurement of the memory usage.

- SMV time: 0.48s
- SMV BDD nodes allocated: 71,595
- LESAR time: 5.04s

The next sections will present the same results, for the optimized system. This evaluates the impact of our optimizations.

7.3 Semantic Preserving Transformations

The transformations presented in this section preserve the semantics of the HPIOM system considered. This means the set of possible executions will be the same for the transformed system and the original one. An immediate consequence is that all properties are preserved.

7.3.1 Abstract Addresses Expansion

7.3.1.1 Abstract Addresses in LUSSY

In section 5.3.2, we presented an encoding for an abstraction of the address space in HPIOM. This abstraction uses a dedicated construct in HPIOM in the translation from SystemC to HPIOM, that we will transform into a pure Boolean encoding.

7.3.1.2 Expanding Abstract Address Into Arrays of Booleans

An HPIOM variable or signal of type abstract address will be replaced by an array of Boolean, of size the number of intervals in the address range. This transformation is implemented using a transformation visitor (see section 5.5.2.2.2).

Abstract address literals are encoded straightforwardly: the ith Boolean value is given directly by the value for the ith interval in the abstract address literal. Abstract address manipulators are a bit harder to implement, but the algorithms for the constructs we have implemented are derived straightforwardly from their specifications (sections 5.3.2.3 and 5.3.2.4).

7.3.1.3 Alternative Approach

This transformation could have been done on-the-fly: instead of returning a dedicated HPIOM construct, the abstract address operators could have returned the encoding directly in terms of Boolean encoding. Our approach is more flexible since we can do other treatment on the abstract addresses before expanding them into Boolean. It is on the other hand more intrusive in HPIOM: if we consider HPIOM as a generic intermediate format for several tools, it would not be acceptable for each tool to add its own “home made” constructs like we did for abstract addresses.

7.3.2 Non-Deterministic Choices Expansion

When, in an HPIOM automaton, a control point has several outgoing edges, and if the conditions on the edges can not be guaranteed to be mutually exclusive, the ambiguity has to be removed with a non-deterministic choice. The HPIOM API automatically adds it for each control point when necessary, unless specified otherwise (with `set_choice_safe()`).

A non-deterministic choice is an HPIOM condition. A non-deterministic link is attached to a control point, and can link several non-deterministic choices together. The semantics says that one and only one of the choices can be true at a time.

BIRTH can transform this construct into a Boolean encoding: For a choice between n possibilities, it will use $\lceil log_2(n) \rceil$ Boolean unknown values. This transformation is not mandatory since the SMV back-end can do a better encoding using SMV enumerated types.

7.3.3 Reducing the Number of Variables and Inputs

The number of variables in the system to verify is known to be important in particular for symbolic model-checking. It has a direct influence on the size of the BDDs manipulated. The same applies for the number of inputs.

7.3.3.1 Live Variables Analysis

Minimizing the number of variables of a program is a very general problem. Any decent compiler will try to minimize the memory footprint of the generated executable, and will therefore try to reuse the unused memory as much as possible. To minimize the amount of memory necessary to store the local variables of a function, the usual approach is to perform a live variable analysis to know where each variable may be useful, and in a second pass, decide which register and which memory byte will be allocated for each variable at which point in the code. The general algorithm is presented for example in [BFG99, ASU86].

We use a slightly modified version of the traditional algorithm. The computation of live variables didn't change much, but had to be adapted to our model with assignments and guards on automata edges (as opposed to instructions on nodes of a control-flow graph). The optimization resulting from this analysis is a bit different from what we see in compilers, since we have no notion of register.

7.3.3.1.1 Live Variable Computation.

Definitions. Intuitively, for a given edge, a *live variable* is a variable that holds a value that may be used in the future. This means that the variable may have been assigned a value, and that this value may be used before the next access to this variable.

The definition of live variables is based on some other definitions. First of all, we define the set of incoming edges to an edge as the set of incoming edges to the source control point. Similarly, the set of outgoing edges from an edge is the set of outgoing edges from the target control point.

Each of the following definitions apply for each edge of the automaton:

Defined Variables are variables that appear in the left-hand side of an assignment on an incoming edge to the current edge. We note $\mathsf{def}(t)$ the set of defined variables for edge t.

Used Variables are variables that appear either in the right-hand side of an assignment, in the guard, or in a valued signal emission in an outgoing edge from the current edge. We note $\mathsf{use}(t)$ the set of used variables for edge t.

Live-In Variables are variables live on an incoming edge to the current edge. We note $\mathsf{in}(t)$ the set of live-in variables for edge t.

Live-Out Variables are variables live on an outgoing edge from the current edge. We note $\mathsf{out}(t)$ the set of live-out variables for the edge t.

Live Variables in a control point are then variables live-out in an incoming edge of the current control point.

The definition of a live variable is based on the propagation of the notions of used and defined variables along the edges. Intuitively, defined variables will propagate forward, and used variables will propagate backward, stopping when encountering an assignment. Live variables are the ones in the intersection of the result of the propagation for used and defined.

Formally, we define $\mathsf{in}(t)$ and $\mathsf{out}(t)$ the smallest fixed point of:

$$\begin{aligned} \mathsf{in}(t) &= \mathsf{use}(t) \cup (\mathsf{out}(t) - \mathsf{def}(t)) \\ \mathsf{out}(t) &= \bigcup_{t' \in \mathrm{succ}(t)} \mathsf{in}(t') \end{aligned}$$

Iterative Algorithm. The unoptimized algorithm is rather straightforward from the definition. The following sequence can be proved to be increasing, and therefore converging to their fixed point defined above.

$$\begin{aligned} \mathsf{in}_0(t) &= \varnothing \\ \mathsf{out}_0(t) &= \varnothing \\ \mathsf{in}_{n+1}(t) &= \mathsf{use}(t) \cup (\mathsf{out}_n(t) - \mathsf{def}(t)) \\ \mathsf{out}_{n+1}(t) &= \bigcup_{t' \in \mathrm{succ}(t)} \mathsf{in}_n(t') \end{aligned}$$

This algorithm can be optimized in many ways. An obvious optimization is to group the edges between two branching control points, and consider basic blocks instead of HPIOM control points and edges. On a typical automaton obtained from imperative code, this considerably reduces the size of the automaton to consider. Another common optimization is to order the control points to minimize the number of iterations. Starting from the final control point and iterating backwards is usually considered as a good heuristic. Also, using optimized data-structures like bit vectors to represent control points would clearly speed up the process (we currently use the STL's set operators).

7.3.3.1.2 Optimization of the Number of Variables. Our problem is a slight variant from the usual problem compilers solve with live variable analysis. The usual problem is: given a number of registers, find the way to allocate registers to variables that will minimize the number of variables not allocated to a register, or better, minimize the number of access to variables that are not allocated to registers.

We can represent the problem by a graph in which each variable is represented by a node, and then, add one edge between two nodes whenever both of the corresponding variables are live in the same control point. This is called the *conflicts graph*. Then, the problem is reduced to the one of graph coloring, which is a well-known and NP-complete problem. [Cha82] describes some techniques to optimize the register allocation based on graph coloring algorithms. [Mue93] is a review of the existing algorithms.

Our problem is a bit different: we do not have to do the distinction between registers and memory. The problem is only to minimize the number of variables.

We use a simple greedy algorithm, that seem to be a good heuristic in practice. The goal is to compute a set of replacement variables V_{rep} and a map $R : V \to V_{rep}$ (note: the size of V_{rep} is not known yet, but we can consider an infinite ordered set V'_{rep}: we will allocate its first variables in priority. Then, at the end of computation, V_{rep} will be the set of assigned variables in V'_{rep}. Concretely, this means that V_{rep} is allocated on demand during the execution of the algorithm). We will use two temporary maps $A : V'_{rep} \times S \to \{\text{assigned}, \text{unassigned}\}$ and $U : S \to 2^V$. A represents the role of the variables in each control point, which are initially assigned to unassigned in each control point. U is the set of original variables that have not yet been assigned. The algorithm is then as follows:

```
for s ∈ S loop
    for v ∈ U(s) loop
        v_rep := smallest variable for which
            ∀s ∈ set of control points where v is live, A(v_rep, s) = unassigned
        for s_a ∈ set of control points where v is live loop
            A(v_rep, s_a) := assigned
            U(s) := U(s) − v
            R(v) := v_rep
        end loop
    end loop
end loop
```

Remark:	We perform this optimization type-by-type, which means that a variable used to store an array of Boolean, for example, may not be used to store some scalar Boolean variable in other points of the program. To take full advantage of this optimization, we should encode as many types of variables as possible into individual Boolean variables, before performing the live variables analysis.

7.3.3.1.3 Other Optimizations Made Possible by Live Variable Analysis. Live variable analysis also allows some other optimizations. The first one is based on the fact that the initial value for a variable is never used if this variable is not active in the initial control point. We can therefore replace non-deterministic initial values by arbitrary values in this case. This reduces the number of initial global states for the prover, and also reduces the number of inputs in the case of the LUSTRE back-end.

The second optimization is to remove assignments for variables not live-out on the edge on which the assignment is performed. This simplifies the transition relation of the automaton.

Another potential optimization, which hasn't been implemented, would be to reset the variables (assign them a constant value chosen once and for all) when they leave their live scope. This way, the proof wouldn't have to carry different values for the variables when they go out of scope. In Figure 7.1, if x isn't used in the rest of the automaton, then it goes out of its live scope after being tested. Without the proposed optimization, the value of x would not be the same at the end of the upper branch and the end of the lower branch of the automaton. The resulting BDD would therefore include a condition of x. Our optimization would consist in adding the assignment $x := true$ to the edges where x is tested and therefore goes out of scope. The value of x in the joining control point would then always be $true$.

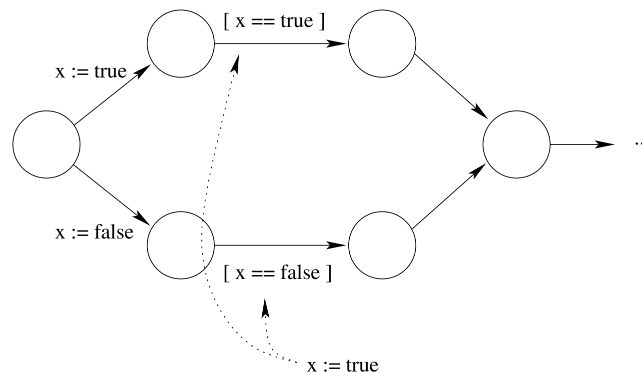
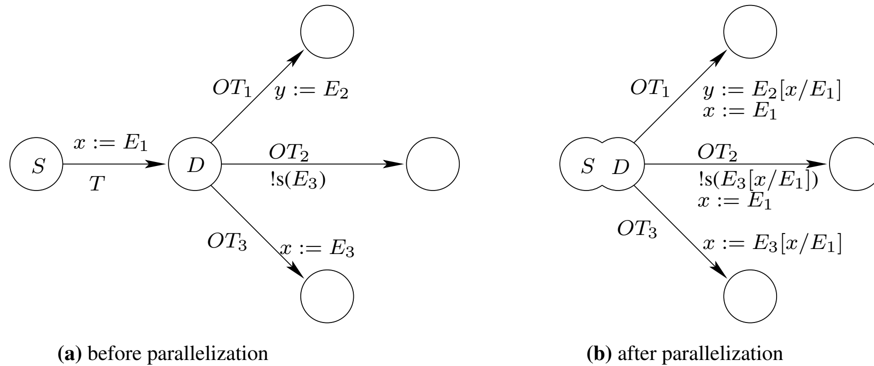


Figure 7.1: Illustration of the Reset Optimization

7.3.3.1.4 Practical Results. On the HPIOM generated for the EASY platform, the total number of boolean variables (we don't consider non-boolean variables since they will be abstracted before the proof in our case) is decreased from 1,105 to 793. In other words, the number of variables is decreased by 28%. The number of unknown values is also reduced from 293 to 206 (because we eliminated unused initial values). On the example platform, the number decreased from 51 to 43 (the gain is 17% in this case), which changes the time taken for the proofs:

- SMV time: 0.47s (used to be 0.48s)
- SMV BDD nodes allocated: 70,564 (used to be 71,595)
- LESAR time: 1.242s (used to be 5.039s)

We can guess from the results that SMV already did the same kind of optimization (but not exactly the same, since the number of BDD node slightly decreased). On the other hand, the speedup using LESAR is impressive.



(a) before parallelization **(b)** after parallelization

Figure 7.2: Merging Two Sates

7.3.3.2 Reuse unknown values

7.3.3.2.1 The Algorithm. We implemented another trivial but interesting optimization, to minimize the number of inputs the system needs to model HPIOM unknown values. The idea is that unknown values will typically be used once only in the automaton, but will, in the unoptimized version, require one input per unknown value. If two unknown values are not used at the same time, then we can use the same unknown value for both.

The problem is similar to the minimization of the number of variables described above, but there is one important difference: since unknown values do not keep a value from one instant to the other, we can compute the replacement control point per control point, which allows us to compute the optimal solution (not a heuristic) easily: we keep a list of unknown values used as replacement, globally for the automaton (initially empty, and filled-in on demand). For each control point, we compute the list of unknown values used on each outgoing edge and assign them replacements.

7.3.3.2.2 Practical Results. On the EASY platform, this optimization allowed us to reduce the number of boolean unknown from 293 to 211 (28% gain). On the example platform, the number is decreased only from 13 to 12, and we get the following performance results:

- SMV time: 0.48s (used to be 0.48s)
- SMV BDD nodes allocated: 71,577 (used to be 71,595)
- LESAR time: 4.8s (used to be 5.039s)

Once again, LESAR benefited of the optimization noticeably, but the results with SMV did not change much. We can guess that SMV already has a well optimized management of non-deterministic values.

7.3.4 Reduce the Number of States by Parallelizing Transitions

7.3.4.1 The Algorithm

The main problem with model-checking is the state explosion. Any optimization able to reduce the state-space is therefore appreciable.

We implemented a transformation able to reduce the number of control points and edges in an HPIOM automaton, by parallelizing independent transitions that would have been executed in sequence. Note that this transformation does not preserve the semantic of an HPIOM automaton in the general case: since HPIOM has a synchronous semantics, doing in one step something that should have been done in two does matter. For processes automata, however, the number of transitions doesn't matter because of the non preemptiveness of the scheduler. Reducing the number of transitions is a kind of automatic parallelization.

When a control point has only one outgoing edge, targeting a control point with only one incoming edge, if the edge emits no signal, then we can merge the control point and its successor. Figure 7.2 illustrates this transformation.

The algorithm is as follows: let T be the edge considered, S its source control point, and D its destination control point. Actions of T are "propagated" onto each outgoing edge OT_i of D: we copy each assignment of T on OT_i, and the left value is substituted in all existing expressions in OT_i. S and D are "merged" (incoming edges to S are redirected to D and S is deleted).

7.3.4.2 Practical Results

On the EASY platform, applying this algorithm reduces the number of edges from 1,789 to 1,747 (a gain of 42, i.e. 2.3%), and the number of control points from 1,074 to 1,032 (3.9% gained). On the example, the optimization were not applicable on any edge (the source code doesn't contain sequential, independent actions).

Those results are a bit disappointing. It would be interesting to refine the condition of application of the transformation and see if we can get better results.

7.4 An Approximation Conservative for Safety Properties: Abstracting Away Numerical Values

Management of numerical values is known to be one of the most difficult problems in model-checking. It makes the general problem undecidable in the case of unbounded integers, and the subproblems (abstract interpretation and exact model-checking with bounded integers) costly in terms of performance.

Among the tools we use in the back-end, none can manage integers in a satisfying way for large systems: SMV encodes integer ranges as a finite set of values. This allows exact model-checking, but a few 32-bits variables will lead to a state explosion. NBAC uses a conservative approximation based on polyhedra, which allows to prove some properties on larger systems. However, we quickly gave up trying to use it on large systems, since NBAC took several hours and thousands of megabytes of memory to prove properties on a ridiculously small example. LESAR will by default make a complete abstraction of integers. the `-diag` option allows cutting some transitions with a local satisfiability analysis. In practice, this means that removing completely the integers from the HPIOM model will not penalize us: in some cases, it will not change anything (case of LESAR), and the other cases are the cases when the proof engine was anyway not able to do anything (NBAC and SMV on non-trivial systems).

It is therefore reasonable to completely get rid of integer values in HPIOM, before entering the back-end. This is implemented and optional for LUSSY. When using the LUSTRE back-end, and LESAR without the `-poly` option, this will actually not change anything. The SMV back-end does not manage integers at all (this could be added easily, but we have no hope to be able to do anything with it anyway ...).

Integer removal is implemented using a transformation visitor. It considers `int` values and array of numerical values as numerical values and removes them from the system:

- Numerical variables are removed from each automaton,
- Comparisons (condition depending on numerical variables) are replaced by boolean unknown,
- Valued signal carrying a numerical value are replaced by pure signals. Conditions on their presence is left unchanged and condition on their value are replaced by non-determinism.

7.5 Non-conservative Approximations

The approximation presented up to now are *conservative* in the sense that any property true for the approximate model remains true for the exact model. We will present here some non-conservative abstractions. The set of behaviors of the approximate system is only a subset of the behaviors of the exact system.

In theory, such approximations are useless. First, this means that the validity of a proof can never be ensured (only the validity of a counter-example is guaranteed, which is much less interesting). Furthermore, applying an over-approximation in addition to an under-approximation can lead to ... just *any* system. Neither the answer "true" nor the answer "false" from the proof engine can be ensured.

However, experimenting with such kind of approximation can be interesting: the result of the proof will clearly not be relevant, but the time to get an answer can give an idea af the bottlenecks. If we find

a non-conservative approximation that allows a real speedup of the proof, this means we have to search a way to achieve a similar speedup in a conservative way.

7.5.1 Initialize variables deterministically

This approximation can give an idea of the importance of initial values. When an HPIOM variable is created without initial value, normally, an unknown value is used. This approximation replaces it with a constant value.

The results are particularly impressive with the example platform: the proof completed with LESAR in 0.8s (almost 10 times faster). An interesting result is that using live variable analysis in addition does not increase the performance. It seems the main effect of the live variable analysis on the performance of LESAR was the removal of unused non-deterministic initial values.

This means that a program that never uses the initial value of an uninitialized variable will be easier to prove (this is anyway a good coding guideline, and can be checked statically in a pessimistic way). On a program complying with this rule, live variable optimization is superior to this inexact optimization.

7.5.2 Limit the Depth of the Proof

We will now present two ideas of non-conservative abstractions, that have not been implemented by lack of time. They are presented here since they are approximations, but would actually have to be implemented in BISE.

The embedded systems on which formal verification works well are usually systems on which the diameter (the greatest distance between two control points) of the state-space is small. Commercial tool demonstration often exhibit proudly a counter-example of size 15 or 20, while we got counter-examples longer than 100 on relatively small systems. The problem is that if the system has a big diameter, the proof engine will most probably "blow up" (running indefinitely or requiring more memory than the machine has) without answering. It is legitimate to prefer an approximate answer to no answer at all[1].

A solution is to limit the length of the execution. We can modify the semantics of SystemC to say for example "at the end of the δ-cycle, the simulation ends successfully", or "at the end of the nth δ-cycle, the simulation ends successfully". A bit more elaborated would be an instruction added in the SystemC code to say "the simulation ends successfully now", like `ASSERT(false)` says "the simulation ends *un*successfully now". This would allow to prove that nothing bad happens before a certain point in simulation.

Concretely, this would mean adding a sink-control point to the scheduler, send the scheduler to this control point when the specified condition is met, and make sure no process can be elected while the scheduler is in this control point.

7.5.3 Specify the Initial State Manually

Another approximation, which hasn't been implemented either, would be to specify an execution prefix for the set of executions. In other words, lead the simulation to a certain point, and launch the proof from this point.

Specifying the path to the initial state could be done either from HPIOM of from SystemC. If done with SystemC, it would be interesting to specify it with a traditional debugger like GDB. This would mean extending PINAPA to be able to go beyond the end of elaboration before analyzing the platform, and modify BISE also to be able to model the initial state correctly based upon the information of PINAPA.

Changing the initial state can reduce the complexity of the proof when the platform's execution starts by a relatively deterministic boot phase to which it never goes back. In this case, the boot phase would be simulated in SystemC, and the actual behavior would be proved for a given boot sequence.

In many cases, however, changing the initial state would not reduce much the depth of the state-space to perform a successful proof. It could however be used for bug searching: the user would lead the system

[1] As a reminder, we're not verifying critical systems on which the goal is "0-bugs", but systems on which bugs are costly. The goal is therefore "find as many bugs as possible, and otherwise give relative confidence in the correctness"

to a state that he suspects close from a bad state using a debugger, and then launch the proof, possibly a bounded proof as explained in the previous section. The system would then answer either “true property” (“there’s no bug near where you are. Search somewhere else”) or give a counter example (“here’s how to reach the buggy state you’re looking for”).

Many variations around this principle can be implemented. Using formal method combined with traditional debugging can provide very efficient tools and methodologies. See for example [JJGM03] for how this kind of techniques has been successfully applied to make LUDIC, NBAC and the test generation LURETTE work well together.

7.6 Conclusion

The results for the HPIOM system with all conservative optimizations enabled are the following for the EASY platform:

- 137 automata
- 1,032 control points (used to be 1,074)
- 1,748 edges (used to be 1,789)
- 793 boolean variables (used to be 1,105)
- 124 boolean unknown values (used to be 293)

For the smaller example, we get the following:

- 29 automata
- 102 control points (used to be 103)
- 196 edges (used to be 197)
- 43 boolean variables (used to be 51)
- 7 boolean unknown values (used to be 13)

And the resulting performances for the proof engine were:

- SMV time: 0.46 (used to be 0.48s)
- SMV BDD nodes allocated: 70,328 (used to be 71,595)
- LESAR time: 0.93s (used to be 5.04s)

note that some numbers are better than those given previously. This is because the combination of optimizations can be better that the optimizations taken individually. For a non-trivial platform, the proof will be much longer than the optimizations. The best strategy is probably to enable all the optimizations anyway.

The results are rather pleasing for SMV: first, the results of SMV on the unoptimized model are better than the results of LESAR on the optimized one. Furthermore, SMV does not benefit from BIRTH’s optimizations as much as LESAR, which means that SMV probably already has similar optimizations.

However, we can see a slight improvement in the performances. This can be explained by the fact that our optimizer works on HPIOM, on which we have a clear distinction between control and data, which makes some optimizations simpler and/or more efficient to implement. For example, live variable analysis intrinsically rely on the control structure. Automatic parallelization is even worse, since it changes the semantic of an automaton. We can perform this optimization only because we know that the control structure was built from the one of the C++ code, and that the number of steps to perform an action isn’t relevant.

As a conclusion, the fact that we use an intermediate representation such as HPIOM did not only allow us a better modularity of the code, but also allowed some optimizations that would hardly have been possible otherwise.

The HPIOM API is being separated from LUSSY, and will probably become a common intermediate format for other Verimag tools. The first step is to give it a textual syntax, with import and export tools. An XML format has been defined by Muhammad Muzammil Shahbaz, and import/export capabilities are being implemented. The next step will be to implement the basic automata transformations, such as the product.

Chapter 8

Back-Ends: Connecting HPIOM to Verification Tools

Contents

8.1 Introduction **121**
8.2 Presentation of the Verification Tools **122**
8.2.1 The LUSTRE Tool-Chain 122
8.2.2 SMV Language and Model-Checker 127
8.3 Encoding in LUSTRE **129**
8.3.1 Encoding State Machines in a Data-Flow Language, the Case of LUSTRE 129
8.3.2 Encoding Variables 131
8.3.3 Encoding HPIOM Communication in LUSTRE 131
8.3.4 Results of the LUSTRE Generator for the EASY Platform 131
8.4 Encoding in SMV **131**
8.4.1 Defining a Sequence: `next` Vs `pre` 131
8.4.2 State Machines 132
8.4.3 Encoding Variables 133
8.4.4 HPIOM Communication in SMV 133
8.4.5 Results of the SMV Generator for the EASY Platform 134
8.5 Validation and Debugging **134**
8.5.1 Validating one Back-End at a Time 134
8.5.2 Comparing LUSTRE and SMV Code to SystemC 135
8.5.3 Comparing the Back-Ends 135
8.6 Tools Comparison **136**
8.6.1 Input Languages 136
8.6.2 Performances of the Proof Engines 137
8.7 Validating Real-Life Platforms **142**

8.1 Introduction

The previous chapter presented the extraction of the semantics of a SystemC programs in our intermediate representation HPIOM. With the final objective of formal verification, we need one more step to transform this HPIOM representation into a format exploitable by a model-checker or other proof engines.

We present here the work done to connect LUSSY to the LUSTRE tool chain, and the work done in co-operation by Muhammad Muzammil Shahbaz [Sha05], Master student at Verimag, to connect to the SMV model checker.

The first section (8.2) presents the LUSTRE and SMV tool-chains, in the context of the synchronous reactive systems. We explain our encoding and code generation for LUSTRE (section 8.3) and SMV (section 8.4), and the method we used to validate and debug the implementation (section 8.5). Section 8.6 compare the tools of the SMV and the LUSTRE tool-chain, in terms of input languages and performance of proof engines, and we conclude on the applicability of our verification approach on real life platforms, in section 8.7.

The original motivation for this work was to be able to perform formal verification. We have been able to formally prove properties on small platforms. Unfortunately, we encountered state explosion before reaching the size of real-life platforms. Model-checking as a back-end for LUSSY has very few chance to scale up if done on the complete platform. The next step would be to work on a verification component by component, that would use the current approach of LUSSY to prove each component, but split the proof into smaller parts before launching the model-checker.

The global proof approach was anyway a very interesting experiment. The framework we developed allowed us to do some interesting benchmarks that could not have been done otherwise. This resulted in an interesting comparison of the output languages, and the identification of some weakness of LUSTRE to model state-machines. The back-ends themselves can be considered as part of the HPIOM API and could be reused outside LUSSY: any tool manipulating automata can generate HPIOM automata and benefit from the connection to the LUSTRE and the SMV tool-chain at the same time.

8.2 Presentation of the Verification Tools

8.2.1 The LUSTRE Tool-Chain

8.2.1.1 Reactive Systems

LUSTRE was originally created to implement *reactive systems*, as opposed to transformational systems. A transformational system is a system that reads its input, perform a computation, emits an output and then terminates. A reactive system, on the other hand, is a system that never terminates. It periodically reads inputs and responds with outputs. Figure 8.1 illustrates the way the system interacts with its environment.

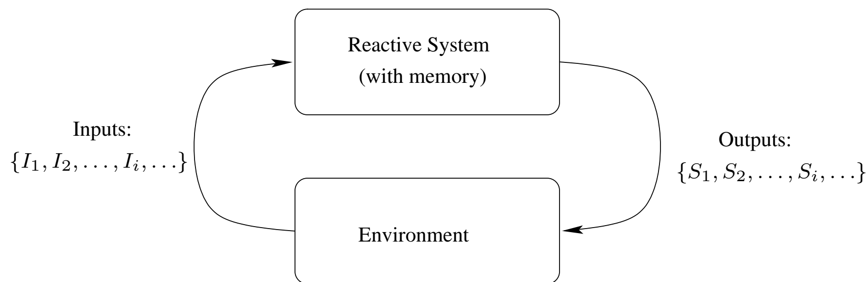


Figure 8.1: Interaction of a Reactive System With its Environment

Each input emitted is a function of the inputs previously received. In other words, it is a function of the current input and the history kept in the memory of the system.

8.2.1.2 The Synchronous Approach

The *synchronous approach* proved to be a very efficient model to design critical reactive systems. The general constraints for such system are the following:

Bounded memory: The system has to be able to run forever, and runs of a physical system on which memory is finite. To make sure that the system will never go out of memory, we have to make sure that the system's memory usage is statically bounded (and make sure this bound is lower than the physical memory of the machine). The best way to enforce this requirement is to ban dynamic

memory allocation. This means in particular that the memory of the system can not contain the whole history, but only a bounded subset of this information.

Parallel description: To allow a component-based approach, a reactive system needs to be able to deal with parallelism. “A system with two components” means two components executing in parallel, not one after the other.

Determinism: To validate the implementation of a system, it is desirable to have a deterministic system, i.e. always get the same sequence of outputs from a given sequence of inputs.

The last two requirements are sometimes conflicting, since parallelism can be a source of non-determinism. The synchronous approach solves this problem by introducing the notion of logical global clock: the execution of the system is split into an infinite sequence of instants. At each instant (or *clock tick*), the system reads some inputs, update its internal memory (or internal state), and emits some output. Communication and computation is assumed to be done in zero time.

The different components are logically executed in parallel, and in practice, an order of execution is computed statically and deterministically, satisfying data dependency. The execution can be fully sequential although the description is parallel (this mechanism is often referred to as *compiled parallelism*).

If a system has no memory, then, the output at each instant is a function of the inputs at the same instant. Such system is called a *combinational* system. More generally, the computation performed at each clock tick is called the combinational logic. Computations depending on the value of the memory is called the *sequential* logic.

8.2.1.3 LUSTRE: A Synchronous Data-Flow Language

In the middle of the 1980s, the Verimag laboratory developed the synchronous data-flow language LUSTRE. It was originally developed to suite the needs of control engineers who were used to manipulate systems of equations on data-flows to describe the behavior of their systems, and had to implement those systems on sequential machines.

The basic principle of LUSTRE is that the objects manipulated are infinite sequences of data ($X = \{x_1, x_2, \ldots, x_i, \ldots\}$). Some operators are defined to compose those sequences, and a LUSTRE program is a set of equations on the sequences in a particular form.

8.2.1.3.1 Combinational Operators. Classical, or combinational operators act on the sequences point by point. For example, the $+$ operator in LUSTRE is defined as follow:

$$\begin{aligned} X + Y &= \{x_1, x_2, \ldots, x_i, \ldots\} + \{y_1, y_2, \ldots, y_i, \ldots\} \\ &= \{x_1 + y_1, x_2 + y_2, \ldots, x_i + y_i, \ldots\} \end{aligned}$$

The following operators are defined in the same way:

- Boolean operators: `or`, `and`, `not`, `xor` and implication `=>`,
- Usual arithmetic operators: `+`, `-`, `*`, real numbers division `/`, integer division `div` and modulo `mod`,
- Comparison operators on integers and real numbers: `=`, `>=`, `>`, `<=` and `<>` (the `=` operator can also be used on Boolean),
- Conditional structure: the expression `if C then E1 else E2` (where `C` is a boolean expression and `E1` and `E2` are 2 expressions of the same type) describe the flow `X` such that $\forall n.\ \texttt{if}\ \texttt{C}_n\ \texttt{then}\ \texttt{X}_n = \texttt{E1}_n\ \texttt{else}\ \texttt{X}_n = \texttt{E2}_n$.

8.2.1.3.2 Operators on Flows. The combinational operators do not allow the behavior to depend on history. LUSTRE defines two operators on streams: `pre` (for “previous”) and “`->`” (for “followed by”), defined by:

$$\texttt{pre}(\{x_1, x_2, \ldots, x_i, \ldots\}) = \{nil, x_1, x_2, \ldots, x_{i-1}, \ldots\}$$

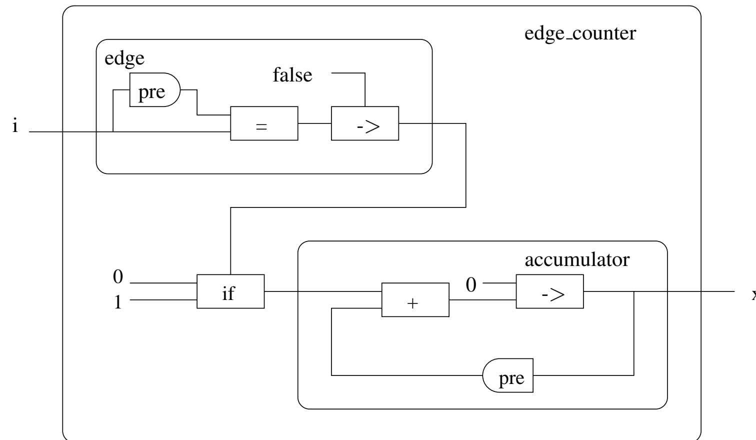


Figure 8.2: Example of a Graphical LUSTRE Program

The operator `pre` allows the system to have some memory. The operator `->` is the necessary complement. Since `pre` leads to a flow whose first element is undefined, we need an operator to define the first element of the flow:

$$\{x_1, x_2, \ldots, x_i, \ldots\} \texttt{->} \{y_1, y_2, \ldots, y_i, \ldots\} = \{x_1, y_2, \ldots, y_i, \ldots\}$$

Those operators are often used together like in

$$X- > \texttt{pre}(Y) = \{x_1, x_2, \ldots, x_i, \ldots\} \texttt{->} \texttt{pre}(\{y_1, y_2, \ldots, y_i, \ldots\}) = \{x_1, y_1, y_2, \ldots, y_{i-1}, \ldots\}$$

8.2.1.3.3 Equations. A variable `v` of a LUSTRE program is defined by an equation of the form `v =` *expr* `;`. The equation can be recursive like in `n = 0 -> (pre(n) + 1)`, which defines the flow $\{0, 1, 2, \ldots\}$.

8.2.1.3.4 Nodes. A LUSTRE node is a set of inputs, a set of outputs, a set of local variables, and a set of equations. It can be compared to the notion of functions in traditional imperative languages.

8.2.1.3.5 Graphical Syntax. LUSTRE programs can also be defined graphically. The graphical syntax of LUSTRE is the one used for its commercial version, SCADE™. It also gives an idea of how a LUSTRE program could be synthesized in hardware. Flows are represented with wires, and operators are "boxes" connecting those wires. Figure 8.2 gives an example of a graphical LUSTRE program. Figure 8.3 is the textual form of the same program.

8.2.1.3.6 The EC format. A program in the LUSTRE language can use any number of nodes. Nodes can be hierarchical. In Figure 8.2, the program is the composition of a node `edge`, that emits `true` on its output whenever the input value changes, and the node `accumulator` that computes the sum of its inputs over time. In the composition, the operator `if` is used to transform the Boolean `true`/`false` in the output of `edge` into the numerical `0`/`1` for the input of `accumulator`. The result is itself a node that could be encapsulated in a larger program, that counts the number of edges on its input.

A LUSTRE program can be "flattened": all the internal nodes can be inlined into the main node. In the graphical representation, this means simply removing the boxes around the operators. In the textual version, the resulting program contains the union of the equations of the nodes. The resulting format is a subset of the LUSTRE language, called *EC*.

EC is a very simple language, easy to parse and to manipulate. Most tools manipulating LUSTRE code in the Verimag tool chain use it as an intermediate format (i.e. call `lus2ec` and work on the resulting file).

```
1   node edge(i: bool)
2   returns (x: bool)
3   let
4     x = false -> (i <> pre i);
5   tel
6
7   node accumulator(i: int)
8   returns (sum: int)
9   let
10    sum = 0 -> pre(sum) + i;
11  tel
12
13  node edge_counter(i: bool)
14  returns (x: int)
15  var
16    edge_int: int;
17  let
18    x = accumulator(edge_int);
19    edge_int = if (edge(i)) then 1 else 0;
20  tel
```

Figure 8.3: Textual Version of the Counter Program

```
1   node edge_counter
2     (i: bool)
3   returns
4     (x: int);
5
6   var
7     V10_edge_int: int;
8
9   let
10    x = (0 -> ((pre x) + V10_edge_int));
11    V10_edge_int = (if (false -> (i <> (pre i))) then 1 else 0);
12  tel.
13
```

Figure 8.4: EC Version of the Counter Program

Remark:	We presented here only the single clock flavor of LUSTRE. LUSTRE actually allows different variables to be computed according to different clocks, but we won't use this feature in LUSSY.

8.2.1.4 Model-checking LUSTRE Programs: LESAR, GBAC

The set of equations in a EC file can actually be seen as the definition of a function

$$(\mathit{output}, \mathit{next\ state}) = F(\mathit{inputs}, \mathit{state})$$

This is exactly the kind of input a model-checker expects.

LESAR is a model checker for LUSTRE developed by Pascal Raymond in the Verimag laboratory. Its input is a LUSTRE program whose first input (necessary a Boolean) gives the property to prove: the property will be "The first output remains true forever". Any safety property can be reduced to this class of property, but LESAR is not able to prove liveness properties.

LESAR first transforms the LUSTRE program into an EC program. Then it removes all the numerical variables and replace all conditions depending on Boolean variables by Boolean inputs. The resulting program is a Boolean, flat program.

This program is then encoded using BDDs (as explained in section 3.2.1.2). LESAR can use either a symbolic or an enumerative exploration of the state-space. The symbolic exploration can be done either forward (starting with the set of initial states, iterating until an error state is found or the fixed point is reached) or backward (starting with the set of error states, applying the inverse of the transition function until an initial state or the fixed point has been reached).

LESAR can also eliminate some numerically unsatisfiable transition guard: when used with the `-poly` option, it will represent the conditions appearing on transitions with polyhedra. If a polyhedra is empty, then the transition can be deleted. Of course, the programmer is not supposed to write automata with unsatisfiable transitions, but they often appear when computing the product of several automata.

The tool GBAC, also developed by Pascal Raymond in Verimag is a prototype for the next generation of LUSTRE model-checkers. It is a symbolic model checker, with several improvements over LESAR, that usually makes it much more efficient. Since it is a prototype, it still doesn't have all the functionalities of LESAR. The main difference between LESAR and GBAC is that LESAR computes the BDD for the transition function statically whereas GBAC computes the BDD at each step. The resulting BDDs are therefore simpler, since the possible values of state variables are known at the time the BDD is computed. GBAC therefore performs many small computations (where LESAR performs fewer costly computations) which is in practice much faster in most cases.

8.2.1.5 Abstract Interpretation: NBAC

NBAC is part of another family of verification tools: it is an abstract interpreter written by Bertrand Jeannet during his Ph.D Thesis in Verimag. It has been designed to verify properties depending on numerical values, and in particular, counters. Local satisfiability would not be of any help in the case of counters: the value of the counter depends on its previous value at each instant. It is therefore impossible to say anything about it locally.

NBAC implements dynamic partitioning to compute a set of relevant control points, and associates with each control point a superset of the possible valuations of the variables. This information propagates from one control point to another.

Initially, NBAC defines three control points: one representing the set of initial states, one representing the set of states that we want to prove unreachable, and one for all other states. All transitions are supposed to be possible.

8.2.2 SMV Language and Model-Checker

8.2.2.1 Introduction

The SMV system is a tool for checking finite state systems against specifications in the temporal logic CTL. It was originally developed by Ken McMillan and serves as a basis for Cadence formal verification tools. The native input language of SMV is designed to allow the description of finite state systems that range from completely synchronous to completely asynchronous, and from the detailed to the abstract. One can readily specify a system as a synchronous Mealy machine or as an asynchronous network of abstract non-deterministic processes. Since it is intended to describe finite state machines, the basic data types in the language are finite scalar types. Static structured data types can also be constructed. A rich class of temporal properties including safety, liveness, fairness and deadlock freedom, is allowed to be specified in CTL, in a concise syntax. SMV uses the OBDD based symbolic model checking algorithm to efficiently determine whether specifications expressed in CTL are satisfied or not.

8.2.2.2 The SMV Language

The primary purpose of the SMV input language is to provide a symbolic description of the transition relation of a finite Kripke structure. Any propositional formula can be used to describe this relation. This provides a great deal of flexibility, and at the same time a certain danger of inconsistency. For example, the presence of a logical contradiction can result in a deadlock – a state or set of states with no successor. This can make some specifications vacuously true, and makes the description unimplementable. While the model checking process can be used to check for deadlocks, it is better to avoid the problem when possible by using a restricted description style. The SMV system supports this by providing a parallel assignment syntax. The semantics of assignment in SMV is similar to that of single assignment data flow languages. A program can be viewed as a system of simultaneous equations, whose solutions determine the next state. By checking programs for multiple assignments to the *same variable*, *circular dependencies* and *type errors*, the compiler insures that a program using only the assignment mechanism is implementable. Consequently, this fragment of the language can be viewed as a hardware description language, or a programming language. The following sections describe its formal semantics in detail.

The SMV language contains all basic ingredients of an input language. It provides basic data types such as Boolean, Enumeration, subrange types and Arrays. It also supports Vectors and Structural data types, as well the self explanatory constructs for Conditions, Loops and module instantiations. All basic Arithmetic, Comparison and Logical operators are of course, included in the language.

8.2.2.2.1 A Trivial Example. Consider the description of a very simple combinational circuit in Figure 8.5, with some assertion added. This example is written in SMV's native language.

```
1   module =  main(req1, req2, ack1, ack2)
2   {
3     input req1, req2 : boolean;
4     output ack1, ack2 : boolean;
5
6     ack1 := req1;
7     ack2 := req2 & ~req1;
8
9     mutex : assert ~(ack1 & ack2);
10  }
```

Figure 8.5: Example of an SMV Program: A Trivial 2-bits Arbiter

The example shows most of the basic elements of an SMV module. The module has four *parameters*, `req1`, `req2`, `ack1` and `ack2`, of which the former two are inputs, and the latter two are outputs. It contains:

- *Type Declarations:* `req1`, `req2`, `ack1` and `ack2` are declared to be type `boolean`. The '`input`' and '`output`' are specialized forms of type declarations.
- *Assignments:* Giving logical functions to outputs `ack1` and `ack2` in terms of inputs `req1` and `req2`.
- *Assertions:* '`assert`' statement defines a property `mutex` to be proved.

The program models a (highly trivial) two bit priority-based arbiter, which could be implemented with a two-gate circuit. The property `mutex` says that `ack1` and `ack2` are not true at the same time. '`&`' stands for logical "and" and '`~`' stands for logical "not".

***8.2.2.2.2 From* LUSTRE *to* SMV.** We will not detail here all the constructs of the SMV language. Starting with the language manual [McM99] or the tutorial [McM01] is a much better way to learn the language anyway. We will present the principles of the language, and insist on the difference with LUSTRE that we introduced above.

The SMV language is similar to LUSTRE in many way. It is also a data-flow language, with hierarchical modules (the equivalent of a node in LUSTRE). It has the power of a synchronous language, although it allows asynchronous modeling: the modules can be executed either using one global clock or using independent clocks for each module. Note that modeling asynchrony can be done this way in any synchronous language ([HB02] gives a general methodology for that), whereas the opposite is not true.

There are a few syntactical differences between LUSTRE and SMV. For example, SMV uses `next(x) := next(y) + z; init(x) := t;` where LUSTRE would use `x = t -> (y + pre(z));`, but this doesn't make a big difference in practice. Note that SMV distinguishes combinational variables (assigned directly) and state variables (whose `next()` or `init()` value is assigned): if a variable `x` is used in "`x :=`" once in the program, then it is stateless and cannot appear in "`next(x) :=`" or "`init(x) :=`", and vice-versa.

Although it has data-flow semantics, the SMV language allows an imperative style. For example, one can write

```
if (x) {
   z := y;
   t := 1;
} else {
   z := 0;
   t := y;
}
```

which would have to be written

```
z = if x then y else 0;
t = if x then 1 else y;
```

in LUSTRE.

SMV also provides some higher level constructs that are harder to implement in LUSTRE, and that the SMV model-checker can use to optimize its encoding. For instance, an SMV program can have intrinsic non-determinism whereas a LUSTRE program is fully deterministic (but non-determinism can be modeled by adding inputs to the program). Non-determinism can be achieved either by providing no value for a combinational variable, or by using the set notation {`X, Y, ...`}, like in `x :=` {`0, 1`}. Using explicit non-determinism, the model-checker may be able to perform some optimizations that couldn't be performed on inputs of the program.

SMV has a built-in notion of enumerated types, i.e. a type with a finite and statically known set of possible values. The value can then be encoded using an efficient data-structure like Numerical Decision Diagrams (NDD) [KPR+97] or other form of log encoding. The typical usage for enumerated types is the encoding of discrete automata: each control point of the automaton is represented by a value of an enumerated type.

The SMV language has no interpreter and is currently used only as input for the model-checker. This is sufficient for us to connect LUSSY to the SMV model-checker, but it makes it very hard to debug and validate: to get confidence in an SMV generated program, one can only read the program, and try to prove properties on it, but not execute it.

8.2.2.2.3 Temporal Properties. SMV uses Temporal Logic, CTL for specifying properties for verification. We will use only a trivially small subset of CTL in LUSSY, since we generate SMV from HPIOM in which the property is already encoded in the automata (using synchronous observers if necessary). We will therefore not describe CTL in details here, but only give a short overview:

CTL, like any temporal logic, is an extension of the combinational logic. A CTL property can be either true or false at each instant t. It can be

E**:** A boolean expression of the variables of the program (no temporal notion here)

`G(`E`)` (the *Globally operator*): true if E is true at *all* times $t' > t$.

`F(`E`)` (the *Future operator*): true if E is true at *some* time $t' > t$.

E `U` F (the *Until operator*): true if F is true at *some* $t' > t$ and E is true at *all* $t'' \in [t, t'[$.

`X(`E`)` (the *Next operator*): true if E is true at $t + 1$.

In LUSSY, the property will always be expressed as `G(`E`)`, where E will be the encoding of “nothing bad happens at time t” (a non-temporal property).

8.2.2.3 Model-Checking SMV Programs

SMV is a symbolic model checker, based on BDD. A SAT engine also exists, but is not available in the public release, so we couldn’t test it.

Model Checking by itself is limited to fairly small designs, because it must reach every possible state that a system can reach. For large designs, especially those including substantial data path components, the user must break the correctness proof down into parts small enough for SMV to verify. This is known as *compositional* verification. SMV provides a number of tools to help the user reduce the verification of large and complex systems to small finite state problems. These techniques include *refinement verification*, *symmetry reduction*, *temporal case splitting*, *temporal case splitting*, *data type reduction*, and *induction*. However, many of the advance concepts are currently out of the scope of thesis. These can be considered during optimization or during the extended work phase.

SMV manages integers in an exact, and therefore very inefficient way: Integers are provided as ranges (*min*`..`*max*) in the source program, and SMV will consider the *max* $-$ *min* $+ 1$ values possible for this type (in a way similar to what is done for enumerated types). This approach is interesting for complex properties depending on small size integers, but even very simple programs containing just one or two 32-bits integers can lead to a state explosion. For a real-life program containing many integer variables, the user will have to find a more efficient encoding, or a complete abstraction before writing the SMV input program.

8.3 Encoding in LUSTRE

8.3.1 Encoding State Machines in a Data-Flow Language, the Case of LUSTRE

Encoding state machines into a data-flow language is a very common problem. Most high level languages have a notion of control-flow, and therefore a notion of explicit state, or control point. Hardware implementations, on the other hand, are purely data-flow. The problem of hardware synthesis is therefore an instance of the problem of encoding state machines into a data-flow.

The general scheme is described in Figure 8.6. The machine receives inputs and emits outputs. It keeps a bounded abstraction of the history in a “state”. In a hardware implementation, the “state” part of the machine corresponds to a set of registers. In LUSTRE, it corresponds to expressions on which the

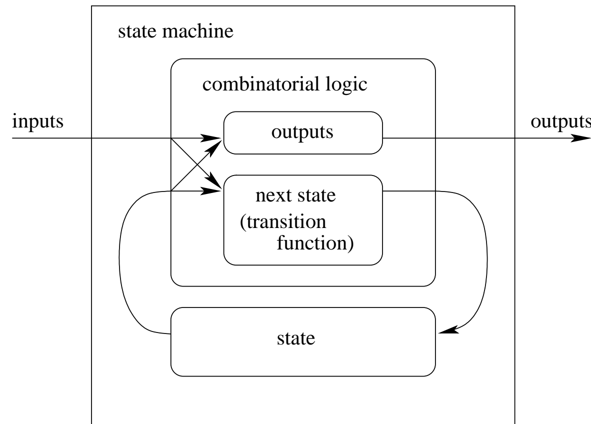


Figure 8.6: General Scheme of a State Machine Encoding

operator `pre` is called. The combinational logic represents two functions: *next state* $= F($*inputs*$,$ *state*$)$ and *outputs* $= G($*inputs*$,$ *state*$)$. In the hardware implementation, it corresponds to wires and gates not combined to build registers. In LUSTRE, this corresponds to everything but the `pre` and `->` operators.

There are always several data-flow implementations of a given state machine. They are discussed for example in [JFL97]. It is relatively easy, given the state encoding, to optimize the combinational logic, and also easy to minimize the number of registers necessary to encode the state. The difficulty is to find the trade-off to optimize both at the same time. If the machine has n different control points, the simplest strategies are:

the *one-hot encoding*: Each control point is represented by a boolean variable. One, and only one of those variables may be true at a time. In this encoding, the computation of the next control point is simple, but the encoding of the state itself is costly in terms of number of state variables (or registers).

The logarithmic encoding: In this encoding, the control points are numbered from 0 to $n - 1$ and each control point is represented by its binary representation, which needs only $\lceil log_2(n) \rceil$ Boolean variables. This encoding minimizes the number of bits of the state encoding, but makes the combinational logic much more complex. In terms of Binary Decision Diagram, this means having less variables, but also less regular BDDs. The result is that a log-encoding usually makes the proof harder than the unoptimized version.

A good trade-off is to use the one-hot encoding by default, and when building the product of automata (which is roughly the concatenation of the equations when the automaton is described in data-flow format), see if some variables are functions of the others.

We have chosen the one-hot encoding mainly because it was the simplest to implement. Allowing a logarithmic encoding would not be hard, but hasn't been implemented by lack of time. We will therefore declare one variable per control point (actually, a trivial optimization that we implemented is to use $n - 1$ variables, since the last variable is true when and only when all the other are false).

The variable representing the initial control point is initially true, while all the others are false. After the initialization, a variable encoding a control point is true when the equation representing one of the edges to this control is true.

We also declare one variable for each edge. Note that these variables are only here to factor the code: they never appear inside the scope of a `pre`, or as input or output of the system, and could be equivalently replaced by their value wherever they appear. The variable representing a transition is true when the variable encoding its source state was true the instant before and when the guard is true.

The equation of the variable for a state is therefore of the form

```
control_point_foo = false/true ->
   (transition_x or transition_y);
```

while the equation of an edge from control point `control_point_foo` to state `control_point_bar` is of the form

transition_x = false -> (pre(control_point_foo) and *guard*);

8.3.2 Encoding Variables

HPIOM variables are just a concise way to express a big, or infinite number of states. It is therefore not surprising that their encoding can be similar to the one of control points: we use one LUSTRE state variable for each HPIOM variable. The variable's value is read with `pre(`*variable*`)` (we define the current state as a function of the previous state), and its value is defined by an equation of the form

variable = initial_value ->

  if transition_x then *value assigned on transition_x* else

  if transition_y then *value assigned on transition_y* else

  pre(variable); -- keep the previous value.

8.3.3 Encoding HPIOM Communication in LUSTRE

HPIOM pure signals are either present or absent, with no value. This is strictly equivalent to a Boolean variable, always present, but either true or false. In LUSTRE, we encode this into a Boolean flow.

Valued signal need two LUSTRE flows: one Boolean to encode the presence of the flow, and one of the type of the signal to encode, to carry its value. The equations for a valued signal in LUSTRE are therefore of the form

valued_signal_present = transition_x or transition_y

valued_signal_value = *any value* ->

  if transition_x then *value emitted on transition_x* else

  if transition_y then *value emitted on transition_y* else

    any value;

any value can be any value of the type of the signal. Concretely, the C++ class representing a type has a method `get_a_value()` that is used to get arbitrary values for any type.

8.3.4 Results of the LUSTRE Generator for the EASY Platform

LUSSY's LUSTRE code generator worked well on the EASY platform. It generates 11,795 lines of code, split in 48 files. The corresponding EC file is 16,278 lines long. None of the LUSTRE tools we tried were able to prove a property on the platform.

8.4 Encoding in SMV

The difference and similarities between the SMV input language and the LUSTRE language have been described in section 8.2.2.2.2. This section will present the impacts of those differences on the encoding of HPIOM.

8.4.1 Defining a Sequence: `next` Vs `pre`

In LUSTRE, the sequential aspect comes from the construct `pre`. In other words, we define the value at each clock tick as a function of the previous values. SMV does the opposite: using the construct `next`, it defines the next state as a function of the current state. The encoding follows this principle.

8.4.2 State Machines

One important difference between LUSTRE and SMV is the presence of enumerated types in SMV. LUSSY can take advantage of this to avoid encoding the state with Boolean variable, delegating the choice of the low-level encoding (one hot, logarithmic, ...) to the tool SMV itself.

The HPIOM state encoding will therefore use only one variable per automaton, of an enumerated type having one value for each control point of the automaton to encode. The imperative style of the SMV language also allows a more readable generated code. In LUSTRE, the variable for a control point was defined based upon its incoming transitions. In SMV, we will define the next control point based upon the outgoing transitions of the current one.

Here is a small example of an expression of a simple finite state machine in SMV.

```
module main()
{
    request : boolean;
    -- declaring states of the machine
    state : {ready, busy};
    -- initializing 'state'
    init(state) := ready;
    -- next state decision box
    switch(state) {
        -- case when state is 'ready'
        ready : {
            if(request) {
                next(state) := busy;
            }
            else {
                next(state) := ready;
            }
        }
        -- case when state is 'busy'
        busy : {
            next(state) := {ready, busy};
        }
    }
}
```

There are two control points `ready` and `busy` and one Boolean variable `request`. The states of the machine are expressed by a single variable of enumerated type, `state`. The initial state is `ready`. When `request` is true, the system changes its state from `ready` to `busy`. Otherwise it remains on `ready`. Once on `busy`, the system may either stay here or change its state to `ready` non-deterministically. The graphical look of this state machine is given in figure 8.7.

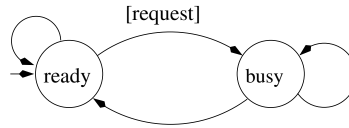


Figure 8.7: Example of a State Machine

8.4.3 Encoding Variables

The encoding of variables in SMV is similar to the LUSTRE encoding. We use one SMV variable per HPIOM variable. One difference is that we do not have to provide a centralized definition for the variable, but can spread the encoding of the assignments throughout the SMV code. The generated code is therefore of the form

```
variable : type;
state : {s1, s2};
init(state) := s1;
default {
    next(variable) := variable;
} in {
    switch(state) {
        s1 : {
            if(condition for this transition) {
                next(state) := s1;
                next(variable) := value assigned on this transition;
            } else {
                next(state) := s2;
            }
        }
        [...]
    }
}
```

The default section is necessary to specify that the variable keeps its value by default, and the assignments are encoded one by one in the section corresponding to the transition on which the assignment is done.

8.4.4 HPIOM Communication in SMV

The encoding of pure and valued signals in SMV are not different from what is done in LUSTRE. We use stateless variables to describe them. Their value are set in the portion of code corresponding to the transitions on which they are emitted, in a way similar to what is done for variables, except for the presence of the `next` keyword.

8.4.5 Results of the SMV Generator for the EASY Platform

The SMV code generator is also operational, and can generate the code for the EASY platform. The generated code counts 17,648 lines of SMV code. Unfortunately, SMV was not able to prove any property on the platform. After a few hours, the process was taking all the 2Gb of RAM available on our machine.

8.5 Validation and Debugging

8.5.1 Validating one Back-End at a Time

The first approach for the validation of the LUSTRE and SMV back-ends is to try LUSSY with the back-end to validate, and to manually examine the output. To avoid having to take into account the full LUSSY tool chain, we implemented a small unit test framework: LUSSY is compiled as a library. The LUSSY binary is simply a trivial function calling the main function of LUSSY linked against this library, and we can implement each unit test as a function that calls some functions in LUSSY, and that contains a main function. Unit tests for SMV and LUSTRE back-ends are functions that builds a simple HPIOM system, and dumps it into LUSTRE and SMV code.

We followed this approach at the early development stages of both back-ends, but this isn't reasonable for integration tests: it would be tedious and inefficient for platforms that generate several thousands of lines of code (it is very easy to miss a bug by manually reading it, and section B.4.3 page 174 even describes a case where the most careful manual examination wouldn't have been sufficient).

Figure 8.8 shows the different steps from SystemC to LUSTRE and SMV, and how they can be compared.

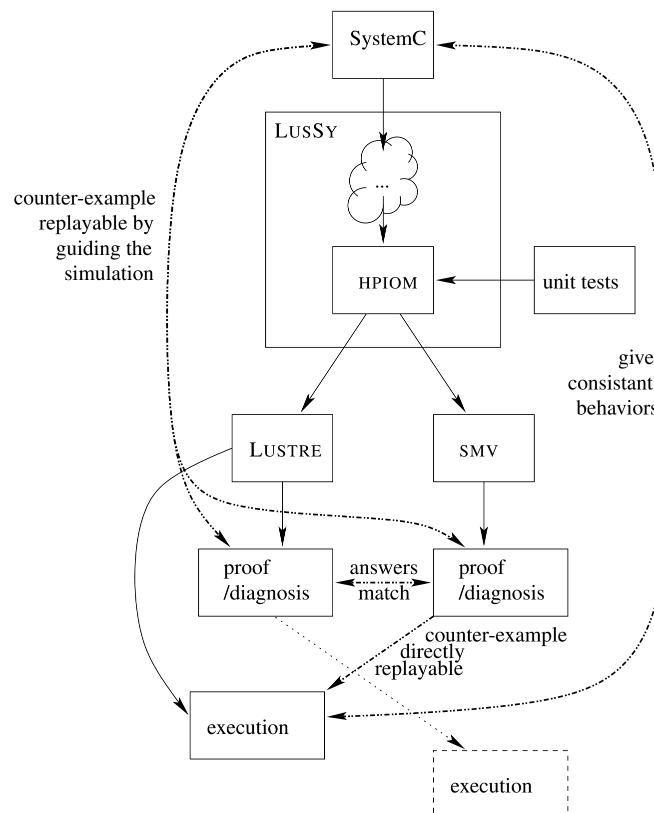


Figure 8.8: Testing LUSTRE and SMV back-ends

The first remark is that we can take advantage of the specificities of the target language. Both of them are provable (that's why we have chosen them . . .), and LUSTRE is also executable, and has a good debugger, LUDIC [MG00].

8.5.2 Comparing LUSTRE and SMV Code to SystemC

We didn't implement an HPIOM interpreter, so we can't directly compare the execution of the generated LUSTRE with the HPIOM system, but we can compare it with the original SystemC platform. This validates the full chain (PINAPA + BISE + back-end) at a time, which is both an advantage and an inconvenient: when the executions are consistent, we ensure the correctness of all the components at a time, but when the executions are inconsistent, it makes it much harder to identify the origin of the problem. In addition to execution, proving false properties and trying to replay the counter-example on the SystemC platform is also very interesting.

Claude Helmstetter, Ph.D student in Verimag and STMicroelectronics, developed RVS, a variation of SystemC which provides among other features a scheduler instrumented to prompt the user instead of choosing automatically a process when several processes are eligible during the evaluation phase. This, associated with GDB to set the values of some variables at runtime if necessary allows to guide the simulation manually, and to reproduce the counter-example provided by LESAR. When LESAR provides a counter-example, it should be reproducible in SystemC, and when LESAR proves the property, the SystemC should never violate an assertion.

8.5.3 Comparing the Back-Ends

Comparing the outputs of both back-ends on the same HPIOM system is a test only for the back-ends and the HPIOM API. If BISE, for example, introduces a bug in the generated HPIOM, the bug should be present in the output of both back-ends, and they should still be equivalent.

Historically, the SMV back-end was started after the LUSTRE back-end was finished, and tested. The LUSTRE back-end could therefore be used as a reference for the SMV one.

8.5.3.1 Comparing Answers

For Boolean programs, the answers from LUSTRE and SMV model-checkers must give the same answer. The counter-examples may differ, but their length should be at least similar: symbolic model-checkers (SMV and LESAR if used with the `-forward` or `-backward` option) always give the shortest counter-example, so, the length should be equal. LESAR does by default an enumerative model checking, which does not guarantee the shortest counter-example. The only guarantee is that LESAR's counter example can not be shorter than SMV's one. Getting a counter-example much longer than the shortest is unlikely to happen, and in practice, we almost always got the shortest even with the enumerative algorithm.

8.5.3.2 Replaying SMV's Counter-Example on the LUSTRE Program

Although the counter-examples provided by both model-checkers can be different, each must be replayable on the other model. Replaying an SMV counter-example in the LUSTRE program does not cause theoretical problem, since we can execute the LUSTRE program with LUDIC, looking at the SMV counter-example.

It does however cause a practical problem: for a reasonably small SystemC platform, we got a 12,000 lines long SMV counter-example that we thought was incorrect (the length of SMV counter-example was 13 while the LESAR diagnosis had more than 60 transitions, and seemed to be the correct shortest counter-example when compared to the SystemC source). We tried to manually examine the trace, but quickly gave up given its size. The graphical user interface VW provided with SMV has not been of a great help: all the variables of the program were displayed at the same level, it was really hard to get the relevant information to replay in LUDIC.

A solution would have been to parse the SMV output and to generate a `.rif` file, that describes the sequence of inputs of a LUSTRE program, and that can be replayed automatically in LUDIC. The transformation is feasible but not trivial since the encoding of some constructs is not the same in SMV and LUSTRE (in particular, the management of non-determinism). We didn't have time to implement it.

Our solution has been to develop a rudimentary SMV trace explorer in Emacs-lisp. It parses the output and splits it into one file per state. The command `diff -u0` can be used to find the difference between two states. We defined key bindings to move easily from one state file to the next and the previous one. A filter can be provided in the form of a regular expression to display only a subset of the variables that

appear in each state. This tool took only a few hours to develop, and allowed to understand the cause of a bug in less than a day, after investigating unsuccessfully for several days without an appropriate tool.

8.5.3.3 Replaying LESAR's Counter-Example on the SMV Program

In theory, the counter-example provided by LESAR should be replayable on the SMV program too. However, since we have no SMV interpreter, we can't do it in practice (this is why the "execution" box is dashed on Figure 8.8). It would, however, be possible to check that the SMV program can replay the counter-example: we could add a state-machine providing the correct inputs to the program at each step (this would be a trivial state machine in which each state has one and only one successor). The property should remain false with this additional constraint. Additionally, we could encode the property "The outputs of the system are the ones obtained in the LUSTRE counter-example", that would have to be encoded in a state-machine similar to the one used to encode the inputs.

8.6 Tools Comparison

The LUSSY tool-chain has several back-ends, and is able to connect to a number of different model-checkers. It is interesting to compare the results of the different tools for several reasons: first of all, we obviously want to know which tool is the best as a LUSSY back-end. But there's something more. If we use the same parameters for all but the back-end part of LUSSY, we get a way to compare tools that would hardly be comparable otherwise: to compare model-checkers for LUSTRE and SMV, we could write a LUSTRE program, test it, and then translate it into SMV, but this is not very satisfying: an automatic translation would not be able to use all the specificities of SMV (for example, it could not refactor Boolean encoding into enumerated types), and a manual translation may introduce arbitrary differences. Here, we start from a common intermediate representation, and generate the code for both languages. Both code generators can take advantage of the target language specificities. The results presented here are therefore interesting outside the scope of LUSSY.

8.6.1 Input Languages

A difference between the tools are their input language. The input language is important because it has an impact on the performance of the prover (a too low-level language would force the user to break down its higher-level constructs, preventing the prover from taking advantage of those particular constructs) and on the possible errors of the code generator (the strictest the syntax check is, the easiest it is to find errors in the generated code).

8.6.1.1 Available Constructs

The main interesting feature of LUSTRE that SMV does not have is the ability to use the `pre` operator on any expression. SMV does not use `pre`, but `next`, which can only be used on the left value of an assignment. This is a powerful feature of LUSTRE, but it is not of any use for LUSSY !

SMV, on the other hand, has several interesting constructs that LUSTRE lacks. The presence of enumerated types, non-deterministic assignments, imperative-style `if-then-else` and `switch` are more difficult to implement in LUSTRE, and SMV can do some optimizations that LUSTRE tools can hardly perform: enumerated types can be encoded efficiently, non-deterministic assignments will presumably be encoded into inputs of the system, but SMV can try to minimize the number of such inputs when it can guarantee that some assignments are mutually exclusive.

8.6.1.2 Ease of Use

LUSTRE has been designed to be simple, can be learned quickly, and has a very clear semantics. SMV is much more complex to learn and understand. This has led to several bugs in LUSSY.

8.6.1.3 Type Checking

SMV has a much weaker type-checking than LUSTRE. The program in Figure 8.9, for example, is accepted by SMV with only one warning, line 7, saying only “warning: assignment to x may be out of range”.

```
1   module main ()
2   {
3     x: boolean;
4     y: array 1..12 of boolean;
5     z: 0..100000;
6
7     x := y;
8     y := 1;
9     z := x;
10  }
```

Figure 8.9: Implicit Conversions Between Boolean and Integers in SMV

There are implicit conversion operators between integers and array of Boolean (an integer being equivalent to its encoding in an array of Boolean), and a Boolean is simply an integer in the range `0..1`. Worse than that: the Boolean operators extends to array elements by elements, whereas the integers operators extend to array considering the arrays as integers. `[0, 1, 0] & [1, 0, 0]` is equal to `[0 & 1, 1 & 0, 0 & 0]`, whereas `[0, 1, 0] + [1, 0, 0]` is equal to `2 + 4` !

As a conclusion, we can say that the SMV language has many features that LUSTRE does not have, but sometimes too many ...

8.6.2 Performances of the Proof Engines

8.6.2.1 Quick Reminder of the Tools Available, and the Way we Use Them

A detailed explanation of the mechanisms of the tools we are using has been given earlier. We give here some precisions on the way we used them, so that the benchmark be reproducible.

8.6.2.1.1 LESAR**.** LESAR is Verimag’s original LUSTRE model-checker. It was developed by Pascal Raymond. Its default strategy is the enumerative one, but it also provides a symbolic strategy (forward, with the option `-forward` or backward, with the option `-backward`). We use the current version as of July 21, 2005.

8.6.2.1.2 GBAC**.** GBAC is a prototype of model-checker also developed by Pascal Raymond in Verimag. It uses a better BDD library and better algorithms when used with the `-fb` (for “Forward Bis”) option. The key point is the computation of BDDs at each step of verification, which is simpler than computing one big, generic BDD at the beginning of the proof. The `-sift` option enables BDD dynamic variable reordering, which usually fasts up the proof.

It doesn’t use LUSTRE as its input, but the BA format, which is obtained from LUSTRE with the command `lus2ba`. The version of GBAC used is the current as of July 21, 2005.

8.6.2.1.3 NBAC**.** NBAC is the abstract interpreter developed by Bertrand Jeannet. We use the version current as of January 6, 2004. Some improvements have been made since that time, but are not sufficient to compete with the Boolean model-checkers for what we try to do anyway, so, we haven’t investigated further. We use it without any option.

8.6.2.1.4 **SCADE*TM*.** SCADE™ is the commercial and graphical environment for LUSTRE, developed by Esterel Technologies. It can be provided with a proof engine called PROVER PLUG-IN™ for SCADE™ . We use the version provided with SCADE™ v5.0. The prover itself is not available as a separate component to check LUSTRE programs, and has to be used through the integrated development environment. We used a somewhat dirty hack to import our generated LUSTRE files in SCADE™: first, the LUSTRE program is compiled into the EC format (flattened LUSTRE). Then, we have to modify it slightly to conform to the SCADE™ syntax. The perl script doing this modification is quite trivial:

```
s/^let$/let  equa well_scade_needs_this_so___[ , ]/;
s/^tel.$/tel;/;
```

We create a template project using SCADE™ itself, with a sample program, and the property stating that the output `OK` must remain true and save it. The SCADE™ project is actually a directory containing several files, among which the file containing the LUSTRE sample program (`Node2.saofd` in our case) plus many annotations in comments. We can simply replace this file with the EC file modified as described above, and reload the project in SCADE™.

8.6.2.1.5 **SMV.** SMV is a symbolic model checker, in the same family as LESAR and GBAC, except that it does not use LUSTRE as an input language. We use the free version, v10-11-02p46, which is limited in terms of functionalities. For example, it does not provide a SAT engine, whereas the commercial version does. We used SMV without command-line option.

8.6.2.2 Benchmarking the Tools on a Simple Platform

8.6.2.2.1 The Test Platform. We use a simple SystemC platform to test the performances of the different proof engines. The platform contains one BASIC channel, one master that will simply launch one transaction on the channel (Figure 8.10), one slave that will raise an error whenever it receives a transition (Figure 8.12) and a variable number n of transmitter (Figure 8.11). When a transmitter receives a transaction, the processing of the transaction notifies an event that wakes up a thread that will itself send a transaction to the next module. The master talks to the first transmitter, the ith transmitter ($i < n$) talks to the $i + 1^{\text{th}}$ one, and the nth transmitter talks to the slave.

```
1  void compute(void) {
2    while(true) {
3      initiator_port.write(m_addr, m_data);
4      while(true) {
5        wait(20, SC_NS);
6      }
7    }
8  }
```

Figure 8.10: Benchmark platform: Master Process

8.6.2.2.2 The Results. The results are given in table 8.1. “NT” means “not tested”, “OM” means the prover went out of memory before the end of the proof. Results of the form “> X” means the prover was killed at time “X”, with no success. The bold numbers correspond to the last n for which the proof engine answered in reasonable time.

The results are given in terms of user time, on a dual AMD Athlon(TM) MP 2000+ with 512Mb of RAM, running Linux.

```
1  tlm_status write(ADDRESS_TYPE addr_in,
2                   DATA_TYPE * source,
3                   int number,
4                   int port_id ,
5                   basic_metadata& metadata) {
6    tlm_status response;
7    e.notify();
8    response.set_ok();
9    return response;
10 }
11
12 void compute() {
13   tlm_status status;
14   while(true) {
15     wait();
16     status = initiator_port.write(m_addr, m_data);
17   }
18 }
```

Figure 8.11: Benchmark platform: Transmitter

```
1  tlm_status write(ADDRESS_TYPE addr_in,
2                   DATA_TYPE * source,
3                   int number,
4                   int port_id ,
5                   basic_metadata& metadata) {
6    ASSERT(false);
7    tlm_status response;
8    response.set_ok();
9    return response;
10 }
```

Figure 8.12: Benchmark platform: Slave Method

number of transmitters	1	2	3	4	5	6	7
PROVER time (prove)	>2h	NT	NT	NT	NT	NT	NT
NBAC time (no AA)	3.51	**81**	OM	NT	NT	NT	NT
PROVER time (debug)	11.9	**2,295**	12,797	> 5h	NT	NT	NT
GBAC time	0.13	0.64	4.76	**120**	> 1h	NT	NT
NBAC time	1.96	13	76	**296**	> 1h30	NT	NT
LESAR time	0.2	0.4	3.15	105	**4,647**	NT	NT
LESAR -forward time	0.26	0.79	4.63	113	**4,783**	NT	NT
GBAC -fb -sift time	0.2	0.76	1.95	4.4	10.5	36	**147**
GBAC -fb time	0.2	0.76	1.97	4.4	10.5	36	**191**
SMV time	0.46	1.46	4.82	16.6	84	360.82	41.5
counterexample size	34	60	86	112	138	164	190
SMV BDD nodes	74,888	138,103	294,448	483,183	977,629	2,588,187	1,183,562
LESAR BDD nodes	20,110	57,576	192,592	722,259	2,633,568	NT	NT

number of transmitters	8	9	10	11	12	13
PROVER time (prove)	NT	NT	NT	NT	NT	NT
NBAC time (no AA)	NT	NT	NT	NT	NT	NT
PROVER time (debug)	NT	NT	NT	NT	NT	NT
GBAC time	NT	NT	NT	NT	NT	NT
NBAC time	NT	NT	NT	NT	NT	NT
LESAR time	NT	NT	NT	NT	NT	NT
LESAR -forward time	NT	NT	NT	NT	NT	NT
GBAC -fb -sift time	OM	OM	OM	OM	NT	NT
GBAC -fb time	OM	OM	NT	NT	NT	NT
SMV time	67.75	150.21	697	1,648	**8,951**	13,691
counterexample size	216	242	268	294	320	246
SMV BDD nodes	1,396,510	1,863,109	2,839,374	4,243,537	7,433,346	14,964,410
LESAR BDD nodes	NT	NT	NT	NT	NT	NT

Table 8.1: Comparison of Proof Engines: Results of the Benchmark

8.6.2.2.3 Conclusions About the Proof Engines. Let's first get rid of the hopeless solutions.

PROVER PLUG-IN™ for SCADE™ gives surprisingly bad performances. It is known as a good SAT solver, and wouldn't have been chosen by Esterel Technologies if it gave so bad results in the general cases, but we are clearly in a case where it is awfully bad. In proof mode, we killed it after one hour and a half for the smallest example on which all other tools answered in less than 4 seconds! In Debug mode, it gave a few results, but several orders of magnitude worse than all other tools, and it would anyway not be applicable on correct platforms, since the only termination condition for the debug algorithm is when a counter-example is found.

When used with the abstract address encoding in LUSSY, NBAC gives bad but acceptable results. Note that the platform we use does not contain integer values, so, we do not take advantage of NBAC's strength, but use it in a case for which hasn't been optimized. No big surprise, then ... If we do not use the abstract addresses encoding, then, NBAC will perform its own abstractions on the addresses. Since the abstraction used (based on polyhedra) is more precise and more costly than our encoding, the bad results obtained are not surprising either. This solution is therefore more precise but clearly hopeless in the case of realistic platforms (even the biggest platform tried in this benchmark, with $n = 13$, can still be considered as "small" compared to real-life designs).

Then comes LESAR. The enumerative strategy (the default) give results comparable to the results of the symbolic forward strategy (`-forward` option). LESAR allowed us to go further than all the above-mentioned tools. GBAC, with the `-f` option, uses algorithms similar to LESAR and gives comparable results.

Remark:	We didn't mention the results using `lesar -backward` since `-forward` is obviously better: our HPIOM encoding results in a LUSTRE program with a small set of initial sets, a huge amount of by-construction unreachable states (any state where more than one variable encoding a control point is true for example), and therefore a huge amount of trivially unreachable bad states. The backward strategy will therefore start with a large set of bad states, and iterate backward mainly in the set of unreachable states. The forward strategy will start with a smaller set of states, and iterate forward inside a very constrained set of reachable states. Even on the smallest example, `lesar -backward` doesn't find the counterexample after running more than an hour, i.e. 15,000 times the time it took for forward verification. The bad performance for the SAT engine may come from the same problem, since SAT has a global view of the state-space, and does not necessarily start from the initial states.

GBAC, when used with the option `-fb`, gives better results. The fact that BDDs are recalculated at each step in a simpler way dramatically improves the performances over LESAR and `gbac -f`. The results in terms of execution time are satisfying, but it quickly explodes in terms of memory. It would be interesting check GBAC's memory usage, to see whether the algorithm actually needs so much memory or GBAC has a memory leak problem. Using the `-sift` option slightly improves the performance, but did not allow us to prove a larger platform.

SMV gives the best results. It is slightly slower than the other tools for small systems, but is able to find the counter-example with $n = 11$ in less than one hour, and we even pushed it to $n = 12$ although the verification is really long in this case. Looking at the number of BDD nodes allocated, we can see that SMV seem to manage its BDD in a better way: for $n \geq 3$, the number of BDD nodes allocated by LESAR explodes quickly while SMV manages to keep the progression relatively linear. What's surprising is that the problem with $n = 7$ is solved faster, and allocating less BDD nodes than the one with $n = 6$.

Globally, we can say that SMV is the most adapted proof engine in our case.

8.7 Validating Real-Life Platforms

We have seen that none of the tools we tried were able to prove any property on the EASY platform. What's unfortunate is that EASY is meant to be an *example* of platform, and is still very small compared to real-life platforms. Since the complexity of the proof can be exponential in theory, and is in practice at least much worse than linear in terms of size of the platform, trying prove a platform ten or hundred times larger than a platform that we're not able to handle would require a machine or a proof engine more powerful than the ones we tried, by many orders of magnitude. This doesn't seem to be realistic. The only way to complete a proof with our approach would probably be to perform more abstractions, and get a simpler HPIOM model.

This conclusion may seem very pessimistic. However, this doesn't call into question the whole approach, but only means that using LUSSY *as-it-is* for formal verification of large designs *as a whole* is not realistic. Nevertheless, the techniques commonly used to scale up such as component by component verification, or abstraction techniques, would all need all the components of LUSSY. We implemented global and formal verification as a first application of the tool chain. To apply LUSSY on real-life platforms, there is still a lot to do. The next chapter will give more details on the reusable contributions of LUSSY.

Conclusion and Perspectives

Chapter 9

Conclusion

9.1 Context

We have presented the notion of Transaction Level Modeling (TLM), and the way it is implemented in SystemC in particular at STMicroelectronics. At this level, the platform is defined by a set of modules exchanging data through atomic sets called *transactions*. The details of the bus protocols are not modeled. TLM models can be written much faster than their Register Transfer Level (RTL) implementations, and also simulate much faster. They can be used for early development of embedded software, to help architects take decisions about the partitioning of the system, and as reference models for the validation of lower-level implementations.

In the design flow promoted by STMicroelectronics, the transaction level is one of the first models in a refinement flow towards gate level, layout, and finally the physical chip. This new level of abstraction really makes sense when created before the lower levels. Extracting a TLM model from the RTL implementation would be possible, although really hard to do efficiently and automatically, but it would not be as useful as a TLM model written prior to the RTL implementation, since the uses for the TLM model appear *before* the availability of the RTL in the design flow. This succession of levels of abstraction is an example of a model-driven methodology: the most abstract model is written first and gives a system-level view of the SoC. Implementation details are added afterwards. Comparing the entry point of the design flow (the algorithms level, or the Programmer View level, which is the most abstract of the flavors of TLM) with the final design is hardly feasible. There are so many changes in the architecture and the granularity of data and timing, that the first and the last steps of the design flow do not even seem to talk about the same thing. On the other hand, a step by step validation is feasible, although hard to do formally and exhaustively.

The notions of model-driven methodology and refinement have a wider range of application than Systems-on-a-Chip design. The ideas behind TLM (reusable components, abstract communication channels, early execution, etc.) are relatively new in the context of electronic design, but are extensively used in other domains of computer science.

TLM is only a concept. To be applicable in practice, this concept needs an implementation. We are particularly interested in the SystemC infrastructure for TLM, whose main advantage is to be a standard, built on top of other standard technologies (SystemC is a library for the C++ programming language). It was already used inside STMicroelectronics (using the abstract communication channels TAC and BASIC) before the beginning of the Ph.D, and is now used in several important projects inside and outside the company.

This new level of abstraction and new technologies raised the need for new tools to create a full development and verification environment. Among the problems to be solved, are the validation—formal or not—of transactional models, their test by simulation, and the understanding and formalization of the levels of abstraction. All this is really interesting only if carried out on real programs. The choice of SystemC allows us to use real case studies. This was the beginning of a co-operation between STMicroelectronics and the Verimag laboratory. We have laid the cornerstone, and other tools on related problems will come in the future.

9.2 Results and Discussion

9.2.1 Contributions of the Thesis

This document presented the works carried out during a Ph.D thesis, started in October 2005, and presented in December 2005. The thesis being the first one in co-operation between the « System Platform Group » (SPG) team of STMicroelectronics and the « Synchronous Languages and Reactive Systems » team of Verimag, an important part of the work has been to understand the context, and to identify a precise subject.

We quickly realized that treating SystemC and the additional TAC and BASIC components to model abstract communication channels was mandatory to keep up with the industry, and in particular with STMicroelectronics developments.

The concrete contribution of this thesis is the toolbox LUSSY, a set of building blocks for the manipulation of transactional models of Systems-on-a-Chip, written in SystemC. With the SMV and the LUSTRE back-ends, it is able to formally verify properties on a TLM model. LUSSY is the composition of a SystemC front-end (PINAPA), a semantic extractor (BISE), an optimizer (BIRTH) and several code generators (LUSTRE, SMV, and DOT back-ends). The general approach was presented in the first part of this document. The next parts presented the components one by one.

The intermediate formalism used in LUSSY, is a simple interpreted automata format called HPIOM, cornerstone between the extraction part and the back-ends. It was presented in section 5. The key point of HPIOM is that it is both a formal and an executable formalism. Extracting an HPIOM model from a SystemC program is therefore a way to give an executable semantics to SystemC. The correctness of the translation can not be ensured formally since there is no official formal semantics for SystemC, but comparing HPIOM and SystemC executions allows to get a good confidence in it.

Although HPIOM has been created to answer one precise need (to have an intermediate formalism between SystemC and the proof engines in LUSSY), it appears to be relatively generic, and benefits from the expressive power of the synchronous languages. One can design deterministic or non-deterministic behaviors with HPIOM. It is easy to ensure that an HPIOM automaton is deterministic, and non-determinism is also easy to express. In any case, it remains executable. Non-deterministic constructs will simply be converted into inputs of the system.

HPIOM can be used in many other contexts. We have seen in chapter 8 that using a formalism like HPIOM with a notion of control-flow allows some optimizations that could hardly be done on a completely data-flow oriented format. HPIOM could therefore be used as an alternative to LUSTRE in some places of the Verimag tool chain. Any tool using HPIOM can automatically benefit from the optimizations implemented in BIRTH. Some practical problems are being solved to make HPIOM completely independent from LUSSY, and give it a textual syntax.

LUSSY is designed to have as few limitations as possible. PINAPA, the front-end, presented in chapter 4, uses a novel approach to extract all the information from the source program. It is based on a real C++ front-end, which avoids creating a syntax different from the official one. Where some other tools consider that the architecture information must be "statically known", we actually execute the elaboration phase of the model to analyze, using a slightly modified version of the SystemC official library. The link between the architectural and the syntactical information may be problematic if dynamic data-structures are used to manipulate SystemC objects during simulation, which is anyway forbidden by most SystemC coding guidelines. The cases where PINAPA's approach fails are cases where any other SystemC front-end would fail. As opposed to this, PINAPA can successfully treat programs with a complex elaboration phase, where all other SystemC front-ends we are aware of would fail. One contribution of PINAPA is simply a clarification of the notions of static and dynamic information for SystemC.

BISE, extracting the HPIOM model, presented in chapter 6, has to perform a few abstractions, but does not abstract more than needed. The abstractions are conservative for safety properties. It provides a model for the most common C++ and SystemC features, and takes into account the specificities of the SystemC scheduler (immediate and delayed notifications, notions of δ-cycle and time elapse, ...). It also has a model of TAC and BASIC channels. Other SystemC constructs didn't appear in the example on which we tried our tool, but could easily be added.

Specifying properties in BISE does not require a new language. The properties can be directly expressed in C++, or the user can choose among a set of generic properties built in BISE.

We implemented some transformations of the generated HPIOM in the component BIRTH, as described in chapter 7. Most are well-known optimizations available in optimizing compilers, adapted to our automaton formalism. We implemented live variable analysis, minimization of the numbers of inputs necessary to model non-determinism and a simple automatic parallelization algorithm. Each optimization improved the performance on the prover side (great improvement when using LESAR as a prover, but less so with SMV). Other transformations are here to ease the job of the back-ends. For example, we expand abstract addresses and non-deterministic choices into simpler equivalent. We also make a few experiments with non-conservative approximations.

Our LUSTRE and SMV back-ends allowed us to prove some properties on small enough platforms, as discussed in chapter 8. Unfortunately, we encountered state explosion before reaching the size of the EASY platform, which is representative of the type of code people write in real platforms, but much smaller. It doesn't seem to be reasonable to try to formally verify complete real-life platforms as a whole using *only* the techniques implemented in LUSSY. The two most common approaches to verify large systems are to perform more abstractions, or to spit the proof and prove components one by one. In both cases, the first step is . . . to be able to prove something! In other words, the basic block for a component-oriented prover or a more elaborated abstract interpreter is exactly what LUSSY does.

Although this was not the initial goal, experimenting with LUSSY and its LUSTRE and SMV back-ends provided an interesting benchmark for the LUSTRE tool-chain. Verimag's new model-checker prototype, GBAC, seems promising and gives much better results than LESAR. However, SMV is the fastest on the platforms we tried, as soon as the platform is non-trivial.

The components of LUSSY have been designed to be reusable. PINAPA, in particular, is distributed publicly as a free software, and promoted by the company GreenSoCs (see `http://greensocs.sourceforge.net/pinapa/`). It already has several uses inside and outside STMicroelectronics. BISE allows traditional formal methods to be applied on a simple model, HPIOM. HPIOM itself can, and will probably be reused outside the scope of LUSSY. Only the code generators for LUSTRE and SMV are specific to formal verification by model-checking and abstract interpretation. Together, the components of LUSSY form the basis to build other tools for the analysis of TLM models of SoCs. They are the foundation for a more durable co-operation between Verimag and STMicroelectronics, and possibly other partners.

Indeed, since the beginning of the Thesis, two other Ph.D students (Claude Helmstetter and Jérôme Cornet) joined the project. One subcontractor (Frédéric Saunier) employed by Silicomp is currently working in the SPG team of STMicroelectronics to continue developments related to PINAPA. A SPIRIT generator has been started and already allows the visualization of the architecture of simple platforms. The development of a lint tool based on PINAPA is planned. The co-operation between Verimag and STMicroelectronics also involves another project, between different teams, and is being officialized in a joint laboratory. Projects involving other partners are also being set up. In INRIA Rhône-Alpes (France), a project named SC2PROM (carried out by Yvan Roux), similar to LUSSY has been started. It reuses PINAPA as a front-end, and contains a component similar to BISE, targeting a different intermediate formalism. Although no code has been reused from BISE itself, the publications describing it and some informal advices, based on the experience of writing BISE, reduced the effort for this project. A pre-competitive RNTL (National Network of research and innovation in Software Technologies) project has been submitted, aiming at developing an open-source infrastructure for the development of transaction level models. It involves new partners, namely Silicomp-AQL and Certess. A Minalogic project is also to be submitted (Minalogic is a competitiveness center dedicated to the development of software for micro and nanotechnologies).

The scientific content of the Ph.D has also been acknowledged externally. Two publications have been accepted in international ACM conferences ([MMMC05a] and [MMMC05b]). The "formal verification" section of the book "Transaction-Level Modeling with SystemC. TLM Concepts and Applications for Embedded Systems" [Ghe05] has also been written as part of this thesis, as well as an article in the "Special Section on Hardware/Software Codesign and System Synthesis" of the journal "IEEE Transactions on VLSI Systems".

9.2.2 The Choice of SystemC

The choice of SystemC to write TLM platforms seems controversial since frameworks designed with formal methods in mind already exist for the same purpose. Working formally on SystemC gives an interesting point of view on its qualities and flaws. SystemC has mainly been designed with practical concerns in mind, and is often criticized regarding its weak theoretical foundation. After three years studying the details of SystemC mechanisms, and discussing with the developers and the users of TLM components in STMicroelectronics, it's time to weigh up the pros and the cons. The two questions we are trying to answer here are: "As a research institute, did Verimag do the right choice studying SystemC?" and "Did STMicroelectronics do the right choice with SystemC as an infrastructure to implement transaction-level models?".

SystemC is based on the C++ language. It is very intrusive in the user's source code, uses both advanced features of C++ and preprocessor's macros in the API. It would therefore hardly be possible to write a binding for the library for any other language. This means that SystemC forces the use of C++. C++ is a very powerful language, but it is also a very *unsafe* language, in the sense that many dangerous constructs are not forbidden by the language, and give unexpected, or even undefined behavior at run-time[1]. The C++ language is also very complex and therefore difficult to learn.

SystemC is not a real *language*. It is a library designed in a way the user has the feeling that he is programming in a new language. There are very good reasons to design SystemC in such a way, but it also presents a number of flaws compared to a dedicated language with a dedicated compiler: many verifications that could be done statically have to be done at run-time, or can not be done at all.

The fact that SystemC is not a language also makes the task harder for people writing analyzers for it. A parser for a new language could have been written using usual compilation techniques. Writing a SystemC front-end is a much harder problem. PINAPA solves it in a relatively elegant way, but still has limitations in some cases. Other approaches have more intrinsic limitations. Research works carried out on SystemC is therefore also made more difficult, since manipulating programs would clearly be much easier in a clean and simple language, with only the necessary constructs.

From a theoretical point of view, the approach followed by languages like SpecC, with a new language, a new grammar, and new tools is therefore much superior. A framework like Metropolis, designed with formal methods in mind, would seem to be the ideal case. But in practice, the approach followed by SystemC has much more success in the industry. A naive conclusion would be that the industry made the wrong choice, but let's try to go a bit further ...

Although imperfect, SystemC meets the requirements of the users very well. For software developers, SystemC enables early execution of the embedded software, by using abstract models of the hardware. Furthermore, those abstract models are not only fast to write, but also fast to execute. For verification engineers[2], SystemC is a way to write reference models. Tools like Cadence NC-SystemC allows the execution of an heterogeneous system, comprising both a SystemC part and RTL components. They see in SystemC (and optionally the SystemC verification library) a cheap replacement for proprietary languages like the *e* language from Verisity, allowing them to gain independence from CAD vendors.

The first question when adopting a new technology is "What can I do with it?". But other issues are also key points: "How much will it cost?", "If I write some code today, will I still be able to use it tomorrow?", "Which additional services can I get?". The fact that SystemC is based on open source technologies and on established standards is crucial regarding these points. The most basic tools (to be able to write, compile, and execute a model) are provided at no cost. Using open source tools ensures their durability. If the current maintainers of the SystemC library cease the development, in the worst case, one can pay another company to continue the maintenance. The C++ language itself and its associated tools is more than unlikely to disappear in the next 10 or 20 years. Regarding the availability of tools, using C++ is clearly a very good choice. One can find plethora of IDE, compilers, debuggers, profilers, and libraries available for this language. SystemC itself being standardized and open source allows multiple companies to provide tools for it without depending of a central authority. Many CAD vendors provide development tools for SystemC programs today.

[1] The inquisitive reader can have a look at my collection of buggy C and C++ programs located at `http://www-verimag.imag.fr/~moy/c_collection/` if he isn't convinced yet!

[2] the meaning of "verification" in "verification engineer" is the one of the hardware community, without the "formal verification" connotation of the software community.

The learning curve is relatively slow, due to the complexity of C++. Even with a good knowledge of the C language, learning the C++ language and its “best practice” (which are very different from what is expected from a good C programmer) is still long. On the other hand, many programmers already know C++, and since this knowledge is reusable outside the SystemC world, many programmers are willing to learn.

With the experience of several years of development, taking all these factors into account, it appears that using SystemC to develop transactional models in STMicroelectronics was the only reasonable choice. We strongly believe that the industry is going in this direction.

From the research point of view, with the preoccupation of performing a real applied research work, the conclusion is rather obvious: we had to work with the same framework as the industry does. Asking engineers to throw away their work and learn something new, from scratch, has rarely been a successful approach in the history of computer science.

Chapter 10

Perspectives

10.1 Possible Improvements and Uses for LUSSY

One of the best way to scale up using automatic verification techniques is to make a clever use of abstractions. We already implemented a simple abstraction taking advantage of the particular nature of addresses, and a complete abstraction of numerical data.

We have seen that abstractions like the one we used for abstract addresses lose some information for two reasons: the encoding itself can not represent all the information from the source code, and at the time we perform the abstraction, we may not have all the information to generate the optimal encoding. It would be interesting to perform a static analysis automaton per automaton, to get as much information as possible about the possible values of variables, and generate a better encoding based on this information, to ease the global proof. For example, NBAC can compute an over-approximation of the set of possible values for a variable of type address (the typical case would be an access to an array, where the index is a loop variable. The current abstract address encoding can not do much about it, but an abstract interpreter could easily give the interval of the possible values for the array index).

Another commonly used scaling up technique is the verification component by component. Before being able to compose the proofs of individual components, it is necessary to be able to perform the individual proofs, which is what LUSSY does (since SystemC components are hierarchical, there is not much difference between a component and a platform). To be able to scale up, we need to add a "component" layer to LUSSY, that would split the platform, either at the SystemC or at the HPIOM level, and that would use the current features of LUSSY to perform the proofs on the individual components, and then to compose the results of the proofs.

Interaction between a proof engine and a traditional debugger can lead to very efficient debugging tools. There are a number of problems to solve to make the output of LUSSY directly replayable on SystemC, or at least more user-friendly. Another necessary feature would be the ability to provide the initial state for the proof based on the state of the program at a given point of the simulation. This would allow the integration of LUSSY in a traditional debugger like GDB. The user would for example start the simulation manually, and launch a proof to see if he can find a bug starting from the current state, or asking the model-checker how to go to a particular state. Optionally, he could provide a limit like "in the current δ-cycle" or "before N processes are executed from now", to make sure the proof terminates in reasonable time.

The combination of a proof engine with an automatic test environment is another powerful application of proof techniques. The diagnosis from the proof engine is actually a super-set of the traces leading to a state violating the property. Similarly to the approach described for debugging, by giving a test objective as a property to prove, we get a way to guide the program to the test objective. Depending on how precise the diagnosis and the test objective are, the program may or may not reach it, but is anyway more likely to do so than it would be with random simulation.

10.2 Towards a Complete Development Environment for SystemC

The general objective of the co-operation between STMicroelectronics, Verimag, and more recently other partners such as Silicomp-AQL (through the OpenTLM project) is to provide a complete development and verification environment for Transaction Level Models written in SystemC. LUSSY provided the building blocks and an application to formal verification, but the work is far from being finished.

We have experimented with formal verification in LUSSY. As opposed to this, the current validation techniques used in production are simple executions, with a relatively basic oracle. These run-time methods could be greatly improved. The works carried out by Claude Helmstetter aim at improving the coverage of the test-benches while avoiding redundant test as much as possible. It is based on an instrumented simulation detecting suspicious executions at run-time. The algorithms are currently designed, and are partially implemented, but require an instrumentation of the platform to analyze. This manual instrumentation could be made automatic with a tool able to parse the platform and insert the necessary pieces of code in the right places. PINAPA can be a very helpful basis for such automatic instrumentation tool. The run-time algorithms could also be improved by working in combination with a static code analyzer to reduce false positive when several processes access a shared resource. The other components of LUSSY can be helpful for this purpose.

Some work is still to be done to get a better understanding of the timed and untimed layers inside the TLM level of abstraction, and improve the methodologies to manage timing in TLM platforms accordingly. Jérôme Cornet started working on the subject in 2004, and already developed abstract models, similar to timed automata, to represent the possible executions of a timed system. The final goal is to have a methodology, supported by development and possibly validation tools to add the “timing” layer to a pure functional model, avoiding modifications of the original functional model, and ensuring that no bugs are added during the process.

One of the limitations of our approach is that model-checking does not work on very complex code (typically, code using dynamic data-structures), and other techniques such as abstract interpretation would not scale up either. The typical example is the case of an instruction set simulator: to execute a simple statement in the embedded software, an instruction set simulator interprets the machine code which itself is the result of the compilation, that loses all the structural information from the source code that would have been crucial for the proof. A platform containing an instruction set simulator can therefore not be proved directly. The components that are too complex to be modeled have to be abstracted, either by replacing them by completely non-deterministic ones, or by replacing them by local specifications, such as a *contract*. The idea of design by contract was introduced a long time ago by Bertrand Meyer [Mey92] in the Eiffel programming language.

The first application of the notion of contract is the run-time checking of the validity of the contract. From this point of view, contracts are simply an organized way to write assertions in the source code.

But contracts can also have applications in static verification. It allows a better partitioning of the proof: one can prove that each component meets its contract, individually, and then replace some components by their contract in system-wide proofs. This is comparable to the works of Yvan Roux on SC2PROM, except that in this case, the local specification is not *proved* locally, but *computed* automatically. We have been contacted by Edmund M. Clarke (Carnegie Mellon University) and started some discussion to build a bridge between LUSSY and the tools they developed for compositional and incremental verification (see for example [SCCS05]).

Contracts can also be seen as a non-deterministic abstraction of the actual implementation. It can be written before the implementation, and is normally much faster to write. By replacing non-determinism by randomization or additional inputs, this specification can become executable. SystemC and TLM are already allowing the early execution of a system. A non-deterministic local specification such a contract can allow going one step forward, and allow the execution of the platform without even having all components implemented.

HPIOM is a candidate formalism for the expression of local specifications for all kind of components. It has all the necessary to model hardware components, and can also model software components, as long as it is acceptable to abstract dynamic constructs.

Appendixes

Appendix A

Example of translation into HPIOM: `switch` and `while` statement

Contents

A.1 Introduction **155**

A.2 The platform **155**

A.3 C++ constructs **157**

A.3.1 `while` Loop 157

A.3.2 `switch` Statement 158

A.4 SystemC Constructs **162**

A.4.1 Processes and the Scheduler 162

A.4.2 Communication: the Example of the `sc_signal` 165

A.5 Global Picture **165**

A.5.1 Main Files 165

A.1 Introduction

Chapter 6 presented the theoretical translation. This section presents the results and the automata produced on an example to illustrate some details of the translation of SystemC into HPIOM.

The platform used as an example is not meant to be representative of a real platform, even trying to prove any property on it is irrelevant. It was chosen because the automata generated are readable and are a good support for the explanation of the translation of SystemC into HPIOM. Note that all the pictures of automata provided in this section are automatically generated, using our visualization back-end (as presented in section 3.3.3.5).

This representation is not exhaustive. Assignments and guards, for example are not displayed. Transitions and states have names (which have no influence on the semantics). When a transition is the only one leaving from a state, it is displayed in bold. Transitions whose guard is a default condition are displayed dashed.

A.2 The platform

The platform is made of two modules, communicating together with a `sc_signal`. The source code is provided in Figure A.1.

```
1   #include "systemc.h"
2   #include <iostream>
3   #include "scp-verification.h"
4   using namespace std;
5
6   SC_MODULE(module_switch) {
7     sc_in<bool> in;
8
9     void compute() {
10      bool b1, b2, b3;
11      switch(in.read() == true ? 1 : (b1 ? 2 : 3)) {
12      case 1:
13        next_trigger(20, SC_NS);
14        break;
15      case 2:
16        ASSERT(b1 == true);
17      case 3:
18        b1 = true;
19        break;
20        b2 = false;
21      default:
22        b3 = true;
23      }
24    }
25
26    SC_CTOR(module_switch) {
27      SC_METHOD(compute);
28      sensitive_pos << in;
29    }
30  };
31
32  SC_MODULE(module_while) {
33    sc_out<bool> out;
34
35    void compute() {
36      int x = 0;
37      while (x++ <= 42) {
38        wait (20, SC_NS);
39        if (x != 13)
40          out.write(true);
41      }
42    }
43
44    SC_CTOR(module_while) {
45      SC_THREAD(compute);
46    }
47  };
48
49  int sc_main (int argc, char ** argv) {
50    sc_signal<bool> sig;
51    module_switch mod_switch("switch");
52    module_while  mod_while("while");
53    mod_switch.in(sig); mod_while.out(sig);
54    sc_start();
55  }
```

Figure A.1: Example of SystemC Program: Main Function

A.3 C++ constructs

A.3.1 `while` Loop

A.3.1.1 Generated Automaton

The algorithm for the generation of automaton for the `while` loop has been described in section 6.2.2. Figure A.2 shows the generated automaton for the process `module_while::compute` declared line 35. Plain arrows are normal edges. Arrows are bold when they are the only outgoing edge from a control point, and dashed arrows are for edges whose guard is a default condition.

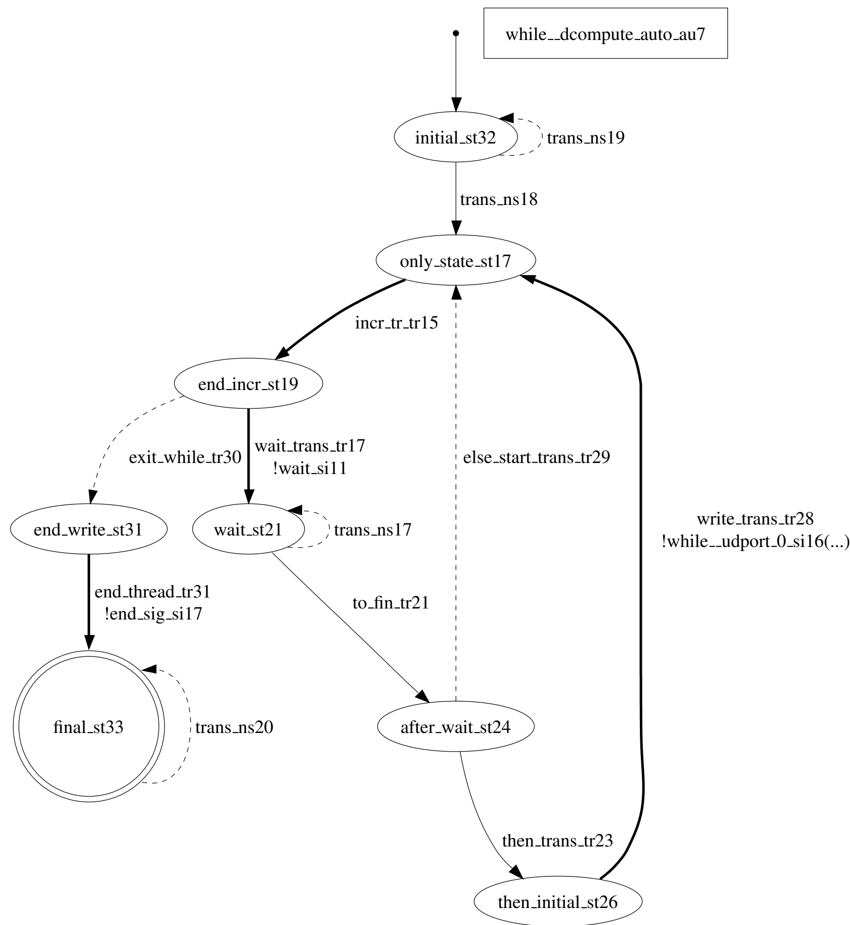


Figure A.2: Translation of a `while` loop

A.3.1.2 Generated LUSTRE Code

The generated LUSTRE code is provided in Figures A.3, A.4 and A.5. The automaton is transformed into a LUSTRE node taking as inputs the input signals emitted by other automata (with the suffix `_siNN` and the unknown values used in this automaton (`wu_timewhile__dcompute_un2`). The outputs are the output signals. We declare one statefull variable per state and per HPIOM variables, and one stateless variable per transition.

The process computes the condition `++x <= 42` on transition `incr_tr_tr15`. This increment appears line 79 in the generated code. Then, the test to choose between entering the loop (`wait_trans_tr17`, line 25) and exiting the loop (`exit_while_tr30`, line 35) is performed in state

```
1   node while__dcompute_state_au1 (time_elapse_si10: bool;
2                                   elect_si2: bool;
3                                   end_sig_si17: bool;
4                                   wait_si11: bool;
5                                   wu_timewhile__dcompute_un2: bool)
6   returns (eligible_0_st3, run_0_st5, sleeping_wait_st22, ended_st34: bool;
7            wait_si4: bool);
8   var
9     election_trans_tr1: bool;
10    trans_ns13: bool;
11    trans_ns14: bool;
12    wait_trans_tr18: bool;
13    wake_up_tr19: bool;
14    trans_ns15: bool;
15    trans_ns16: bool;
16    end_trans_tr32: bool;
17
```

Figure A.3: Generated LUSTRE code for a `while` loop (header)

`end_incr_st19`. The `wait` statement line 38 corresponds to state `wait_st21`, and the `if` statement is executed between state `after_wait_st24` and `only_state_st17`.

A.3.1.3 Generated SMV Code

As a comparison, we provide the generated SMV code too. Figure A.6 shows the header of the module, while Figure A.7 gives the body. The encoding of the state machine is much more straightforward than the LUSTRE version. The module contains one SMV switch statement, with one case for each control point of the automaton, and an `if - then - else` statement to choose between the outgoing transitions of this state.

HPIOM variables and states are assigned with the `next` construct (line 30 in Figure A.7 for example), while signals are instantaneous values emitted on transitions (line 37 for a pure signal, 64 and 65 for a valued signal). If another automaton needs use the `in_state` conditional on a state, the presence of the automaton in the state is exported through a boolean SMV variable, like line 30 in the example.

A.3.2 `switch` Statement

The process `module_switch::compute` declared line 9 is transformed into the automaton of Figure A.8. We can see the `switch` statement itself on state `only_state_st35`, the `case` statements in the states `to_label_stNN`. State `to_break_st57` corresponds to the piece of dead code line 20.

A.3.2.1 Generated LUSTRE Code

We won't give the complete LUSTRE code here, but the portions of Figure A.9 illustrate some interesting concepts that didn't appear in the transformation of the `while` loop:

Lines 9 to 20 show the equations for the outgoing transitions from state `only_state_st35`. Since this state has more than one transition with a non-default condition, it may have been the case that more than one transition is enabled at a time. LUSSY has therefore added a non-deterministic choice to those transitions to make this choice unique. The choice has to be done between `switch_jmp_tr65`, `switch_jmp_tr62` and `switch_jmp_tr63`, and `switch_jmp_tr64` will be taken if none of the other is enabled. To chose between 3 transitions, we need $\lfloor \log_2(3) \rfloor = 2$ Boolean variables (`make_choice_unique_nd11_un16` and `make_choice_unique_nd11_un17`).

```
18  let
19
20    -- Transitions:
21    incr_tr_tr15 = false -> (pre(only_state_st17) and true);
22                          -- > end_incr_st19
23    wait_trans_tr17 = false -> (pre(end_incr_st19) and
24                       (((pre x_val_val0) + 1) <= 42));  -- > wait_st21
25    trans_ns17 = false -> (pre(wait_st21) and (not elect_si2));
26                          -- > wait_st21
27    to_fin_tr21 = false -> (pre(wait_st21) and elect_si2);
28                          -- > after_wait_st24
29    then_trans_tr23 = false -> (pre(after_wait_st24) and
30                       ((pre x_val_val0) <> 13));  -- > then_initial_st26
31    write_trans_tr28 = false -> (pre(then_initial_st26) and true);
32                          -- > only_state_st17
33    else_start_trans_tr29 = false -> (pre(after_wait_st24) and
34             (not ((pre x_val_val0) <> 13)));  -- > only_state_st17
35    exit_while_tr30 = false -> (pre(end_incr_st19) and
36             (not (((pre x_val_val0) + 1) <= 42)));  -- > end_write_st31
37    trans_ns18 = false -> (pre(initial_st32) and elect_si2);
38                       -- > only_state_st17
39    trans_ns19 = false -> (pre(initial_st32) and (not elect_si2));
40                       -- > initial_st32
41    end_thread_tr31 = false -> (pre(end_write_st31) and true);
42                       -- > final_st33
43    trans_ns20 = false -> (pre(final_st33) and true);
44                       -- > final_st33
45
46    -- States:
47    only_state_st17 = false ->
48      (write_trans_tr28 or else_start_trans_tr29 or trans_ns18);
49    end_incr_st19 = false ->
50      (incr_tr_tr15);
51    wait_st21 = false ->
52      (wait_trans_tr17 or trans_ns17);
53    after_wait_st24 = false ->
54      (to_fin_tr21);
55    then_initial_st26 = false ->
56      (then_trans_tr23);
57    end_write_st31 = false ->
58      (exit_while_tr30);
59    initial_st32 = true ->
60      (trans_ns19);
61    -- final
62    final_st33 = not(only_state_st17 or end_incr_st19 or
63                     wait_st21 or after_wait_st24 or
64                     then_initial_st26 or end_write_st31 or
65                     initial_st32);
```

Figure A.4: Generated LUSTRE code for a `while` loop (transitions and states)

```
66      -- Pure signals
67      end_sig_si17 = false -> end_thread_tr31;
68      wait_si11 = false -> wait_trans_tr17;
69
70      -- Valued Signals
71      while__udport_0_si16_present = false -> write_trans_tr28;
72      while__udport_0_si16_value = true ->
73        if write_trans_tr28 then  true else true;
74
75      -- Continuous signals
76
77      -- Variables
78      x_va1_va10 = 0 -> (
79        if incr_tr_tr15 then (((pre x_va1_va10) + 1)) else
80           pre (x_va1_va10));
81    tel.
```

Figure A.5: Generated LUSTRE code for a `while` loop (signals and variables)

```
1     module while__dcompute_auto_au7(elect_si2, wait_si11, end_sig_si17,
2                                     while__udport_0_si16_present,
3                                     while__udport_0_si16_value) {
4     INPUT elect_si2: boolean;
5     OUTPUT wait_si11: boolean;
6     OUTPUT end_sig_si17: boolean;
7     OUTPUT while__udport_0_si16_present: boolean;
8     OUTPUT while__udport_0_si16_value: boolean;
9       -- States:
10    state : {only_state_st17, end_incr_st19, wait_st21,
11             after_wait_st24, then_initial_st26,
12             end_write_st31, initial_st32, final_st33};
13      -- Variables:
14    removed_int_un11 : boolean;
15    removed_int_un12 : boolean;
16    init(state) := initial_st32;
17
18    default {
19      wait_si11 := 0;
20      end_sig_si17 := 0;
21      while__udport_0_si16_present := 0;
22      while__udport_0_si16_value := 0;
23    } in
```

Figure A.6: Generated SMV code for a `while` loop

```
24    -- States:
25  {
26  switch(state) {
27
28  only_state_st17: {
29    if (1) {
30      next(state) := end_incr_st19;
31    }
32  }
33
34  end_incr_st19: {
35    if (removed_int_un11) {
36      next(state) := wait_st21;
37      wait_si11 := 1;
38    }
39    else    {
40      next(state) := end_write_st31;
41    }
42  }
43
44  wait_st21: {
45    if (elect_si2) {
46      next(state) := after_wait_st24;
47    }
48    else    {
49      next(state) := wait_st21;
50    }
51  }
52
53  after_wait_st24: {
54    if (removed_int_un12) {
55      next(state) := then_initial_st26;
56    }
57    else    {
58      next(state) := only_state_st17;
59    }
60  }
61  then_initial_st26: {
62    if (1) {
63      next(state) := only_state_st17;
64      while__udport_0_si16_present := 1;
65      while__udport_0_si16_value := 1;
66    }
67  }
68
69  end_write_st31: {
70    if (1) {
71      next(state) := final_st33;
72      end_sig_si17 := 1;
73    }
74  }
75
76  initial_st32: {
77    if (elect_si2) {
78      next(state) := only_state_st17;
79    }
80    else    {
81      next(state) := initial_st32;
82    }
83  }
84
85  final_st33: {
86    {
87      next(state) := final_st33;
88    }
89  }
90  }
91  }
92  }
```

Figure A.7: Generated SMV code for a `while` loop (transitions and states)

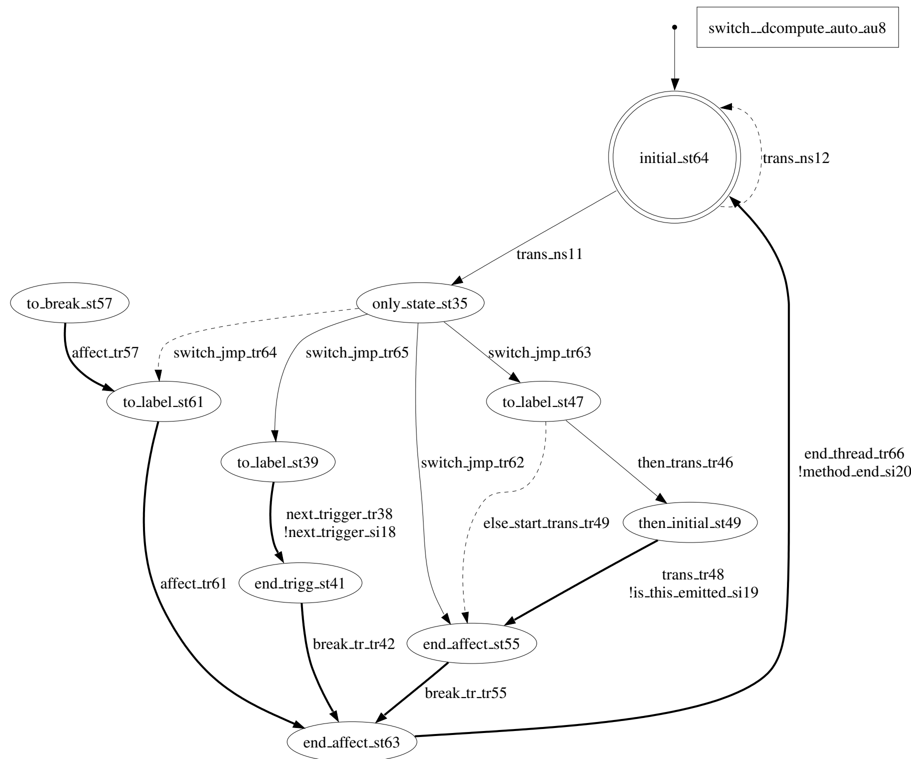


Figure A.8: Translation of a `switch` statement

A.3.2.2 Generated SMV Code

We provide the equivalent section in the SMV code for comparison: Figure A.10. The imperative style makes it a bit more readable, and the boolean log encoding of the choice is replaced by a native SMV non-deterministic variable, with three values.

A.4 SystemC Constructs

This section will present the generated automata for a few SystemC constructs.

A.4.1 Processes and the Scheduler

A.4.1.1 SystemC `SC_THREAD` s

We have seen that BISE generates one automaton for the control flow of each process, and an additional one to model the state of each process in the scheduler. The automaton representing the state of `while.compute` is given in Figure A.11. The eligible/run/sleep loop appears clearly, and we see the additional sink state for the process termination.

A.4.1.2 SystemC `SC_METHOD` s

In the case of `SC_METHOD` s, the "running" and "sleep" states are split into several states. The `next_trigger` functions switches from one state to the other.

```
1  node switch__dcompute_auto_au8 ([...]) returns ([...]);
2  var
3
4    [...]
5
6  let
7    -- Transitions:
8    [...]
9    switch_jmp_tr62 = false -> (pre(only_state_st35) and
10     (removed_int_un8 and
11      ((make_choice_unique_nd11_un16 = false) and
12       (make_choice_unique_nd11_un17 = false))));  -- > to_label_st39
13   switch_jmp_tr63 = false -> (pre(only_state_st35) and
14     (removed_int_un9 and
15      ((make_choice_unique_nd11_un16 = true) and
16       (make_choice_unique_nd11_un17 = false))));  -- > end_affect_st55
17   switch_jmp_tr64 = false -> (pre(only_state_st35) and
18     (not [...]));  -- > to_label_st61
19   switch_jmp_tr65 = false -> (pre(only_state_st35) and
20     (removed_int_un10 and (not ([...]))));  -- > to_label_st47
21
22   [...]
23
24   -- Pure signals
25   next_trigger_si18 = false -> next_trigger_tr38;
26   method_end_si20 = false -> end_thread_tr66;
27   is_this_emitted_si19 = false -> trans_tr48;
28
29   [...]
30
31 tel.
```

Figure A.9: Portions of the generated LUSTRE code for `module_switch::compute`

```
1   make_choice_unique_nd11 : {choose_this_one_nd12, choose_this_one_nd13,
2                               choose_this_one_nd14};
3
4   [...]
5
6     only_state_st35: {
7       if ((removed_int_un8 &
8            make_choice_unique_nd11 = choose_this_one_nd12)) {
9         next(state) := to_label_st39;
10      }
11      else if ((removed_int_un9 &
12                make_choice_unique_nd11 = choose_this_one_nd13)) {
13        next(state) := end_affect_st55;
14      }
15      else if ((removed_int_un10 &
16                make_choice_unique_nd11 = choose_this_one_nd14)) {
17        next(state) := to_label_st47;
18      }
19      else    {
20        next(state) := to_label_st61;
21      }
22    }
```

Figure A.10: Portions of the generated LUSTRE code for `module_switch::compute`

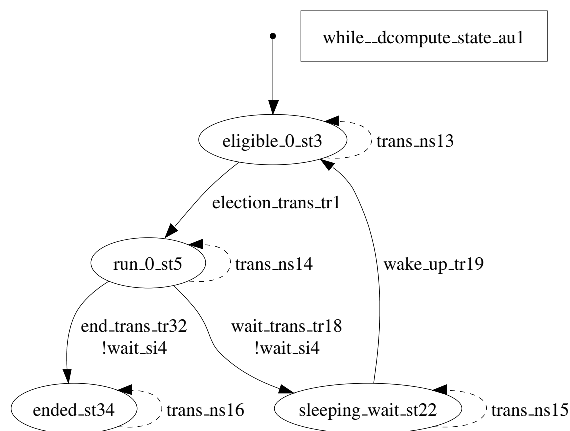


Figure A.11: Process State of an `SC_THREAD`

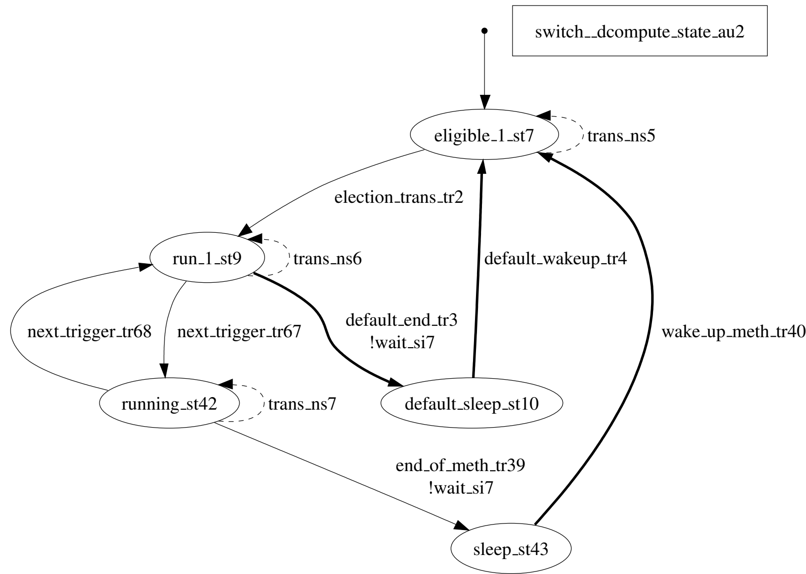


Figure A.12: Process State of an `SC_METHOD` with Several `next_trigger` Statements

A.4.1.3 The Scheduler

The automaton for the scheduler is given in Figure A.13. We see the three nested loops: execution phase, with the choice of the eligible process from state `selecting_process_st14` to `process_running_st12`, the delta-cycle corresponding to the loop including one evaluation phase and one update phase (`update_delta_st13`, `end_of_update_tr7`. The time elapse loop is the last one, including state `time_elapse_state_st16`.

A.4.2 Communication: the Example of the `sc_signal`

The generated dot automaton for the `sc_signal` is provided in Figure A.14. As it is, it is not very interesting since it contains only one state: the behavior is only managed with variables of the automaton.

The LUSTRE code is more interesting (Figure A.15). We see one variable for the current value (`curr_val_va2_va7`) and one variable for the next value (`next_val_va3_va8`). The current value is updated on transition `write_tr24` which is itself triggered by the input signal `input_si12_present`. The scheduler emits the signal `update_si9` when entering the update phase. Depending on the equality of `curr_val_va2_va7` and `next_val_va3_va8`, we choose between `req_up_void_tr25` and `req_up_action_tr26` (emitting `event_si13`) when we receive this signal.

A.5 Global Picture

The connections between the automata of the system is illustrated by picture A.16.

We can see the scheduler, with a rather central position, the two pairs of automata of the processes, the signal, plus automata like `sig_input_or_au6` whose role is only to perform combinational operations on signals (a logical or in this case).

A.5.1 Main Files

The main file instantiates the components. Each component is instantiated once and only once. If the same component appears several times in the system, it will be generated several times. We do not use the hierarchical possibilities of the target language: the automata can not be nested. Figure A.17 shows the generated LUSTRE file, and Figure A.18 shows the SMV equivalent. They are very similar.

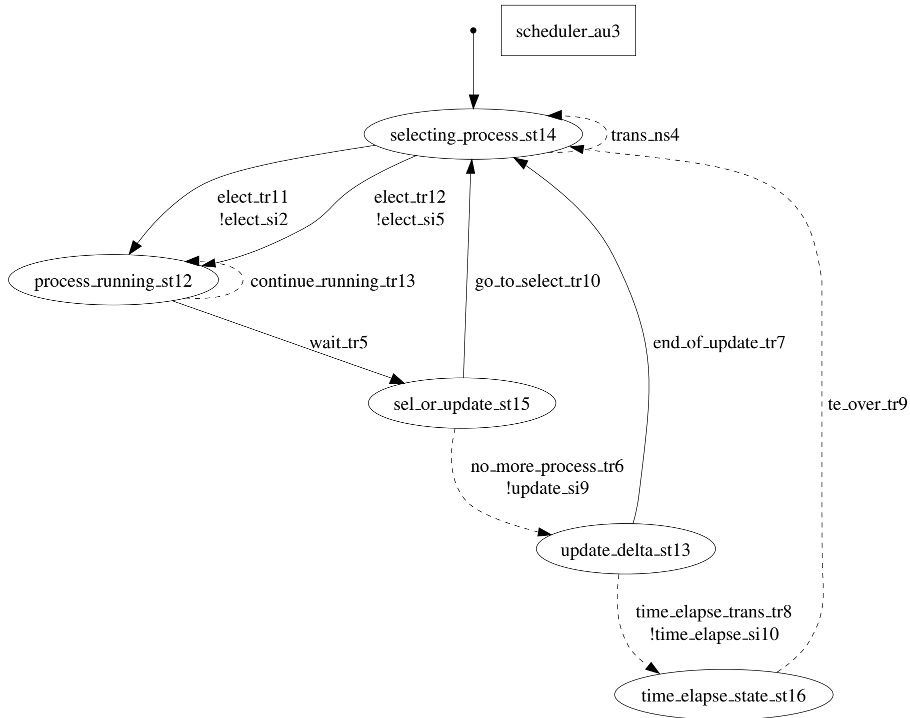


Figure A.13: Automaton for the Scheduler

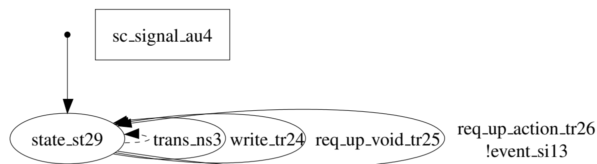


Figure A.14: Automaton for an `sc_signal`

```
1  node sc_signal_au4 (input_si12_present: bool;
2                      input_si12_value: bool;
3                      update_si9: bool;
4                      siginit_sc_signal_au4_un3: bool;
5                      make_choice_unique_nd4_un13: bool;
6                      make_choice_unique_nd4_un14: bool)
7  returns (state_st29: bool; event_si13: bool;
8           cur_si15_present: bool; cur_si15_value: bool);
9  var
10   next_val_va3_va8 : bool;
11   curr_val_va2_va7 : bool;
12   trans_ns3: bool;
13   write_tr24: bool;
14   req_up_void_tr25: bool;
15   req_up_action_tr26: bool;
16 let
17   -- Transitions:
18   trans_ns3 = false -> (pre(state_st29) and
19     (not [...]);  -- > state_st29
20   write_tr24 = false -> (pre(state_st29) and
21     (input_si12_present and
22      ((make_choice_unique_nd4_un13 = false) and
23       (make_choice_unique_nd4_un14 = false))));  -- > state_st29
24   req_up_void_tr25 = false -> (pre(state_st29) and
25     ((update_si9 and
26       ((pre curr_val_va2_va7) = (pre next_val_va3_va8))) and
27       ((make_choice_unique_nd4_un13 = true) and
28        (make_choice_unique_nd4_un14 = false))));  -- > state_st29
29   req_up_action_tr26 = false -> (pre(state_st29) and
30     ((update_si9 and
31       ((pre curr_val_va2_va7) <> (pre next_val_va3_va8))) and
32      (not (((make_choice_unique_nd4_un13 = false) and
33             (make_choice_unique_nd4_un14 = false)) or
34            ((make_choice_unique_nd4_un13 = true) and
35             (make_choice_unique_nd4_un14 = false))))));
36                                             -- > state_st29
37   -- States:
38   state_st29 = true;
39   -- Pure signals
40   event_si13 = false -> req_up_action_tr26;
41
42   -- Valued Signals
43
44   -- Continuous signals
45   cur_si15_present = true;
46   cur_si15_value = true -> (pre curr_val_va2_va7);
47
48   -- Variables
49   next_val_va3_va8 = siginit_sc_signal_au4_un3 -> (
50     if write_tr24 then (input_si12_value) else
51        pre (next_val_va3_va8));
52   curr_val_va2_va7 = siginit_sc_signal_au4_un3 -> (
53     if req_up_action_tr26 then ((pre next_val_va3_va8)) else
54        pre (curr_val_va2_va7));
55 tel.
```

Figure A.15: Generated LUSTRE code for an `sc_signal`

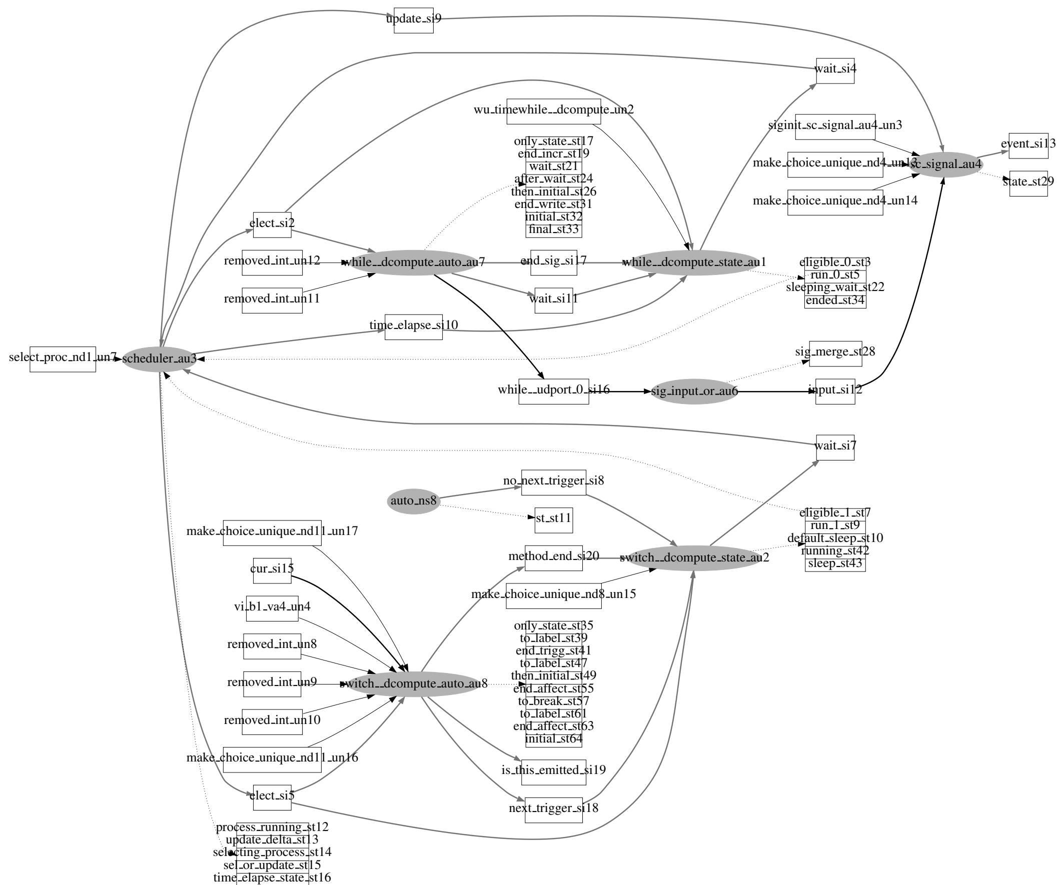


Figure A.16: Global view of the HPIOM system

```
1  include "bool_signal.lus"
2  include "sc_signal.lus"
3  include "scheduler.lus"
4  include "switch__dcompute_auto.lus"
5  include "while__dcompute_auto.lus"
6
7  node main (siginit_sc_signal_au4_un3: bool;
8             make_choice_unique_nd4_un13: bool;
9             make_choice_unique_nd4_un14: bool;
10            select_proc_nd1_un7: bool;
11            make_choice_unique_nd8_un15: bool;
12            vi_b1_va4_un4: bool;
13            removed_int_un8: bool;
14            removed_int_un9: bool;
15            removed_int_un10: bool;
16            make_choice_unique_nd11_un16: bool;
17            make_choice_unique_nd11_un17: bool;
18            wu_timewhile__dcompute_un2: bool;
19            removed_int_un11: bool;
20            removed_int_un12: bool)
21   returns (OK : bool);
22 var
23 sig_merge_st28 : bool;
24   input_si12_present: bool; input_si12_value: bool;
25 [...]
26   while__udport_0_si16_present: bool; while__udport_0_si16_value: bool;
27 let
28   OK = not (is_this_emitted_si19);
29   sig_merge_st28, input_si12_present, input_si12_value =
30     sig_input_or_au6(while__udport_0_si16_present,
31                      while__udport_0_si16_value);
32   state_st29, event_si13, cur_si15_present, cur_si15_value =
33     sc_signal_au4(input_si12_present, input_si12_value, update_si9,
34                   siginit_sc_signal_au4_un3, make_choice_unique_nd4_un13,
35                   make_choice_unique_nd4_un14);
36   process_running_st12, update_delta_st13, selecting_process_st14,
37       sel_or_update_st15, time_elapse_state_st16, time_elapse_si10,
38       elect_si5, update_si9, elect_si2 =
39     scheduler_au3(wait_si4, wait_si7, eligible_1_st7, eligible_0_st3,
40                   select_proc_nd1_un7);
41   eligible_1_st7, run_1_st9, default_sleep_st10,
42       running_st42, sleep_st43, wait_si7 =
43     switch__dcompute_state_au2(no_next_trigger_si8, next_trigger_si18,
44                                elect_si5, method_end_si20,
45                                 make_choice_unique_nd8_un15);
46   st_st11, no_next_trigger_si8 = auto_ns8(true);
47   only_state_st35, to_label_st39, end_trigg_st41, to_label_st47,
48       then_initial_st49, end_affect_st55, to_break_st57,
49       to_label_st61, end_affect_st63, initial_st64,
50       next_trigger_si18, method_end_si20, is_this_emitted_si19 =
51     switch__dcompute_auto_au8(cur_si15_present, cur_si15_value, elect_si5,
52                               vi_b1_va4_un4, removed_int_un8, removed_int_un9,
53                               removed_int_un10, make_choice_unique_nd11_un16,
54                               make_choice_unique_nd11_un17);
55   eligible_0_st3, run_0_st5, sleeping_wait_st22, ended_st34, wait_si4 =
56     while__dcompute_state_au1(end_sig_si17, wait_si11, time_elapse_si10,
57                               elect_si2, wu_timewhile__dcompute_un2);
58   only_state_st17, end_incr_st19, wait_st21, after_wait_st24, then_initial_st26,
59       end_write_st31, initial_st32, final_st33, end_sig_si17, wait_si11,
60       while__udport_0_si16_present, while__udport_0_si16_value =
61     while__dcompute_auto_au7(elect_si2, removed_int_un11, removed_int_un12);
62 tel.
```

Figure A.17: Main File For a Generated LUSTRE program

```
1   #include "bool_signal.smv"
2   #include "sc_signal.smv"
3   #include "scheduler.smv"
4   #include "switch__dcompute_auto.smv"
5   #include "while__dcompute_auto.smv"
6
7   MODULE main() {
8    input_si12_present: boolean;
9    input_si12_value: boolean;
10   event_si13: boolean;
11   cur_si15_present: boolean;
12   cur_si15_value: boolean;
13   time_elapse_si10: boolean;
14   elect_si5: boolean;
15   update_si9: boolean;
16   elect_si2: boolean;
17   wait_si7: boolean;
18   eligible_1_st7_sn : boolean;
19   no_next_trigger_si8: boolean;
20   next_trigger_si18: boolean;
21   method_end_si20: boolean;
22   is_this_emitted_si19: boolean;
23   wait_si4: boolean;
24   eligible_0_st3_sn : boolean;
25   end_sig_si17: boolean;
26   wait_si11: boolean;
27   while__udport_0_si16_present: boolean;
28   while__udport_0_si16_value: boolean;
29
30  sig_input_or_au6_inst : sig_input_or_au6
31      (while__udport_0_si16_present, while__udport_0_si16_value,
32       input_si12_present, input_si12_value);
33  sc_signal_au4_inst : sc_signal_au4
34      (input_si12_present, input_si12_value, update_si9,
35       event_si13, cur_si15_present, cur_si15_value);
36  scheduler_au3_inst : scheduler_au3
37      (wait_si4, wait_si7, eligible_1_st7_sn, eligible_0_st3_sn,
38       time_elapse_si10, elect_si5, update_si9, elect_si2);
39  switch__dcompute_state_au2_inst : switch__dcompute_state_au2
40      (no_next_trigger_si8, next_trigger_si18, elect_si5,
41       method_end_si20, wait_si7, eligible_1_st7_sn);
42  auto_ns8_inst : auto_ns8(no_next_trigger_si8);
43  switch__dcompute_auto_au8_inst : switch__dcompute_auto_au8
44      (cur_si15_present, cur_si15_value, elect_si5,
45       next_trigger_si18, method_end_si20, is_this_emitted_si19);
46  while__dcompute_state_au1_inst : while__dcompute_state_au1
47      (end_sig_si17, wait_si11, time_elapse_si10, elect_si2,
48       wait_si4, eligible_0_st3_sn);
49  while__dcompute_auto_au7_inst : while__dcompute_auto_au7
50      (elect_si2, end_sig_si17, wait_si11,
51       while__udport_0_si16_present, while__udport_0_si16_value);
52
53  assert G (  ~(is_this_emitted_si19) );
54
55  }
```

Figure A.18: Main File For a Generated SMV program

Appendix B

Why Formal Languages Should be Simple . . . and Formal

Contents

B.1 Introduction . . . **171**
B.1.1 Formal and Informal Languages . . . 171
B.1.2 Formal Proof, 100% Safety? . . . 172
B.2 The Rudimentary Approach: LUSTRE . . . **172**
B.3 A Higher Level Language: SMV . . . **172**
B.4 An Example of Problem Using Higher Level Language . . . **173**
B.4.1 The Problem . . . 173
B.4.2 Trying to Understand . . . 173
B.4.3 Consequence on LUSSY . . . 174
B.5 Executability and Determinism: a Way to Clarify the Semantics of Informal Languages . . . **175**
B.6 Conclusion . . . **175**

B.1 Introduction

This section discusses the expected qualities for a formal language. We take the examples of LUSTRE and SMV, and compare the drawbacks and advantages of their respective approach. Based on this, we also take conclusions about HPIOM: although it has no concrete syntax, it is a formalism very close to LUSTRE and SMV, and some of the design choices for HPIOM were guided by our observations on those languages.

B.1.1 Formal and Informal Languages

A simple-minded approach to the question of formal language may distinguish clearly between two categories of programming languages: *formal languages*, and the others, that we will call *informal languages*. A formal language can be defined as a language for which a clean semantic theory exists. It can be seen as the concrete form of a mathematical theory. For example, the simplest flavors of the lisp language are mainly a concrete syntax for the lambda calculus.

A formal language can, by definition, be manipulated mathematically. Formal proofs can be performed on it (undecidability can be a limitation, but there are always some cases where a tool can conclude “I’m 100% sure that the program meets its specifications”).

Unfortunately, the distinction between formal and informal languages is not so clear. A well-enough specified general purpose programming language can be very close to a formal language, and a language supposed to be formal can actually be informal if there is a hole in its specification.

B.1.2 Formal Proof, 100% Safety?

Formal methods are up to now rarely applied to general purpose programs, but proved to be very efficient and appreciated in domains where bugs are very costly, either in terms of financial costs or in terms of physical danger (a video game is not validated in the same way as a plane controller). In this case, the goal is not to eliminate as many bugs as possible, but to eliminate *all* bugs.

If we trust the hardware, the operating system, the proof engine (both the theory and the implementation), the compiler, etc . . . then we can trust the result of the proof, in the sense that this result is the correct answer to a question that has been asked to the prover.

If formal proof was so reliable, then why would famous computer scientists like Donald E. Knuth write sentences like “Beware of bugs in the above code; I have only proved it correct, not tried it.”?

The open problem is whether the *question* that was asked the prover was correct. There are a few good properties that we want any program to verify (no arithmetic overflow, no division by zero, . . .). They are necessary but not sufficient conditions for the program’s correctness. For a complete proof of correctness, we need to give the specifications of the program to the prover. The formal proof will then consist in comparing two views of the program, either written in the same language (for example, LUSTRE), or written in two different languages (for example, SMV and CTL). For the proof to be meaningful, we have to be confident in one of the view representing the expected behavior, and the other representing the actual behavior. This part of the validation of the software can not be formal, because neither the original requirements nor the way the program will interact with its physical environment are originally formal. In other words, formal methods apply to model, and the soundness of the model regarding the physical world has to be verified informally.

The solution to make this informal verification reliable is to make it trivial enough. It would not be reasonable, for example, to write 10 pages of equation containing only Greek letters and obscure symbols, and to say “trust me, this is a model of your car. Now, I will prove that the break-by-wire is correct based upon that”. This can be a good LaTeX benchmark, not a good proof.

B.2 The Rudimentary Approach: LUSTRE

The approach followed by the LUSTRE language is to make the language as simple as possible. The language is based on a flow algebra. Most operators are trivial extensions of standard arithmetic operators. `pre` and `->` are added to be able to describe sequential machines, but their definition is simple and unambiguous.

The result is a language very simple to learn and to understand. It has a graphical version SCADE™, which is also easy to use. The properties are expressed using the same language as the implementation, so, the user has only one language to learn.

The drawback of this approach is that some constructs are very complex to describe. For example, adding a “reset” input signal to a LUSTRE program means a complete transformation of the program (the reset signal has to be added to each node, and a condition on this signal has to be added to each equation). Describing a state machine in LUSTRE can also lead to complex and error-prone code.

B.3 A Higher Level Language: SMV

SMV follows the opposite approach. It is a very rich, but much more complex language. It mixes data-flow and imperative style, synchronous and asynchronous semantics, . . . Most of the SMV constructs could be expressed in a simpler language like LUSTRE, but the syntax of SMV allows to be more concise.

If we keep our naive Manichean view of the world, can we consider SMV as a formal language? The SMV language reference is a document written in natural English, so SMV can not be considered formal. However, this is the input language of a formal verifier, and the SMV tool needs a precise and exhaustive semantics of its input language. We may consider that SMV is formally and exhaustively specified, but that the specification hasn’t been published.

B.4 An Example of Problem Using Higher Level Language

B.4.1 The Problem

An example of a construct for which the natural language specification of SMV is not sufficient is the case when a signal is not assigned. The manual says *"each signal may be assigned only once."*, which explicitly forbids cases where the number of assignments is greater or equal to 2, but isn't clear about what happens if the signal is not assigned at all. Another potentially relevant paragraph in the manual says *"When a signal is assigned a value that is not an element of its declared type, the result is to assign the special value *unknown* to that signal. Unknown values can occur in other circumstances (for example, in the case of a conditional with no "else" clause). The result of assigning the unknown value to a signal is implementation dependent. In some cases, it may be desirable to propagate unknown values. In other cases, the unknown value may be replaced by a nondeterministic choice among the values in the signal's declared type (see next section)."*. We may be in the case of the "other circumstances".

The problem doesn't appear in LUSTRE, since the rule is that each variable has to appear once and only once on the left hand side of an equation.

The possible semantics in this case are:

1. Reject the program, like LUSTRE does.
2. Use a non-deterministic value for the variable if the variable is not assigned. This is a way to make `default X :=` {*all, possible, values, for, X*} `in {...}` implicit.
3. Keep the previous value for the variable. This would be strange for stateless variables, but makes sense for variables whose `next` value can be assigned: it means that a variable doesn't change its value unless explicitly requested.
4. Accept the program, and say it's behavior is an "undefined behavior" (i.e. the complete program may behave in any way from this point). This is acceptable for a general purpose programming language, but hardly so for a formal language.

B.4.2 Trying to Understand

Since we didn't find a satisfying answer in the manual, we can try some experimentation to get an empirical answer. The first option (reject the program) is immediately eliminated: the program of Figure B.1 is accepted by SMV.

```
1   module main ()
2   {
3   v: boolean;
4   i: boolean;
5   init(i) := 1;
6   next(i) := 0;
7
8   if (i) {
9     v := 1;
10  }
11
12  prop2: assert (G v);
13  }
14
```

Figure B.1: Example of an SMV Program With Non-Assigned Variable

Moreover, SMV gives a counterexample for the property, saying that `v` can become false at the second step. In this case, the second rule (non-deterministic value) was applied. But let us not stop here. Consider the program of Figure B.2.

```
1   module main ()
2   {
3   v: boolean;
4   init(v) := 1;
5
6   default {
7
8   } in {
9     if (0) {
10      next(v) := 0;
11    }
12  }
13
14  prop2: assert (G v);
15  }
16
```

Figure B.2: Example of an SMV Program With Non-Assigned Stateful Variable

It is accepted and proved correct by SMV. So, in this case, the third rule (keep the previous value) seems to have been applied. At this point, we could conclude "if a variable isn't assigned, then a non-deterministic value is used for stateless variables, and the previous value is kept for stateful variables". But once again, let's not stop here. The programs of Figures B.3 and B.4 are accepted by SMV, and there properties are falsified.

```
1   module main ()
2   {
3   v: boolean;
4   init(v) := 1;
5
6   default {
7
8   } in {
9
10  }
11  prop2: assert (G v);
12  }
```

Figure B.3: Never-Assigned Stateful Variables

The behavior therefore depends on the presence of at least one assignment, and on the presence of a `default` statement. Whether this is a bug or a feature is left to the appreciation of the reader. Anyway, the conclusion of this experiment is that we can not rely on the behavior of SMV if a variable is not always assigned. A reasonable rule is to provide a default value for all variables in the `default` clause.

B.4.3 Consequence on LUSSY

This strange behavior was the cause of a bug in the SMV back-end of LUSSY: HPIOM variables were translated into SMV stateful variables, but we didn't provide a default value for them, and relied on the fact that they would keep their value by default. In 99% of the cases, we had at least one assignment on the `next` value of the variable, and the `default` statement was generated unconditionally, so, the assumption

```
1   module main ()
2   {
3   v: boolean;
4   init(v) := 1;
5
6   if (0) {
7     next(v) := 0;
8   }
9   prop2: assert (G v);
10  }
11
```

Figure B.4: With Non-Assigned Stateful Variables, Without `default` Statement

was verified. As a result, we discovered the bug very late, and spent a hard time identifying it among 3,000 lines of generated SMV code.

The particularity of this bug is that LUSSY was actually doing what we *thought* was the correct behavior. The problem was not in the algorithm or the implementation of LUSSY, but in our understanding of the semantics of SMV. We could easily have missed this bug, which is still a real bug from the user point of view: given a SystemC program, the diagnosis of LUSSY was incorrect.

B.5 Executability and Determinism: a Way to Clarify the Semantics of Informal Languages

The trial-and-error scenario described above is representative the way many people like to learn and understand a language. When I'm asked whether a construct is valid in C++, and what it is supposed to do, I first try it with one or several compilers, and then try to understand the result and whether the compiler is right, from the specifications.

A problem with the SMV language is that no interpreter or compiler is available. We can prove properties on a program, but not execute it. To get confidence in his understanding of the specifications, the user can only try to prove properties. When the property is false, he can look at the counterexample, and when it's true, he has no way to know if his explanation of the correctness of the property is the right one.

Executability is therefore a key point for a language in the perspective of formal methods. Determinism is also very appreciable, since it means that the question "What does the program do in the situation *X*" have only one answer for a given *X*, and this answer can be obtained by execution of the program.

B.6 Conclusion

We discussed the importance of a simple and exhaustive definition for a formal language, and the importance of executability. Those remarks also apply for an abstract formalism, such as HPIOM. In HPIOM, the basic constructs are very simple, and others are defined based on the basic ones.

Executability of HPIOM is crucial, not only to help understanding its semantics, but also to allow the comparison of its execution with SystemC. This is not a formal proof of equivalence, which is anyway not possible, but increases our confidence in the correctness of the translation.

Bibliography

[Act] Actis Design, LLC. AccurateC™ Rule Checker. http://www.actisdesign.com/. 4.2.1.5

[AN96] JR Abrial and A Nikolaos. *The B-book: assigning programs to meanings*. Cambridge University Press New York, NY, USA, 1996. ISBN:0-521-49619-5. 3.2.1.1

[ARMip] ARM Limited. AHB Example Amba SYstem technical reference manual, http://www.arm.com/pdfs/DDI0170A.zip. 2.5.1

[Arn99] Guido Arnout. C for system level design, September 1999. https://www.systemc.org/projects/sitedocs/document/coWare/en/1. 2.4.1

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers — Principles, Techniques, and Tools*. Addison-Wesley, Reading, MA, USA, 1986. 4.3, 7.3.3.1

[Bak95] Wendell Craig Baker. *Interfacing System Description Languages to Formal Verification*. PhD thesis, University of California, Berkeley, 1995. Available as UCB/ERL M96/26. 3.2.2

[BBDEL96] Ilan Beer, Shoham Ben-David, Cindy Eisner, and Avner Landver. RuleBase: an industry-oriented formal verification tool. In *33rd Design Automation Conference (DAC'96)*, pages 655–660, New York, June 1996. Association for Computing Machinery. 3.2.1.2

[BC04] Yves Bertot and Pierre Casteran. *Interactive Theorem Proving and Program Development*. SpringerVerlag, 2004. 3.2.1.1

[BCC+99] A. Biere, A. Cimatti, E. M. Clarke, M. Fujita, and Y. Zhu. Symbolic model checking using SAT procedures instead of BDDs, September 10 1999. 3.2.1.4

[BCDM03] Fabrice Baray, Philippe Codognet, Daniel Diaz, and Henri Michel. Code-based test generation for validation of functional processor descriptions. In *TACAS*, pages 569–584, April 2003. 3.2.2

[BCH+85] J-L. Bergerand, P. Caspi, N. Halbwachs, D. Pilaud, and E. Pilaud. Outline of a real time data-flow language. In *Real Time Systems Symposium*, pages 33–42, San Diego, September 1985. 1.4, 2.4.2.3, 3.3.3

[Ber00] G. Berry. The foundations of Esterel. *Proof, Language and Interaction: Essays in Honour of Robin Milner*, pages 425–454, 2000. Editors: G. Plotkin, C. Stirling and M. Tofte. 2.4.2.3, 5.1.2.1, 5.2.2.3.4

[BFG99] Marius Bozga, Jean-Claude Fernandez, and Lucian Ghirvu. State space reduction based on live variables analysis. In Agostino Cortesi and Gilberto Filé, editors, *Static Analysis*, volume 1694 of *Lecture Notes in Computer Science*, pages 164–178. Springer, 1999. 7.3.3.1

[BGJ89] A. Benveniste, P. le Guernic, and C. Jacquemot. Synchronous programming with events and relations: the SIGNAL language and its semantics. Technical Report 459, IRISA, Rennes, France, 1989. 6.6.3

[BGO+04] Marius Bozga, Susanne Graf, Iulian Ober, Ileana Ober, and Joseph Sifakis. The if toolset. In *4th International School on Formal Methods for the Design of Computer, Communication and Software Systems: Real Time, SFM-04:RT, Bologna, Sept. 2004*, LNCS Tutorials, Springer, 2004. 5.1.2.1

[BLP+02] Felice Balarin, Luciano Lavagno, Claudio Passerone, Alberto L. Sangiovanni-Vincentelli, Marco Sgroi, and Yosinori Watanabe. Modeling and designing heterogeneous systems. In *Concurrency and Hardware Design, Advances in Petri Nets*, pages 228–273. Springer-Verlag, 2002. 2.4.1

[BR00] Thomas Ball and Sriram K. Rajamani. Boolean programs: A model and process for software analysis. Technical report, Microsoft Research, February 2000. 3.2.2

[Bry92] Randal E. Bryant. Symbolic Boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys*, 24(3):293–318, 1992. 3.2.1.2

[Bul00] Tevfik Bultan. Action language: A specification language for model checking reactive systems. pages 335–344, March 2000. 3.3

[Cad] Cadence Design Systems. NC-SystemC. `http://www.cadence.com/products/functional_ver/nc-systemc/`. 4.2.1.6

[Cad99] Cadence. *Formal Verification Using Affirma FormalCheck*, October 1999. 3.2.1.2

[CC77] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL'77*, Los Angeles, January 1977. 3.2.1.3

[CG03] Lukai Cai and Daniel Gajski. Transaction level modeling: an overview. In *CODES+ISSS '03: Proceedings of the 1st IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pages 19–24, New York, NY, USA, 2003. ACM Press. 2.2.2

[Cha82] G. J. Chaitin. Register allocation and spilling via graph coloring. *ACM SIGPLAN Notices*, 17(6):98–105, June 1982. 7.3.3.1.2

[CK03] Edmund Clarke and Daniel Kroening. Hardware verification using ANSI-C programs as a reference. In *Proceedings of ASP-DAC 2003*, pages 308–311. IEEE Computer Society Press, January 2003. 3.2.2

[CKL04] Edmund Clarke, Daniel Kroening, and Flavio Lerda. A tool for checking ANSI-C programs. In Kurt Jensen and Andreas Podelski, editors, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2004)*, volume 2988 of *Lecture Notes in Computer Science*, pages 168–176. Springer, 2004. 3.2.2

[DCB+90] D. L. Dill, E. M. Clarke, J. R. Burch, K. L. Mcmillan, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond, march 1990. 3.2.1.2

[Deu03] Dr. Alain Deutsch. Static verification of dynamic properties. Technical report, PolySpace Technologies, November 2003. 6.1.1

[DG03] Rolf Drechsler and Daniel Große. Formal verification of ltl formulas for systemc designs, 2003. `http://www.informatik.uni-bremen.de/grp/ag-ram/doc/konf/iscas03_verification_systemc.pdf`. 3.2.2

[Don] Steve Donovan. The UnderC development project. `http://home.mweb.co.za/sd/sdonovan/underc.html`. 4.3.3.1

[Edi] Edison Design Group. Compiler front ends. http://www.edg.com/. 4.2.1.2

[EGK+03] John Ellson, Emden Gansner, Lefteris Koutsofios, Stephen C. North, and Gordon Woodhull. Graphviz - open source graph drawing tools. In *Lecture Notes in Computer Science*, volume Volume 2265 / 2002, page 483. Springer-Verlag GmbH, July 2003. 3.3.3.5

[FGC+04] Görschwin Fey, Daniel Große, Tim Cassens, Christian Genz, Tim Warode, and Rolf Drechsler. ParSyC: An efficient SystemC parser. In *Synthesis And System Integration of Mixed Information technologies*, 2004. http://www.informatik.uni-bremen.de/agram/doc/work/04sasimi_parsyc.pdf. 4.2.1.4

[FN01] Masahiro Fujita and Hiroshi Nakamura. The standard SpecC language. In *ISSS*, pages 81–86, 2001. 2.4.1

[Fre] Free Software Fundation. Free software definition. http://www.gnu.org/philosophy/free-sw.html. 4.4.5.2

[FRS02] Fabrizio Ferrandi, Michele Rendine, and Donatella Sciuto. Functional verification for systemC descriptions using constraint solving. In *DATE*, pages 744–751, 2002. 3.2.2

[Ghe05] Frank Ghenassia, editor. *Transaction-Level Modeling with SystemC. TLM Concepts and Applications for Embedded Systems*. Springer, June 2005. ISBN 0-387-26232-6. 2.3.2, 9.2.1

[GHJV94] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994. 5.5

[GNJ+96] Gopi Ganapathy, Ram Narayan, Glenn Jorden, Denzil Fernandez, Ming Wang, and Jim Nishimura. Hardware emulation for functional verification of K5. pages 315–318, 1996. 2.2.1

[God97] Patrice Godefroid. Model checking for programming languages using VeriSoft. In ACM, editor, *Conference record of POPL '97, the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages: papers presented at the symposium, Paris, France, 15–17 January 1997*, pages 174–186, New York, NY, USA, 1997. ACM Press. 3.2.2

[Goe01] Gregor Goessler. Prometheus - A compositional modeling tool for real-time systems, September 25 2001. http://www.inrialpes.fr/pop-art/share/Publications/goessler/prometheus.ps. 6.6.2

[Hav99] Klaus Havelund. Java pathfinder: A translator from java to promela. page 152, September 30 1999. 3.2.2

[HB02] N. Halbwachs and S. Baghdadi. Synchronous modeling of asynchronous systems. In *EMSOFT'02*, Grenoble, October 2002. LNCS 2491, Springer Verlag. 8.2.2.2.2

[HGR+01] Dirk Hoffmann, Joachim Gerlach, Juergen Ruf, Thomas Kropf, Wolfgang Mueller, and Wolfgang Rosenstiehl. The simulation semantics of systemC, December 21 2001. 6.6.1

[HHL02] D. Heuzeroth, T. Holl, and W. Löwe. Combining static and dynamic analyses to detect interaction patterns, 2002. In IDPT, 2002. (submitted to). Welf Löwe and Markus Noga. 4.2.2.1

[HLR92] N. Halbwachs, F. Lagnier, and C. Ratel. Programming and verifying critical systems by means of the synchronous data-flow programming language LUSTRE. *IEEE Transactions on Software Engineering, Special Issue on the Specification and Analysis of Real-Time Systems*, pages 785–793, September 1992. 3.2.1.2, 3.3.3

[HLR93] N. Halbwachs, F. Lagnier, and P. Raymond. Synchronous observers and the verification of reactive systems. In M. Nivat, C. Rattray, T. Rus, and G. Scollo, editors, *Third Int. Conf. on Algebraic Methodology and Software Technology, AMAST'93*, pages 83–96, Twente, June 1993. Workshops in Computing, Springer Verlag. 6.5.4

[HPV00] Klaus Havelund, Seungjoon Park, and Willem Visser. Java pathfinder - second generation of a java model checker. May 27 2000. 3.2.2

[HSH+] Pei-Hsin Ho, Thomas Shiple, Kevin Harer, James Kukula, Robert Damiano, Valeria Bertacco, Jerry Taylor, and Jiang Long. Smart simulation using collaborative formal and simulation engines. pages 120–126. 3.2.1.2

[Jea00] B. Jeannet. *Partitionnement Dynamique dans l'Analyse de Relations Linéraires et Application à la Vérification de Programmes Synchrones*. PhD thesis, Institut National Polytechnique de Grenoble, September 2000. 3.3.3

[Jea03] B. Jeannet. Dynamic partitioning in linear relation analysis. application to the verification of reactive systems. *Formal Methods in System Design*, 23(1):5–37, July 2003. 3.2.1.3, 3.3.3

[JFL97] Dunoyer Julien, Pétrot Frédéric, and Jacomme Ludovic. Intrinsic limitations of logarithmic encodings for low power finite state machines. In *Mixed Design of VLSI Circuits Conference*, pages 613–618, June 1997. 8.3.1

[JJGM03] E. Jahier, B. Jeannet, F. Gaucher, and F. Maraninchi. Automatic state reaching for debugging reactive programs. In *AADEBUG*, Ghent, September 2003. 7.5.3

[KPR+97] Alain Kerbrat, Amir Pnueli, Anne Rasse, Eugene Asarin, Marius Bozga, and Oded Maler. Data-structures for the verification of timed automata, April 30 1997. 8.2.2.2.2

[KS05] Daniel Kroening and Natasha Sharygina. Formal verification of systemc by automatic hardware/software partitioning. In *Proceedings of MEMOCODE 2005*, pages 101–110. IEEE, 2005. 3.2.2, 6.6.3

[Lam77] L. Lamport. Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering*, SE-3(2):125–143, 1977. 3.2.1.2

[MBPS] Deepak Matthaikutty, David Berner, Hiren Patel, and Sandeep Shukla. SystemCXML. `http://systemcxml.sourceforge.net/`. 4.2.1.2

[McM93] K. L. McMillan. *Symbolic Model Checking*. PhD thesis, Boston, 1993. 1.4, 3.3.3

[McM99] K. L. McMillan. The SMV language. Technical report, Cadence Berkeley Labs, March 1999. 8.2.2.2.2

[McM01] Kenneth L. McMillan. *Getting started with SMV*. Cadence Berkeley Labs, Berkeley, CA 94704 USA, 2001. 1.4, 3.3.3, 8.2.2.2.2

[Mey92] Bertrand Meyer. Applying "design by contract". *IEEE Computer*, 25(10):40–51, 1992. 10.2

[MG00] Florence Maraninchi and Fabien Gaucher. Step-wise + algorithmic debugging for reactive programs: Ludic, a debugger for lustre. In *Automated and Algorithmic Debugging*, 2000. 8.5.1

[MMMC05a] Matthieu Moy, Florence Maraninchi, and Laurent Maillet-Contoz. LusSy: A toolbox for the analysis of systems-on-a-chip at the transactional level. In *International Conference on Application of Concurrency to System Design*, pages 26–35, June 2005. 3.3.3.2, 9.2.1

[MMMC05b] Matthieu Moy, Florence Maraninchi, and Laurent Maillet-Contoz. Pinapa: An extraction tool for systemc descriptions of systems-on-a-chip. In *EMSOFT*, pages 317 – 324, September 2005. 3.3.3.1, 9.2.1

[MR01] F. Maraninchi and Y. Rémond. Argos: an automaton-based synchronous language. *Computer Languages*, (27):61–92, 2001. 5.1.2.1

[Mue93] Frank Mueller. Register allocation by graph coloring: A review, April 14 1993. 7.3.3.1.2

[NPW02] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002. 3.2.1.1

[Ope] Open Source Initiative. Open source definition. `http://www.opensource.org/docs/definition.php`. 4.4.5.2

[Ope03] Open SystemC Initiative. *SystemC v2.0.1 Language Reference Manual*, 2003. `http://www.systemc.org/`. 2.4.2

[ORS92] S. Owre, J. M. Rushby, , and N. Shankar. PVS: A prototype verification system. In Deepak Kapur, editor, *11th International Conference on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752, Saratoga, NY, jun 1992. Springer-Verlag. 3.2.1.1

[Pal04] Nicolas Palix. Modélisation et vérification de systèmes sur puces à base de composants. Master's thesis, ESISAR/INPG, Grenoble, France, 2004. 4.3.4

[Pas02] Sudeep Pasricha. Transaction level modeling of SoC in systemc 2.0. Technical report, STMicroelectronics Ltd, 2002. 2.2.2

[Phi03] Ammar Aljer Philippe. BHDL: Circuit design in b, 2003. 3.2.1.1

[Pro04] The GCC Project. Ssa for trees, 2004. `http://gcc.gnu.org/projects/tree-ssa/`. 6.6.3

[Pér05] Mathias Péron. IS, un domaine numérique abstrait pour l'analyse de programmes manipulant des adresses. Master's thesis, Université Joseph Fourier, VERIMAG, Grenoble, France, Jun 2005. 5.5.0.5

[RHKR01] J. Ruf, D. Hoffmann, T. Kropf, and W. Rosenstiel. Simulation-guided property checking based on multivalued ar-automata. In *Design, Automation and Test in Europe*, pages 742–748, 2001. 3.2.2

[Rou05] Yvan Roux. Rapport intermédiaire du projet vercors. Internal report in INRIA Rhône Alpes, June 2005. 4.3.4

[RR02] Claudio Riva and Jordi Vidal Rodriguez. Combining static and dynamic views for architecture reconstruction. In *European Conference on Software Maintenance and Reengineering*, pages 47–56. IEEE Computer Society Press, 2002. 4.2.2.1

[RSP+05] Adam Rose, Stuart Swan, John Pierce, Jean-Michel Fernandez, and Cadence Design Systems. Transaction level modeling in systemc. Technical report, Mentor Graphics; Cadence Design Systems, 2005. available in the `docs/` directory of the OSCI TLM library. 2.4.3.2

[Sal03] Ashraf Salem. Formal semantics of synchronous systemc. In *DATE*, pages 10376–10381, 2003. 6.6.1

[SCCS05] Natasha Sharygina, Sagar Chaki, Edmund M. Clarke, and Nishant Sinha. Dynamic component substitutability analysis. In *FM*, pages 512–528, 2005. 10.2

[SF02] T. Sakunkonchak and M. Fujita. Verification of synchronization in SpecC description with the use of difference decision diagrams. In *FDL*, 2002. 3.2.2

[SH97] S. Graf and H. Saidi. Construction of abstract state graphs with PVS. In O. Grumberg, editor, *Proc. 9th INternational Conference on Computer Aided Verification (CAV'97)*, volume 1254, pages 72–83. Springer Verlag, 1997. 3.2.2

[Sha05] Muhammad Muzammil Shahbaz. Automatic verification techniques for system-on-chip using symbolic model verifier. Master's thesis, Université Joseph Fourier, Grenoble, France, June 2005. Work performed in Verimag. 8.1

[Sny] Wilson Snyder. SystemPerl home page. `http://www.veripool.com/systemperl.html`. 4.2.1.1

[SPI] SPIRIT Consortium. `http://www.spiritconsortium.com/`. 4.3.3.2

[St] Richard M. Stallman and the GCC developer community. *GNU Compiler Collection Internals*. Free Software Fundation. `http://gcc.gnu.org/onlinedocs/gccint/`. 4.3.1.3.3

[Syn] Synopsys Inc. CoCentric System Studio. `http://www.synopsys.com/products/cocentric_studio/cocentric_studio.html`. 4.2.1.2

[syn03] 2003. Discussion with the team developing the front-end of the CoCentric suite before writting Pinapa. 4.2.1.2

[TBS+04] Jean-Pierre Talpin, David Berner, Sandeep Kumar Shukla, Paul Le Guernic, Abdoulaye Gamatié, and Rajesh Gupta. A behavioral type inference system for compositional system-on-chip design. In *ACSD*, 2004. 6.6.3

[Tec] PROVER Technology. Prover plug-in™ for scade™. `http://www.prover.com/products/ppi/sl.xml`. 3.2.1.4

[TGSG05] J.-P. Talpin, P. Le Guernic, S. Shukla, and R. Gupta. Compositional behavioral modeling of embedded systems and conformance checking. *International Journal on Parallel processing, special issue on testing of embedded systems*, 2005. 6.6.3

[Var01] Moshe Y. Vardi. Branching vs. linear time: Final showdown. In *TACAS 2001: Proceedings of the 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 1–22, London, UK, 2001. Springer-Verlag. 5.4

[vH] Dimitri van Heesch. Doxygen. `http://www.doxygen.org`. 4.2.1.2, 5.5.1.2

[Vil] Javier Castillo Villar. sc2v: SystemC to verilog synthesizable subset translator. `http://www.opencores.org/projects.cgi/web/sc2v/overview`. 4.2.1.4

[WS04] Jim-Moi WONG and Jean-Philippe STRASSEN. EASY platform. STMicroelectronics confidential, May 2004. 2.5.2

Index

δ-cycle . . . *see* delta-cycle
SC_METHOD . . . **25**
SC_THREAD . . . **25**

ABI . . . *see* Application Binary Interface
abstract interpretation . . . 40
abstract address . . . 77, 113
    comparison . . . 81
    literal . . . 79
Abstract Syntax Tree . . . 52, 55
abstracted type . . . **77**
address . . . 32, 57, 77, 113
address map . . . 78
algorithm (level of abstraction) . . . 19
answer . . . 40
Application Binary Interface . . . 54
Application Specific Integrated Circuits . . . **18**
approximation
    conservative . . . 38, 78, 93
    non conservative . . . 118
array . . . 77
ASIC . . . *see* Application Specific Integrated Circuits
assertion . . . 105
AST . . . *see* Abstract Syntax Tree
AST instance . . . **54**
asynchronous . . . 20, 22, 93
Atelier B . . . 38
ATLAS . . . 11
automaton . . . 73

B Method . . . *see* Atelier B
back-end . . . 45, 121
BASIC . . . 31
    model . . . 99
BDD . . . *see* Binary Decision Diagrams
benchmark . . . 138
BHDL . . . 38
bibliography . . . 16
Binary Decision Diagrams . . . **39**, 112
binding . . . 25
BIRTH . . . 111
Birth . . . 45
BISE . . . 91
Bise . . . 45
bug searching . . . 40, 119

byte . . . **23**
byte-enable . . . **23**

C++ front-end . . . 92
C++ interpreter . . . 59
CAD . . . *see* Computer Aided Design
callback . . . 65
CBMC . . . 41
channel
    TAC . . . 32
    user-defined . . . 30
CIFRE . . . 13
clock tick . . . **123**
combinational . . . **123**
    automata . . . 73
communication channels . . . **20**
compiled parallelism . . . **123**
composite tree . . . 82
Computer Aided Design . . . 24, 67
conflicts graph . . . **115**
conservative approximation . . . *see* approximation, conservative
continuous signal . . . 77
contract . . . 152, **152**
contributions . . . 14
    LusSy . . . 14
    Pinapa . . . 50
control point . . . 75
control point . . . **73**
coq . . . 38
cosimulation . . . 18
cycle-accurate . . . **21**

data-member . . . 61
dead-lock . . . 42
decoration . . . 55
def . . . 114
defined variables . . . 114
delta-cycle . . . 99, 105, 119
design flow . . . 18
design gap . . . 12, **17**
design-pattern . . . 82
determinism . . . 123
Direct Memory Access . . . 23
dlopen . . . 61

DMA . . . . . . *see* Direct Memory Access
dot . . . . . . *see* graphviz
dotty . . . . . . *see* graphviz

early execution 152
EASY 33, 67
  TLM 34
EC **124**
edge **73**, 75
ELAB 52
elaboration phase **25**
eligible 95
emacs 135
embedded software 18
encapsulation
  automaton 76
Esterel 30
evaluation phase **28**
executable 92
extraction
  semantic 14
  syntactic 14

formal languages **171**
formal semantics 69, 92
front-end
  C++ 51
  SystemC 43, 51
Future operator **129**

garbage-collector 65
gate-level **18**, **21**
GBAC 137, 141
GCC . . . . . . *see* GNU Compiler Collection
  modifications 65
global state **75**
global transition **75**
Globally operator **129**
GNU Compiler Collection 54
GNU General Public License 66
GPL . . . . . . *see* GNU General Public License
graphviz 45
GreenSoCs 32

hardware 12
hardware/software partitioning **18**
HDL **24**
hierarchy
  automaton 76
HPIOM 45, 69, 93, 146
  communication 73
  implementation 82
  semantics 74
  syntax 71
HPIOM property **82**
HPIOM system **75**

IF 70
implementation
  HPIOM 82
  pinapa 60
in 114
informal languages **171**
initial state 119
initial value 119
initiator module **23**
initiator port **23**
INRIA 107
Intellectual Properties **18**
interconnect **20**
interface
  TLM 31
  user-defined 30
intermediate formalism 69
interrupt 35
IP . . . . . . *see* Intellectual Properties
Isabelle 38

LESAR 137, 141
lesar 40
limitations
  bise 93
  LusSy 15
lint 51
live variable 114, **114**
live-in variables 114
live-out variables 114
liveness property **38**
local specification 152
LusSy 42, 43
  limitations 70
Lustre 30
lustre 14, 123
lvalue 72

mask 18
Matlab 19
model **20**
model-checking **38**, 126
  bounded 40
  enumerative 39
  Java 42
  software 41
  symbolic 39, 112
modules **20**
Moore's Law 11, 17

NBAC 137

nbac . . . 141
Networks-on-a-Chip . . . **23**
Next operator . . . **129**
NoC . . . *see* Networks-on-a-Chip
node
  lustre . . . 124
non deterministic choice . . . 77
non deterministic choices . . . 113
notify . . . 98
numerical values . . . 118

one-hot encoding . . . **130**
Open SystemC Consortium Initiative . . . **24**
optimization . . . 111
OSCI . *see* Open SystemC Consortium Initiative, 31

parallelization . . . 117
parser . . . *see* front-end
Pinapa . . . 43, 49, 50
platform . . . **20**
pointers . . . 59
polychronous . . . 107
port
  TLM . . . 107
ports . . . **23**
process
  handler . . . 54
  SC_METHOD . . . 25
  SC_THREAD . . . 25
prococol
  TAC . . . 31
Programmer View . . . **20**
Programmer View plus Timing . . . **20**
property . . . 42, 105
  HPIOM . . . 81
protocol . . . 31
  BASIC . . . 31
Prover . . . 141
prover plugin . . . 41
PV . . . *see* Programmer View
PVS . . . 38
PVT . . . *see* Programmer View plus Timing

reactive systems . . . 122, **122**
reference implementation . . . **22**
Register Transfer Level . . . **18**, 21
RTL . . . *see* Register Transfer Level
run . . . 97
run-time verification . . . 152

safety property . . . **38**
SAT . . . **40**
SC2PROM . . . 107
SC_METHOD . . . 95
SC_THREAD . . . 95
SCADE . . . 138, 141
scheduler . . . 28, 95
semantic preserving . . . 113
sequential . . . **123**
signal . . . 98, 107
simulation . . . **21**
  RTL . . . 18
simulation phase . . . **25**
SLAM . . . 41
sleep . . . 97
SMV . . . 39, 138, 141
smv . . . 14
SoC . . . *see* System-on-a-Chip
software . . . 12
SPG . . . 13
SSA . . . *see* Static Single Assignment
standard
  SystemC . . . 24
  TLM . . . 31
state . . . 74, **74**, 75
state-explosion . . . 15
state-space . . . 38
Static Single Assignment . . . 107
STMicroelectronics . . . 13
symbolic model-checking . . . *see* model-checking, symbolic
synchronization . . . 20, 43, 95
synchronous approach . . . 122, **122**
synchronous languages . . . 30
System-on-a-Chip . . . **12**, **18**
SystemC . . . 25
  library . . . 52
  modifications . . . 66
  semantics . . . 92
SystemC kernel . . . **25**
systems
  reactive . . . 122

TAC . . . 31
  model . . . 101
target module . . . **23**
target port . . . **23**
test . . . **21**
test-bench
  RTL . . . 22
theorem proving . . . 38
time-to-market . . . 18
timing . . . 20, 152
TLM . . . *see* Transaction Level Modeling
tool chain . . . 43
transaction . . . **20**, 32
Transaction Level Model . . . 92, 99
Transaction Level Modeling . . . **12**, **17**, 22, 145

transition 74, 75
transport 31

unknown values 117
Until operator **129**
update phase **28**, 97
use 114
used variables 114

validation **21**
verification **21**
    formal 41
Verilog **21**
Verimag 13
Verisoft 41
VHDL **21**
visitor 83
    hierarchy 86
visualization 45

word **23**

Abstract

The work presented in this document deals with the formal verification models of Systems-on-a-Chip at the transaction level (TLM). We present the transaction level and its variants, and remind how this new level of abstraction is today necessary in addition to the register transfer level (RTL) to accommodate the growing constraints of productivity and quality, and how it integrates in the design flow.

We present a new tool, called LUSSY, that allows property-checking on transactional models written in SystemC. Its structure is similar to the one of a compiler: A front-end, PINAPA, that reads the source program, a semantic extractor, BISE, into our intermediate formalism HPIOM, a number of optimizations in the component BIRTH, and code generators for provers like LUSTRE and SMV.

LUSSY has been designed to have as few limitation as possible regarding the way the input program is written. PINAPA uses a novel approach to extract the information from the SystemC program, and the semantic extraction implements several TLM constructs that have not been implemented in any other SystemC verification tool as of now. It doesn't require any manual annotation. The tool chain is completely automated.

LUSSY is currently able to prove properties on small platforms. Its components are reusable to build compositional verification tools, or static code analyzers using techniques other than model-checking that can scale up more efficiently.

We present the theoretical principles for each step of the transformation, as well as our implementation. The results are given for small examples, and for a medium size case-study called EASY. Experimenting with LUSSY allowed us to compare the various tools we used as provers, and to evaluate the effects of the optimizations we implemented.

ACM Classification: B.6.3, D.2.4, D.3.1, F.4.3, F.3.1
Keywords: System-on-a-Chip, Formal verification, compilation, SystemC, LUSTRE, SMV, model-checking, semantics, TLM.

Résumé

Les travaux présentés dans ce document portent sur la vérification de modèles de systèmes sur puce, au niveau transactionnel (TLM). Nous présentons le niveau transactionnel et ses variantes, et rappelons en quoi ce nouveau niveau d'abstraction est aujourd'hui nécessaire en plus du niveau de transfert de registre (RTL) pour répondre aux contraintes de productivités et de qualités de plus en plus fortes, et comment il s'intègre dans le flot de conception.

Nous présentons un nouvel outil, LUSSY, permettant la vérification formelle de modèles transactionnels écrits en SystemC. Sa structure interne s'apparente à celle d'un compilateur: Une partie frontale, PINAPA, qui lit le programme source, une extraction de la sémantique, BISE, dans notre formalisme intermédiaire HPIOM, une série d'optimisations dans le composant BIRTH, et des générateurs de code pour les outils de preuves pour LUSTRE et SMV.

LUSSY est conçu et écrit de manière à avoir aussi peu de limitation que possible sur la forme du code SystemC accepté en entrée. PINAPA utilise une approche innovante qui lui permet de s'affranchir de la plupart des limitations dont souffrent les outils similaires. L'extraction de la sémantique implémente plusieurs constructions TLM qu'aucun autre outil disponible aujourd'hui ne gère. Il ne demande pas d'annotation manuelle du code source, toute la chaîne étant entièrement automatisée.

LUSSY est capable de prouver formellement des propriétés sur des modèles de petites taille, et ses composants sont réutilisables pour des outils de preuve compositionnelle, ou d'analyse de code autre que le model-checking qui passeront mieux à l'échelle que l'approche actuelle.

Nous présentons les principes de chaque étape de la transformation, ainsi que notre implémentation. Les résultats sont donnés pour des exemples simples et petits, et pour une étude de cas de taille moyenne, EASY. Les expérimentations avec LUSSY nous ont permis de comparer les différents outils de preuves que nous avons utilisés, et d'évaluer l'efficacité des optimisations que nous avons implémentées.

Classification ACM : B.6.3, D.2.4, D.3.1, F.4.3, F.3.1
Mots Clés : Systèmes sur puce, Vérification formelle, compilation, SystemC, LUSTRE, SMV, model-checking, sémantique, TLM.