A Framework for System Level Verification: The SystemC Case

Ali Habibi

A Thesis

in

The Department

of

Electrical and Computer Engineering

Presented in Partial Fulfillment of the Requirements

for the Degree of Doctor of Philosophy at

Concordia University

Montréal, Québec, Canada

September 2005

Your file Votre référence
ISBN: 0-494-09974-7
Our file Notre référence
ISBN: 0-494-09974-7

Abstract

Ali Habibi, Ph.D.
Concordia University, 2005

Recent advances in hardware design has enabled integration of a complete yet complex systems on a single chip (called System-on-a-Chip: SoC). It is conceivable that the role of traditional Register Transfer level (RTL) languages will diminish to an extent akin to assembly level languages in software design. Therefore, new design languages or so-called System Level Languages (SLL) have emerged. Verification techniques for SOC designs also need to change with this trend. Combining classical verification techniques, such as simulation, with several other formal techniques, into a single approach has been gaining attention in SoC verification.
Classical simulation based verification techniques when used with SystemC face several problems related to the object-oriented aspect of SystemClibrary and due to the complexity of its simulation environment. In this talk, we present our proposed methodology to verify SoC designs modeled in SystemC. To this end, we introduce a hybrid approach combining static code analysis, model checking and assertion based verification. We also propose to augment the approach by a test generation module in order to improve the coverage metrics in comparison to the classical simulation approach (mainly based on random test generation).

To Laila, Yahya and my Parents

Acknowledgments

I have been very fortunate to have Dr. Sofiène Tahar as my supervisor. I am deeply grateful for his strong support and encouragement through out my Ph.D studies. His expertise and competent advice have shaped the character of my research.

Throughout my study in Concordia many people have encouraged and helped me through many obstacles. I have enjoyed studying and working with my colleagues in the Hardware Verification Group in Concordia University, wishing to thank all of them for their support and the nice time we have spent together.

I would like to express my gratitude and thanks to Dr. Börger for accepting to be my external examiner. I could not have a better expert than him worldwide.

I also wish to express my gratitude to the examination committee members, Dr. Grogono, Dr. Ait Mohamed, and Dr. Kharma, for reviewing my thesis and giving me invaluable feedback.

I would like to reserve my deepest thanks to my family for their perpetual love and encouragement. I can never thank them enough.

Table of Contents

Abstract iii

Ackmowledgments vii

Table of Contents ix

List of Tables xvii

List of Figures xix

List of Acronyms xxiii

1 Introduction 1

1.1 Motivation 1

1.1.1 System Level Languages 4

1.1.2 Problem Description 6

1.2 Verification Approaches: State-of-the-Art 7

1.2.1 Simulation 7

1.2.2 Formal Verification 8

1.2.3 Semi-Formal Verification 9

1.3 Proposed Verification Framework 11

1.3.1 Static Analysis by Abstract Interpretation 12
1.3.2 Model Checking 15
1.3.3 Assertion Based Verification 16
1.3.4 Code Coverage Analysis 16
1.4 Thesis Contributions 17
1.5 Overview of the Thesis 20

2 Preliminaries 23
2.1 Notation and Basic Definitions 23
2.1.1 Partial orders 23
2.1.2 Functions 24
2.1.3 Fixpoints 25
2.1.4 Traces 26
2.2 Abstract State Machines 26
2.3 SystemC 28
2.3.1 Structure and Hierarchy 30
2.3.2 Data Types 31
2.3.3 Functionality and Concurrency 32
2.3.4 Time and Clocks 33
2.3.5 SystemC Update 33
2.3.6 SystemC MUTEX 34
2.3.7 Request Grant Protocol 34
2.3.8 Design Rules 35
2.3.9 Language Design 35
2.4 Transaction Level Modeling 36

3 Soundness of the SystemC to AsmL Transformation 39
3.1 Introduction 39
3.2 SystemC Denotational Semantics 41

3.2.1 Syntactical Domains 41

3.2.2 Fixpoint Semantics 43

3.3 AsmL Denotational Semantics 54

3.3.1 Syntactical Domains 54

3.3.2 Semantical Domains 55

3.3.3 Fixpoint Semantics 56

3.3.4 Soundness and Correctness of the Class Semantics 61

3.4 Program Transformation 65

3.5 Summary 68

4 Static Code Analysis **69**

4.1 Introduction 69

4.2 Abstraction Framework for OO Languages 69

4.2.1 Constructing Collecting Semantics 72

4.2.2 Constructing Abstract Semantics 78

4.3 SystemC Static Analysis 82

4.3.1 Constructing Concrete Semantics 83

4.3.2 Constructing Abstract Semantics 88

4.3.3 Analysis Tactics 92

4.3.4 Illustrative Example 94

4.4 Application: Packet Switch 97

4.5 Summary 103

5 Model Checking **105**

5.1 Introduction 105

5.2 Direct Approach 105

5.2.1 Abstraction 106

5.2.2 Model Checking 107

5.2.3 Results and Discussion 108

5.3 Related Work 109
5.4 AsmL based Approach 110
5.4.1 Model Checking Technique 110
5.4.2 Illustrative Example: Packet Switch 112
5.5 Application: AGP Bus 114
5.5.1 AGP Bus Properties Specification 114
5.5.2 Model Checking 116
5.5.3 Experimental Results and Discussion 118
5.6 Summary 120

6 Assertion Based Verification 121

6.1 Introduction 121
6.2 Extending SystemC by SVA 122
6.2.1 System Verilog Assertions 122
6.2.2 Extending SystemC by SVA 124
6.3 Extending SystemC by PSL Assertions 129
6.3.1 Related Work 132
6.3.2 Embedding PSL in AsmL 133
6.3.3 Illustrative Example I: Packet Switch 134
6.3.4 Illustrative Example I: Simple Bus 137
6.4 Application: PCI Bus 139
6.4.1 Assertions Specification 140
6.4.2 Experimental Results and Discussion 141
6.5 Summary 141

7 Enhanced Simulation Coverage 143

7.1 Introduction 143
7.2 Related Work 144
7.3 Proposed Methodology 145

7.3.1 AsmL TLM Models 148
7.3.2 FSM Generation 149
7.3.3 Assertion Scope 151
7.4 Coverage Evaluation 152
7.4.1 State Coverage 153
7.4.2 Transition Coverage 154
7.5 Enhancing the Coverage 156
7.5.1 Genetic Algorithm 156
7.5.2 Fitness Criteria 158
7.5.3 Coverage Preserving From TLM to RTL 160
7.6 Application: LA-1 Interface 162
7.6.1 Coverage Results 162
7.6.2 Discussion 163
7.7 Summary 164

8 SystemC Verification Methodologies 167

8.1 Introduction 167
8.2 Top-Down Design for Verification 167
8.3 Bottom-Up Verification Methodology 168
8.4 Static Code Analysis 169
8.5 Modeling PSL Properties 170
8.5.1 UML Model 170
8.5.2 ASM Model 172
8.6 Modeling SystemC 172
8.6.1 ASM Model 172
8.6.2 Translation to C++ 175
8.7 Verification Techniques 175
8.7.1 Model Checking 176
8.7.2 Assertion Based Verification 177

8.8 Summary 178

9 Conclusions and Future Work 179

9.1 Conclusions 179

9.2 Future Work 180

A System Level Languages 181

A.1 Reuse of Existing Hardware Languages: SystemVerilog 181

A.2 C/C++ Based Approach 182

A.3 Java-Based Proposals 185

A.4 Developing a New Language: SuperLog 186

A.5 Discussion 187

B Embedding PSL in AsmL 189

B.1 Boolean Layer 189

B.2 Temporal Layer 191

B.3 Verification Layer 192

B.4 Modeling layer 193

C Description of Case Study Models 195

C.1 Packet Switch 195

C.2 Simple Bus 196

C.3 PCI Bus 198

C.4 AGP Bus 200

C.5 Look-Aside Interface (LA-1) 202

Bibliography 204

Biography 217

Index 223

List of Tables

5.1 Model Checking Results. 109
5.2 Verification Platform. 109
5.3 Validity of Initialization Conditions. 118
5.4 Model Checking (AGP Mode). 119
5.5 Model Checking (PCI mode). 119

6.1 SVA Sequence Operations. 127
6.2 Simulation Results. 142

7.1 State Space Assertions' Coverage Analysis 163
7.2 Transitions Space Assertions' Coverage Analysis 163

List of Figures

1.1 Hardware Design Evolution [82]. 2

1.2 SoC Verification Framework. 13

2.1 SystemC Simulation Methodology [84]. 29

2.2 SystemC Language Architecture [84]. 36

2.3 Defined System Models at Different Abstraction Levels [13]. 37

3.1 Online Program Transformation. 40

4.1 Abstraction Methodology. 71

4.2 Collecting Semantics. 73

4.3 Hypergraph General Structure. 79

4.4 C++ Illustrative Program. 82

4.5 Possible Generated Hypergraphs 82

4.6 Example of a Generic SystemC Program Structure. 84

4.7 SystemC Module. 86

4.8 SystemC Simulation Semantics. 88

4.9 SystemC Example. 89

4.10 Class Members Abstraction. 90

4.11 Instruction Abstraction. 91

4.12 SystemC Base Module Abstraction. 92
4.13 General Structure of the Abstract Environment. 93
4.14 SystemC Example. 94
4.15 Initial Hypergraph. 95
4.16 Iteration 1: SystemC Transformation Unit Hypergraph. 96
4.17 Hypergraph of the Active Methods. 97
4.18 Final Reduced Hypergraph. 98
4.19 Packet Switch: Initial Hypergraph. 99
4.20 Hypergraph of the Packet Switch Main Method. 100
4.21 Reduced Hypergraph of the Switch Main Method. 101

5.1 Cascading Abstract Interpretation with Model Checking 106
5.2 Hypergraph of the Simple Bus Structure. 107
5.3 Verification of PSL for SystemC Designs in ASM. 111

6.1 Mapping between SystemVerilog assertions and SystemC objects. . . 123
6.2 Extending SystemC by SVA. 125
6.3 Extending SystemC by PSL Assertions 131
6.4 Integrating Assertion $A1$ with SystemC Model of the Packet Switch . 136
6.5 Simple Bus Modes: (a) Blocking Mode (b) Non-Blocking Mode. . . . 138
6.6 Definition of the PSL Property P1 in AsmL. 139

7.1 Proposed Methodology. 146
7.2 Light Simulation Manager in AsmL. 150

8.1 Top-Down Design and Verification Methodology. 168
8.2 Bottom-up Verification Methodology. 169
8.3 Example of a Modified UML Sequence Diagram. 172
8.4 An Example of an ASM Model. 174

B.1 Partial Class Diagram for Embedding PSL in ASM. 190

B.2 AsmL Embedding of PSL BitVector. 191
B.3 AsmL Embedding of PSL SERE. 192
B.4 AsmL Embedding of PSL Verification Layer. 193

C.1 Switch Structure 196
C.2 Packet Structure 196
C.3 Simple Bus Structure. 197
C.4 PCI Bus Structure. 199
C.5 AGP Access Queuing Model [58]. 201
C.6 Look-Aside Interface (4 Banks) [79]. 203

List of Acronyms

ABV	Assertion Based Verification
AGP	Accelerated Graphics Port
AI	Abstract Interpretation
API	Application Program Interface
ASM	Abstract State Machines
AsmL	Abstract state machines Language
Asmlt	Abstract state machines language tool
BDD	Binary Decision Diagram
CAD	Computer Aided Design
DMA	Direct Memory Access
EDA	Electronic Design Automation
FPGA	Field-Programmable Gate Array
FSM	Finite State Machine
GA	Genetic Algorithm
GCC	GNU C Compiler
GNU	GNU's Not UNIX
HDL	Hardware Description Language
HOL	Higher Order Logic

IEEE	Institute of Electrical and Electronics Engineers
JVM	Java Virtual Machine
IP	Intellectual Property
LA-1	Look-Aside Interface
NPU	Network Processing Unit
OO	Object Oriented
OVL	Open Verification Library
PCI	Peripheral Component Interconnect
PE	Processing Elements
PSL	Property Specification Language
QDR	Quad Data Rate Memory
RAM	Random Access Memory
RTL	Register Transfer Level
SCV	SystemC Verification Library
SDRAM	Synchronous Dynamic Random Access Memory
SERE	Sequential Extended Regular Expressions
SLL	System Level Languages
SMV	Symbolic Model Verifier
SRAM	Static Random Access Memory
TLM	Transaction Level Modeling
UML	Unified Modeling Language
VLSI	Very Large Scale Integration
VIS	Verification Interacting with Synthesis

Chapter 1

Introduction

1.1 Motivation

A decade ago, the EDA industry went progressively from gate level to register-transfer level (RTL) abstraction. This is one of the basic reasons why this process gained a great increase in the productivity. Nowadays, an important effort is being spent in order to develop System Level Languages (SLL) and to define new design and verification methodologies at this level of abstraction.

The reason for all this activity is simple. RTL hardware design is too low as an abstraction level to start designing multimillion-gate systems (as shown in Figure 1.1). What is needed is a way to describe an entire system, including embedded software and analog functions, and formalize a set of constraints and requirements – all far beyond the capabilities of existing HDLs. VHDL and Verilog both will become the assembly languages of hardware design. Designers and verifiers will write RTL code for things that are performance-critical, nevertheless, for everything else, they will stop at a higher level.

By looking at the specifications of embedded systems, particularly for communications, portable and multimedia equipment, we can realize an important and rapid

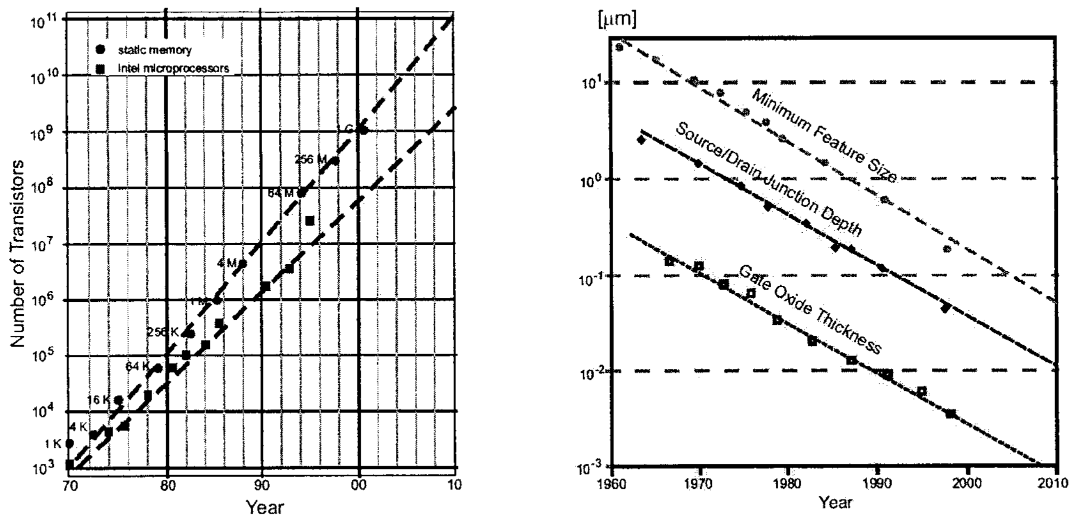


Figure 1.1: Hardware Design Evolution [82].

growth in complexity that will require System-on-a-Chip (SoC) solutions that generally integrate diverse hardware and software. Time-to-market and cost also need to be reduced more than ever before and backed up by an effective marketing-driven strategy that can meet today's highly competitive and demanding circumstances. To achieve all this, the product development process must assure the product specification phase is integrated smoothly with the product design phase, allowing the customer's demands, marketing goals and designer expertise, to be evaluated and analyzed at significantly less cost in time and resources, and to be rapidly incorporated into the final product.

State-of-the-art SLL proposals can be classified into four main classes. First, reusing existing software SLL such as UML. Second, extending classical hardware languages such as extending Verilog to SystemVerilog [57]. Third, readapting software languages and methodologies (C/C++ [88], Java [6], etc.). Third, creating new languages specified for system level design (Rosetta [3] for example).

The SystemC SLL [80] is expected to have a stronger effect in the area of architecture, the co-design and integration of hardware and software [101]. The SystemC library of classes and simulation kernel extend C++ to enable the modelling of systems. The extensions include support for concurrent behavior, a notion of time sequential operations, data types for describing hardware, structure hierarchy and simulation. The core language consists of an event-driven simulator working with events, processes, modules, ports, interfaces and channels.

The verification of SoC is a more serious bottleneck in the design cycle. In fact, defining an SoC design language and methodology is a matter of time, however, the verification is a very open and ambiguous question. Classical functional verification is consuming an inordinate amount of the design cycle time. Estimates vary, but most analysts and engineers agree that as much as 70 percent of the design cycle is consumed by functional verification. In addition, the quality of these verification efforts has become more important than ever because the latest silicon processes are now accompanied by higher re-spin costs. No doubt, classical random simulation is no more able to handle actual designs. Going further in complexity and considering hardware/software systems will be out of the range of the currently used simulation based techniques [64].

Classical verification techniques when used with SystemC will face several problems related to the object-oriented aspect of this language and to the complexity of its simulation environment. In order to solve this problem, we propose a verification methodology based on abstract interpretation. The objective of our endeavor is to build an abstracted representation of the SystemC program that can be used for both model checking and static code analysis. This latter is a well-known technique that has been applied in the past for both the software and the hardware areas[4]. It tries to get an abstract representation of the system and use it to verify some properties such as loops non-termination. However, some proposals try to present a so-called total program analysis [9]. This refers to an analysis that will allow the extraction of all program properties.

For instance, the main trends in defining new verification methodologies are considering a hybrid combination of formal, semi-formal and simulation techniques.

The first step in this direction was the adoption of Sugar language [5] from IBM as a standard for hardware formal specification. The new standard is recalled the Property Specification Language (PSL) [1].

A lot is expected from combining an assertion language such as Sugar with both smart test generation and coverage analysis. This kind of hybrid techniques can offer a partial answer to the question: "Is the verification task complete?" However, an answer to a question like "Is a property always true?" can be only answered by purely formal techniques such as theorem proving [66] and model checking [86]. This latter, despite its problem of of state explosion, is gaining a lot of interest in both academic and industrial areas. A number of proposals offer to abstract the system in order to verify some of its properties using model checkers and then complete the verification process by classical simulation techniques.

1.1.1 System Level Languages

System level design [93], requires system level tools that simultaneously handle both hardware and software, for modeling, partitioning, verification and synthesis of the complete system. The software evolution story is replayed again in the hardware world. Migrating to the system level of abstraction introduces a higher order of complexity. To reach the next level of complexity, EDA vendors and analysts are telling designers that another leap is necessary – from RTL to system-level design. Such a leap implies that an increasing amount of hardware design will be done using C/C++, Java, or other high-level languages, while RTL (VHDL and Verilog) will be relegated to smaller blocks of timing critical logic.

Nevertheless, the highest level of abstraction will be the one that survives, and that is clearly the software domain. Considering the economic reality, this transition will not be abrupt, but it will occur more as an evolution than a revolution. The most likely transition will be along the lines that software followed as it evolved from a strict use of hand-coded assembler in the fifties to extensive use of compilers in the sixties. The most realistic scenario will start by migrating the non-critical portions of time-to-market-driven designs to higher levels. Then, progressively over time, more sophisticated compiler and synthesis technology augmented by increasing hardware functionality will extend the reach of automatic techniques until only extremely critical portions of highly performance-driven designs will be implemented at the register transfer level.

Eventually, the software and hardware design will end by getting into a single flow. Optimistically, in a few years, the difference, if it will exist, will be a matter of compiler options ("-software" or "-hardware"). However, to get to there, we need at first to define system level languages and design methodologies.

The usage of a SLL is a direct result of the way the SoC design process works [59]. Following the software evolution, the most likely methodology would be to push toward successive refinement, an actual successful methodology in software development. A possible solution is to define what are the basic requirements for a system level language; intuitively, we would first have a look to the way software design is actually performed. No doubt, a short-term solution will come from languages that"push up" from or extend current HDL-based methodologies. We cannot omit that languages like SuperLog [96] hold great promise for the next three to four years basically because SuperLog does not require a radical shift in methodologies and allows groups to retain legacy code. This approach can be seen as a refinement of the existent languages. Some people define SuperLog as "Verilog done right". The medium-range solutions will likely be C++ based languages that have hardware design capabilities, we mention here SystemC [81] and Cynlib [29] as two promising languages. The revolution may also come from the Java side. Long-term language solutions will likely come from new languages developed just specifically to work at the system level such as Rosetta [3] which promises to make true system level design a reality [1].

[1]A more detailed description of state-of-the-art SLL proposals is provided in Appendix A.

1.1.2 Problem Description

As SoC designs become a driving force in electronics systems, current verification techniques are falling behind at an increasing rate. A verification methodology that integrates separate but key technologies is needed to keep up with the explosive complexity of SoC designs [64]. An SoC verification methodology must address many more issues than were prevalent even a couple years ago, in particular the integration of purchased and in-house Intellectual Property (IPs) into new designs, the coupling of embedded software into the design, and the verification flow from core to system. Several key concepts are important to understand, including the transition from core to system verification, the re-use and integration of multiple sources of cores, and the support needed to optimize core re-use.

There are two major problem areas with SoC verification today that keep the verification as a true bottleneck: IP core verification, and the System Level Verification (SLV). The verification of today's IP cores tends to be inward focused. For example, the verification of a Peripheral Component Interconnect (PCI) IP core would test the bus modes and address space. This verification is useful, and helps provide information on the core functionality to the IP integrator. However, all this verification does not help a great deal at the system level, when the PCI core is connected to the rest of the system. How should the software team write drivers involving this IP core? What if the core needs modification? Will there be any architectural issues that arise when it is too late to change the core? All these make IP use, and reuse, challenging.

Most of todays research effort is spent in defining new verification *methodologies* combining both simulation and formal methods. Several state-of-the-art projects offer such a combination (SLAM [4] from Microsoft, BANDERA [33] from Kansas State University, etc.). They generally use abstract interpretation [26], model checking [86] and assertion based verification [78] techniques in order to guide and to improve random simulation.

If hybrid solution combining both simulation and formal methods is being a *de facto* for SoC verification, there is a number of open questions, mainly:

- At which abstraction level the verification has to start?
- What are the more suitable techniques that can be used for SoC verification?
- How to combine formal methods with simulation in an efficient verification flow?
- How to improve the coverage metrics?

1.2 Verification Approaches: State-of-the-Art

1.2.1 Simulation

Today, the usual validation method to discover errors in SoC is still simulation. In this method, a simulation run must be performed in each level of abstraction such as transaction, RT and gate level to check if the required characteristics are preserved. With simulation, input signals are injected at certain points in the system and the resulting signals at other points are observed. These methods can be a cost-efficient way to find errors. However, in order to get full confidence in the design we would have to perform a complete simulation which covers all possible input combinations. Exhaustive simulation of even moderately-sized circuits is impossible, and partial simulation offers only partial assurance of correctness. This is an especially serious problem in safety-critical applications, where failure due to design errors may cause loss of life or extensive damage. In these applications, functional errors in circuit designs cannot be tolerated. But even where safety is not the primary consideration, there may be important economic reasons for doing everything possible to eliminate design errors, and to eliminate them early in the design process. A flawed design may mean costly and time-consuming re-fabrication, and mass-produced devices may have to be recalled and replaced.

1.2.2 Formal Verification

A solution to these problems is one of the goals of formal methods [18] for verification of the correctness of SoC. With this approach, the behavior of the system is described mathematically, and formal proof is used to verify that they meet rigorous specifications of intended behavior. However, formal verification is not the golden rule in system verification because of some limitations. A correctness proof cannot guarantee that the real device will never malfunction; the design model of the device may be proved correct, but the resulting hardware/software system actually built can still behave in a way unintended by the designer (this is the case for simulation too). Wrong specifications can play a major role in this, because it has been verified that the system will function as specified, but it has not been verified that it will work correctly. Defects in physical fabrication can cause this problem too. In formal verification, a model of the design is verified, not the real physical implementation. Therefore, a fault in the modeling process can give false negatives (errors in the design which do not exist). Although sometimes, the fault covers some real errors. Because of these limitations we can consider simulation and formal verification as complementary techniques, the methods have to play together.

Formal verification methods can be categorized in two main groups: theorem proving [46] and static analysis [107]. Theorem proving refers to the use of axioms and proof rules to prove the correctness of the systems. In this method, one expresses the system model and specifications in a suitable logic, and constructs a proof in the logic that the system model implies the specifications. The powerful mathematical techniques such as induction and abstraction are the strengths of theorem proving and make it a very flexible and powerful verification technique. The used formal logics make it possible to construct a model at almost every abstraction level and proves properties on all classes of systems. However, it is a time consuming process which can involve generating and proving literally hundreds of lemmas in painstaking detail.

Static analysis is a family of formal methods for automatically deriving information about the behavior of a software or hardware system. Several applications of static analysis include automated debugging, invariant checking, code optimization, etc. Briefly, program analysis – including finding possible run-time errors – is undecidable: there is no mechanical method that can always answer truthfully whether programs may or not exhibit runtime errors. This is a mathematically founded result dating from the works of Church [15] and Turing [103] in the 1930's. There exist two main families of formal static analysis: model checking [17] and static code analysis by abstract interpretation [22].

Model checking considers systems that have finite state or may be reduced to finite state by abstraction. In comparison to theorem proving, it is more limited in scope, but is fast and fully automated. The system model is in essence a finite state machine, and specifications are written in temporal logic. These logics are limited with respect to the very powerful logics handled by general theorem provers, but are quite simple and concise, and can express a wide variety of useful properties.

Static code analysis by abstract interpretation, on the other hand, approximates the behavior of the system by considering more behaviors than can happen in reality. In theory, it is not limited in scope (state space explosion, for example). But, it may get false alarms due an over-approximation of the system.

1.2.3 Semi-Formal Verification

Although full model checking is not yet ready for widespread deployment, the idea of writing and checking properties seems to be catching on, where the checking is done by simulation, possibly *amplified* by some kind of formal method. Several industrial tools have been developed to promote automatically generating checkers (e.g., 0-In Check [54] and Specman Elite [106]). Usually, these checkers concern properties like conformance to input/output protocols, correct management of FIFOs, etc. They are specified with high level directives expressed as comments in the source or as external monitors. If a property is violated during simulation then the checker *fires* and reports a bug.

Assertions (or constraints) may be expressed either *declaratively* or *procedurally.* A declarative assertion is always active, and is evaluated concurrently with other components in the design. A procedural assertion, on the other hand, is a statement within procedural code, and is executed sequentially in its turn within the procedural description.

The most important step towards setting Assertion Based Verification (ABV) as a dominant technique for SoC is the standardization of the property specification language (PSL) [1]. A PSL specification consists of assertions regarding properties of a design under a set of assumptions. A property is built from Boolean expressions, which describe behavior over one cycle, sequential expressions, which describe multi-cycle behavior, and temporal operators, which describe relations over time between Boolean expressions and sequences. PSL provides a means to write specifications that are both easy to read and mathematically precise. It is intended to be used for functional specification on the one hand and as input to functional verification tools on the other.

Another issue with assertion based verification is coverage. How do we know that the verification process is complete? There is both functional or specification coverage and implementation coverage. Metrics for the latter include toggle coverage, code coverage (line, branch and sub-expression coverage), and finite state machine (FSM) coverage (state, arc and path). While easy to measure, it is difficulty to correlate these measures with coverage of functional intent. Functional coverage metrics, including coverage points, are designed to check that the verification tests have exercised specific key functionality – for example, executing all instruction types or transmitting and receiving all packet types. However, functional coverage on the chip inputs and outputs may miss the exercise of internal structures likes FIFIOs.

1.3 Proposed Verification Framework

To the previously discussed open issues facing SoC verification there is no magical solution. There are ongoing proposals sharing relatively similar concepts and diverging in the realization phase. In other terms, to overpass the question of system complexity, for example, a classical solution is to *abstract* the system.

Before going further to any verification proposal, the first step is to define the abstraction level at which the verification is performed. No doubt, the classical way was, for hardware, to consider mainly the RT level. However, as previously discussed this level is no longer suitable for SoC designs. We propose, in this project, to focus on the system level. This choice is motivated by the fact that existing commercial tools offering formal verification techniques such as model checking (FormalCheck [70] from Synopsys,@Verifier [42] from @HDL, etc.) and equivalence checking (e.g., FormalPro from Mentor Graphics [91]) are operational for lower design levels and fail when considering complex systems or a high level of abstraction these tools cannot do much.

Reusing what exists in terms of formal methods and techniques requires abstracting the system before going to any verification technique. Abstraction here means simplifying the system in order to verify some of its properties. For sure, there may be a loss of information about the system. Nevertheless, the most important is to get some aspects of the system well covered and then complement the verification flow by simulation. There exist a number of abstraction approaches, among them abstract interpretation [24] is one of the most relevant techniques, which proved to be quite successful when dealing with large scale systems and various kinds of languages [26].

To explain the concept of our proposed methodology we consider the classical way a programmer deals with semantical errors in his program. If we consider a code that compiles well and then gives wrong output, a classical way to solve such a problem comes in three steps:

- First, analyze the source code if there is some coding errors; e.g., non-initialized variables, wrong procedure call, etc.
- Then, if no errors found, verify if the values of certain variables are equal to what is expected.
- Finally, if the problem is not solved, randomly test the program in order to localize the source of the error.

Figure 1.2 describes the proposed SoC verification framework which is composed from two proposals: AsmL based approach and direct approach. The link between the two paths is established using a syntactical transformation from SystemC to AsmL and vice-versa. The correctness of this transformation guarantees the validity of the results obtained using AsmL for the original SystemC design.

In the direct approach, the SystemC design is abstracted using abstract interpretation. The resulting reduced model is used: (1) for model checking a set of design's properties; and (2) to narrow the test space generation of the best test generator for a set of assertions integrated with the original design as external monitors.

In the AsmL based approach, both the design and the property are modelled in AsmL. We adapted a reachability analysis algorithm inside the Asmlt tool to perform the model checking of the design's PSL properties. The same algorithm is also used to generate the system's FSM which serves as a reference model to evaluate the coverage by simulation of the PSL assertions (compiled using the AsmL compiler to C#).

1.3.1 Static Analysis by Abstract Interpretation

Abstract interpretation is a theory of sound approximation of the semantics of computer programs, based on monotonic functions over ordered sets, especially lattices [26]. It can be viewed as a partial execution of a computer program which gains information about its semantics (e.g. control structure, flow of information) without performing all the calculations. Its main concrete application is formal static analysis,

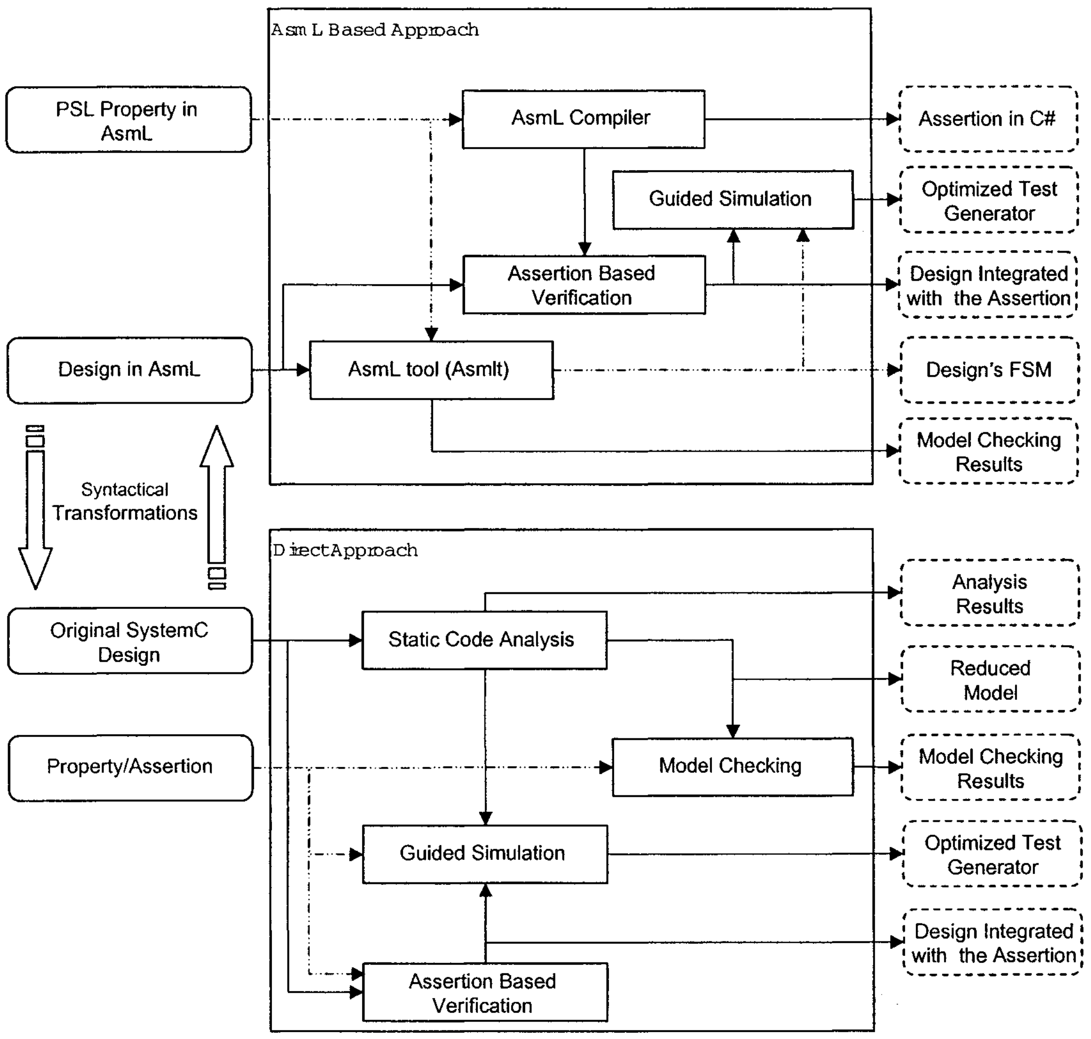


Figure 1.2: SoC Verification Framework.

the automatic extraction of information about the possible executions of computer programs; such analysis has two main usages:

- inside compilers to analyze programs in order to decide whether certain optimizations or transformations are applicable
- for debugging or even the certification of programs against classes of bugs.

Given a programming or specification language, abstract interpretation consists in giving several semantics linked by relations of abstraction. The most precise semantics, describing very closely the actual execution of the program, is called the concrete semantics. For instance, the concrete semantics of an imperative programming language may associate to each program the set of execution traces it may produce - an execution trace being a sequence of possible consecutive states of the execution of the program; a state typically consists of the value of the program counter and the memory locations (stack and heap). More abstract semantics are then derived; for instance, one may consider only the set of reachable states in the executions (which amounts to considering the last states in finite traces).

To apply static analysis, some computable abstract semantics must be derived at some point. For instance, one may choose to represent the state of a program manipulating integer variables by forgetting the actual values of the variables and only keeping their signs (+, - or 0). For some elementary operations, such as multiplication, this abstraction does not lose any precision: to get the sign of a product, it is sufficient to know the sign of the operands. For some other operations, the abstraction may lose precision: for instance, it is impossible to know the sign of a sum whose operands are respectively positive and negative. Such loss of precision may not, in general, be avoided so as to make a decidable semantics (Rice's Theorem [87]). There is, in general, a compromise to be made between the precision of the analysis and its tractability, either from a computability point of view or from a complexity point of view.

In this thesis, we propose a framework based on abstract interpretation, where the program's memory (stack and heap), execution environment and the program itself are described in terms of graphical entities called hypergraphs [105]. Our target is to represent graphically an abstracted version of the program; then to analyze it statically. In a second step, we use the abstracted model (which is supposed to be less complex that the concrete system) to verify formally some properties, through model checking and assertion based verification. Finally, as the previous steps may not cover all cases or may fail to return an answer (e.g., non-termination), it is necessary to augment the whole approach by a simulation phase. This latter is called here *smart test generation* because we intend to consider the information collected in the previous steps in order to improve the coverage metrics.

1.3.2 Model Checking

There are two main state exploration verification techniques used in the EDA industry. The first is *equivalence checking* which refers mainly to a comparison of an RTL version of the design to its gate-level equivalent, or the comparison of two gate-level netlists. The other technique is *model checking* [86] which is concerned with properties verification mainly at the RTL.

Model checkers are the most adequate formal technique to be used at the system level design. With this technique there are no corner cases, because the model checker examines 100% of the state space without having to simulate anything. However, this does mean that model checking is typically used for small portions of the design only, because the state space increases exponentially with complex properties and quickly runs into a *"state space explosion"*.

For instance, there are no *new* model checkers adapted for system level design of the SoC domain. Nevertheless, what is interesting about these techniques is the definition of hierarchical verification allowing the use of the checkers for small design portions.

The most relevant progress for the future use of model checkers for SoC verification is the selection of the Property Specification Language Sugar (PSL) [1] (called Sugar [44] before being taken as a standard by Accellera) as a new standard for formal properties and systems specification. PSL is described as "a declarative formal property specification language combining rigorous formal semantics with an easy to use style" [35]. A PSL specification can also be used to automatically generate simulation checkers, which can then be used to check the design using simulation.

1.3.3 Assertion Based Verification

Assertions are higher abstraction mechanisms that concisely capture design specification. They drive dynamic simulation and formal analysis to pinpoint bugs faster. They are a useful way to represent design specifications that are readable and reusable by multiple tools in the verification flow. HDL simulation tools use assertions to dynamically run *checkers* and *monitors* during simulation. Functional coverage tools analyze simulation activity and provide information on coverage of functional test plans by reporting on coverage of assertions. Assertions are also used as properties that formal analysis engines can use to exhaustively analyze and prove or disprove, greatly enhancing verification confidence.

In the methodology proposed in this thesis, we integrate assertion based verification as a component of the verification framework. For instance, we embedded the whole specification of PSL in AsmL (Abstract state machines Language) [51]. Then, we compile PSL assertions into C# code using the Asmlt tool [74], and integrate them with the original design. Assertions are then verified by simulating the new model that combines the original design and the integrated assertions. This enriches the design language with a powerful and expressive assertion specification layer, and improves the verification procedure by targeting specific properties during simulation.

1.3.4 Code Coverage Analysis

Coverage has become a key technology in the pursuit of efficient and accurate verification of large designs [64]. The easiest form of coverage to introduce into a verification methodology is RTL coverage. Tools are available that, given an existing RTL design and a set of vectors, will provide valuable information about the coverage [36].

Often overlooked, the first limitation of RTL coverage tools is that they do not know anything about what the design is supposed to do. Therefore, the tools can only report problems in the RTL code that has been written. There is no way for these tools to detect that some code is missing. In fact, some bugs are due to incorrectly written RTL, while others are due to RTL that is simply not there.

The second limitation of RTL coverage is the lack of a built-in formal engine. An expression coverage tool would see no problem in reporting that certain combinations of the expression inputs were not covered, even though by construction, they are mutually exclusive and therefore formally unreachable [78]. This *spurious* reporting adds to the confusion and limits the effectiveness of coverage. Constant propagation and effective dead code detection would also benefit from such an analysis.

At system level, functional coverage is the good methodology to test designs exhaustively but never being able to answer the question is verification complete? Traditionally, this metric has not been used because of the historic difficulty in implementing it. The two most complex issues are defining tests that execute the desired functionality and collecting the functional coverage data to determine the metric. Advances in directed randomization technology address the issue of creating tests. The development of tools to capture and analyze transaction–based information addresses the collecting, analysis and determination of the metric. The combination of these two advancements will reduce the complexity of implementing functional coverage metrics.

1.4 Thesis Contributions

In this thesis, we present a verification framework for system level languages with application to SystemC. We propose to combine static code analysis using abstract interpretation, model checking and assertion based verification in order to attain higher verification coverage of the design under verification. We prove the soundness of our approach by establishing an isomorphism relation between the design model in SystemC and its formal representation in AsmL (used for both model checking and automatic assertion monitors generation). Furthermore, we provide a genetic-algorithm based approach to enhance code coverage by guided simulation. A more detailed description of the thesis contributions is listed below:

- In [Bio:Jr–2,Bio:Cf–4,Bio:Cf–5,Bio:Cf–8], we present a methodology to design and verify SystemC transactional models starting from a UML system specification and integrating an intermediate ASM layer. We proposed to upgrade the UML sequence diagram in order to capture transaction related system properties. Then both of the design at its properties are modeled in ASM to enable performing model checking. On the other hand, to cover for the state explosion problem that may result due to the system's complexity, we completed our approach by offering a methodology to apply assertion based verification re-using the already defined PSL properties. To do so, we defined a set of translation rules to transform the design's model in ASM to its implementation in SystemC.

- In [Bio:Jr–1,Bio:Tr–4], we present a fixpoint semantics of the SystemC library including, in particular, the semantics of a SystemC Module that we proved to be sound and complete w.r.t. a trace semantics of a SystemC program.

- In [Bio:Jr–1,Bio:Tr–2], we present the semantics of a subset of AsmL and we proved the soundness and completeness of an AsmL class w.r.t. to a trace semantics of the AsmL program.

- In [Bio:Jr–1,Bio:Tr–1], we prove the existence, for every SystemC program, of an AsmL program having similar behavior w.r.t. an observation function that we set to consider the traces of the system just after the update phase of the SystemC simulator. We have used this SystemC to AsmL transformation to reduce the complexity of SystemC models and enabled their formal verification using model checking and theorem proving approaches used with AsmL and ASM languages in general.

- In [Bio:Cf–9,Bio:Cf–10], we extend the classical abstraction framework, which is usually restricted to the program code to cover the whole SystemC simulation environment. In the adopted methodology, the abstraction output is a complete abstract environment that we modeled in a graphical representation (hypergraphs) in order to allow better interaction between the designer and the static analysis environment. On the hypergraph entities we define reduction operations that can be used to simplify the abstract code or to transform it into a more suitable representation for the static analysis or other verification techniques.

- In [Bio:Cf–1,Bio:Cf–9], we present a bottom-up approach where starting from an existent SystemC design we generate internally a model in AsmL and verify the system property at the ASM level.

- In [Bio:Cf–7], we present a modelling of PSL in AsmL. We provide a deep embedding in AsmL of the three hierarchical layers: Boolean, temporal, and verification of PSL.

- In [Bio:Cf–10], we present a methodology to enhance assertion coverage using transaction level models as intermediate step in the design process. We use AsmL as a TLM language for the sake of automatically generating a finite state machine of the system. We define assertions as a set of states (part of the system's FSM). We introduce two assertion coverage metrics: state and transition coverage. Furthermore, we propose a genetic algorithm based approach to enhance the coverage.

- In [Bio:Cf–13], we propose a methodology to integrate SystemVerilog Assertions (SVA) support to SystemC.

- In [Bio:Cf–3,Bio:Cf–5,Bio:Cf–8,Bio:Cf–12], we illustrate the efficiency of our proposed approach to verify SystemC designs on several industrial and complex case studies including the Bus structure from the SystemC library, the PCI Bus standard, the AGP Bus standard, the Look-Aside Interface (LA-1) standard, and several other examples from the SystemC library.

1.5 Overview of the Thesis

This thesis is made up of nine chapters. Each chapter begins with an introductory paragraph and a section in which the subject of the chapter is informally introduced and compared with the existing literature. The results of Chapters 3, 4, 5, 6, 7 and 8 have been published in the proceedings of international conferences and Journals[2].

Below we sketch the content of the next chapters.

In Chapter 2 we recall some basic definitions, notations and results used throughout the thesis.

In Chapter 3, we establish a formal link between both paths of our proposed SoC verification framework by establishing a trace equivalence between the original SystemC model and its transformed AsmL model.

In Chapter 4, we first extend generic static code analysis by abstract interpretation to support the SystemC semantics. Then, we provide a graphical model for the abstract analysis and debugging of SystemC models.

In Chapter 5, we first present a direct application of the model checking technique to a an RTL model in SystemC. Then, we provide an enhanced approach for general SystemC models based on a transformation to the AsmL language.

In Chapter 6, we propose two distinct approaches to integrate SVA and PSL assertions to SystemC.

In Chapter 7, we present a genetic algorithm based approach in order to enhance the system's state space coverage using guided simulation.

In Chapter 8, we introduce two application methodologies of our proposed SoC

[2]The list of these publication is provided in the Biography Section.

verification framework for the case of SystemC. The first is a top-down approach integrated with the design process where we refine a behavioral specification to a system level design. The second is a bottom-up approach where we verify an existing design.

Finally, in Chapter 9 we conclude the thesis and propose perspectives for future work.

Chapter 2

Preliminaries

In this chapter we introduce the mathematical background used in the rest of the thesis. We fix the notation and we recall some well-known results in lattice theory and fixpoint theory. We overview the notion of Abstract State machines and the AsmL language. We also introduce existent system level languages and describe the most important features of the SystemC language.

2.1 Notation and Basic Definitions

2.1.1 Partial orders

A partial order on a set D is a relation $\sqsubseteq$ on D which is reflexive, antisymmetric and transitive. A set with a partial order defined on it is called a partially ordered set, *poset.* We denote it as $\langle D,\sqsubseteq\rangle$. A relation on D which is reflexive and transitive is called a preorder.

Given a poset $\langle D,\sqsubseteq\rangle$ and a subset U of D, we say that an element $d \in D$ is:

- an *upper bound* for U if $\forall u \in U.u \sqsubseteq d$.
- a *least upper bound* for U if it is an upper bound of U and any upper bound $d' \in U$ is such that $d \sqsubseteq d'$.

- a *largest* element of U if it is an upper bound of U.

Because of the antisymmetric property, if the largest element exists then it is unique. If it exists, we denote the largest element of D with $\top$. Lower bounds, greatest lower bounds and smallest elements are defined dually. In particular, if it exists, we denote the smallest element of D with $\bot$.

A poset $\langle D,\sqsubseteq\rangle$ is called a *lattice* if any two elements of D have both a greatest lower bound and a least upper bound. If a poset admits greatest lower bounds and least upper bounds even for infinite sets then it is a *complete* lattice. In such a case we write either $\langle D,\sqsubseteq,\bot,\top,\sqcup,\sqcap\rangle$ or, when the lattice operators are clear from the context, simply $\langle D,\sqsubseteq\rangle$. We say that the operator $\sqcup$ is *complete* if any subset U of D admits the least upper bound.

2.1.2 Functions

Given two sets D and R, a relation $f \subseteq D \times R$ is called a *function*, or a mapping, if: $\forall d \in D, \forall r_1, r_2 \in R$, $d\,f\,r_1$ and $d\,f\,r_2 : r_1 = r_2$.

If $\langle D,\sqsubseteq\rangle$ and $\langle R,\preceq\rangle$ are complete lattices, then we say that a function $f \in [D \to R]$ is:

- *monotonic* if it preserves the order of the elements: $\forall d_1, d_2 \in D. d_1 \sqsubseteq d_2 {:} f(d_1) \preceq f(d_2)$.

- a *join-morphism* if it preserves least upper bounds: $\forall d_1, d_2 \in D. d_1 \sqcup d_2 {:} f(d_1) \curlyvee f(d_2)$.

- a *complete join-morphism* if it preserves least upper bounds for arbitrary subsets of D.

- *continuous* if it preserves the least upper bound of increasing chains.

Similarly, a function f is said to be a *meet-morphism* if it preserves the greatest lower bound of two elements and a *complete meet-morphism* if it preserves the greatest lower bound for any subset of a complete lattice D.

2.1.3 Fixpoints

Given a set D and a function $f \in [D \rightarrow D]$, a *fixpoint* of f is an element $d \in D$ such that $f(d) = d$. When f is defined over a partial order $\langle D,\sqsubseteq\rangle$, an element $d \in D$ is:

- a *pre-fixpoint* if $d \sqsubseteq f(d)$;
- a *post-fixpoint* if $f(d) \sqsubseteq d$;
- the *least fixpoint* if: $d = f(d)$ and $\forall d' \in D.d' = f(d') =:d \sqsubseteq d'$;
- the *least fixpoint* if: $d = f(d)$ and $\forall d' \in D.d' = f(d') =:d' \sqsubseteq d$.

Given a function f defined over a poset $\langle D,\sqsubseteq\rangle$ and an element $d \in D$, we denote with $\mathrm{lfp}^{\sqsubseteq}_{d} f$, the least fixpoint of f w.r.t. the order $\sqsubseteq$ larger than d, if it exists. Sometimes, when the order and the element are clear from the context, we will simply write $\mathrm{lfp} f$. The definition of $\mathrm{gfp}^{\sqsubseteq}_{d} f$ is dual .

A main result of Tarski [102] is that a monotonic function defined over a complete lattice admits a least and greatest fixpoint:

Theorem 2.1.1 *(Fixpoint Theorem, Tarski [102])*
Let $\langle D,\sqsubseteq,\bot,\top,\sqcup,\sqcap\rangle$ be a complete lattice and let $f \in [D \rightarrow D]$ be a monotonic function. Then the set $F = \{d \in D | f(d) = d\}$ is a non-empty complete lattice w.r.t the order $\sqsubseteq$. Furthermore,

- $lfp^{\sqsubseteq}_{\bot} f = \sqcap\{d \in D | f(d) \sqsubseteq d\}$
- $gfp^{\sqsubseteq}_{\bot} f = \sqcup\{d \in D | f(d) \sqsupseteq d\}$

The result of the theorem is not constructive. An alternative characterization of the least fixpoint for monotonic functions defined over a complete lattice is given in the *Transfinite Iterations* theorem given in [23]).

2.1.4 Traces

Definition 2.1.1. (Traces)
Given a set Σ of states and an $\Omega \notin \Sigma$, a trace τ is a function $\tau \in [\mathbb{N} \rightarrow \Sigma \cup \Omega]$ which respects the prefix condition: $\forall n \in \mathbb{N}.\tau(n) = \Omega =: \forall i > n.\tau(i) = \Omega$.

Roughly, if a trace is undefined for an $n \in \mathbb{N}$, then it is undefined for all the successors of n too. We say that a trace τ is finite if $\exists n \in \mathbb{N}.\tau(n) = \Omega$. If not we say that it is infinite. The sets of finite traces over Σ is denoted by $\mathcal{T}(\Sigma)$.

2.2 Abstract State Machines

States in Abstract State Machines (ASM) are given as many–sorted first–order structures [9]. A structure is given with respect to a signature which is a finite collection of function names, each of a fixed arity. The given structure fixes the syntax by naming sorts and functions. An algebra provides *domains* (i.e., carrier sets) for the sorts and a suitable symbol interpretation for the function symbols on these domains, which assigns a meaning to the signature. Therefore, a state is defined as an algebra of a given signature with *domains* and an interpretation for each function symbol.

A *vocabulary* is a finite collection of function names, each with a fixed arity. Every ASM vocabulary contains the following *logic symbols*: nullary function names *true, false, undef*, the equality sign, the names of the usual Boolean operations, and a unary function name *Bool.* Some function symbols (such as *Bool*) are tagged as *relations.* A *state* S of vocabulary Γ is a non–empty set X (*the superuniverse of* S), together with interpretations of all function symbols in Γ over X. A function symbol f of arity r is interpreted as an r–ary operation over X; if $r = 0$, f is interpreted as an element of X. The interpretations of the function symbols *true*, *false*, and *undef* are distinct, and are operated upon by the Boolean operations in the usual way. The value *undef* is used to code functions whose value is outside the indicated domain.

A state transition into the next state occurs when dynamic functions change their evaluation. *Locations* and *updates* capture this notion. A *location* of a state is a pair $loc = (f, \overline{a})$, where f is a dynamic function symbol and $\overline{a}$ is a tuple of elements in the domain of the function. The element $f(\overline{a})$ at a state is the *value* of the location $(f, \overline{a})$ in that state. For changing values of locations the notion of an *update* is used. An *update* of a state is a pair $\alpha = (loc, val)$ where $loc = (f, \overline{a})$ is a location and val, the update value, is a value in the function domain. To fire an update at a state, the update value is set to the new value of the location. As a consequence, the overall dynamic function f is redefined to map the location onto the new value. Transition rules define the state transitions of an ASM. While terms denote values, transition rules denote *update sets*, which define the dynamic behavior of an ASM. At each state all update sets are fired simultaneously which causes a state change. All locations that are not referred to in the update sets remain unchanged. ASM runs, starting in a given initial state, are determined by a closed transition rule declared to be the *program*. Basic transition rules are *skip*, *update*, *block*, and *conditional rules*.
The *update* rule is an atomic rule denoted as

$$f(t_1, t_2, \dots, t_n) := t$$

It describes the change of interpretation of function f at the place given by $(t_1, t_2, \dots, t_n)$ to the current value of term t.
A *conditional rule* specifies a guarded execution.

```
$if\ guard\ then\ R1$
$else\ R2$
$endif$
```

Where *guard* is a first order Boolean term. R_1 and R_2 denote arbitrary transition rules. The condition rule is fired in state S by evaluating the guard g in S, if it evaluates to *true* R_1 fires, otherwise R_2 fires.

AsmL (the Abstract State Machine Language) [51] is a novel executable specification language based on the theory of ASM. It is fully object-oriented and has a strong mathematical component. In particular, sets, sequences, maps and tuples are available as well as set comprehension, sequence comprehension and map comprehension. ASMs steps are transactions, and in that sense AsmL programming is transaction based. AsmL is fully integrated into the .NET framework and Microsoft development tools providing inter-operability with many languages and tools.

Although the language features of AsmL were chosen to give the user a familiar programming paradigm (supporting classes and interfaces in the same way as C# or Java do), the crucial features of AsmL, intrinsic to ASM, are massive synchronous parallelism and finite choice. These features give rise to a cleaner programming style than is possible with standard imperative programming languages. Synchronous parallelism and inherently AsmL provide a clean separation between the generation of new values and the committal of those values into the persistent state.

2.3 SystemC

In this section we provide a detailed description of the SystemC components including SystemC signal, MUTEX channel, a SystemC protocol, SystemC design rule checks, and SystemC simulator. The SystemC simulation kernel does not impose any order on processes that are simultaneously ready-to-run. In our definition, we treat different kinds of notifications separately. Immediate notifications are not shown in this definition since they are implied in the execution of a method, which we treat abstractly here. Timed and delta notifications are shown below within the simulator definition. The type of events notification can be immediate.

Figure 2.1 illustrates a generic simulation methodology in the SystemC environment [84]. The SystemC model can be written at the system level, behavioral level, or RTL level using C/C++ augmented by the SystemC class library. The class library serves two important purposes. First, it provides the implementation of many types of objects that are hardware-specific, such as concurrent and hierarchical modules, ports, and clocks. Second, it contains a lightweight kernel for scheduling the processes. The design's SystemC code can be compiled and linked together with the class library with any standard C++ compiler (such as GNU's gcc), and the resulting executable serves as the simulator of the user's design. The testbench for verifying the correctness of the design is also written in SystemC and compiled along with the design. The executable can be debugged in any familiar C++ debugging environment (such as GNU's gdb). Additionally, trace files can also be generated to view the history of selected signals using a standard waveform display tool.

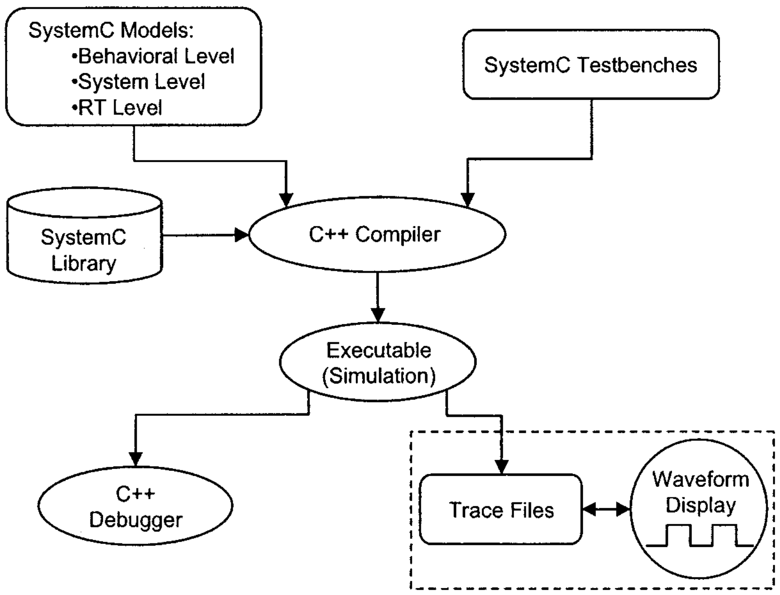


Figure 2.1: SystemC Simulation Methodology [84].

The import of a traditional software development environment into the hardware design and system design scenario entails some powerful advantages. The sophisticated program development infrastructure already in place for C/C++ can be directly utilized for the SystemC verification and debugging tasks. For hardware designers traditionally used to viewing simulation data in the form of waveform displays, the trace file generation facility provides a familiar interface. Conceptually, the most powerful feature is that the hardware, software, and testbench parts of the design can be simulated in one simple and unified simulation environment without the need for clumsy co-simulations of disparate modeling paradigms.

2.3.1 Structure and Hierarchy

Modules

Structural decomposition is one of the fundamental hardware modeling concepts because it helps partition a complex design into smaller entities. In SystemC, structural decomposition is specified with modules, which are the basic building blocks. A SystemC description consists of a set of connected modules, each encapsulating some behavior or functionality. Modules can be hierarchical, containing instances of other modules. The nesting of hierarchy can be arbitrarily deep, which is an important requirement for structural design representation.

Signals and Ports

The simplest means of connecting together different SystemC modules is by using ports and signals. Actually, the interface of modules to the external world can be much more general and sophisticated, but the interface at the lowest and most primitive levels matches the typical facilities available in current HDLs. A port has an associated direction which can be input, output, or bidirectional.

A module is declared with the keyword *SC_MODULE*, and ports are specified with *sc_in*, *sc_out*, and *sc_inout* keywords, with the template parameter <bool> indicating that the type is Boolean (single bit). Other data types, including user defined ones, could also be used as port types. The structural hierarchy is specified inside the constructor for the module, specified with the keyword *SC_CTOR*.

2.3.2 Data Types

In addition to the standard C++ data types such as int, bool, char, etc., SystemC provides a rich set of data types which can be used to model hardware-specific concepts. We outline some useful data types here. The complete list of data types is given in [81]. Below are some illustrative examples:

- **4-state Logic**: In addition to the standard bit values '0' and '1', it is useful to provide a mechanism to indicate that the value of a bit is unknown. This helps identify initialization or conflict (multiple driver) problems during simulations. Further, there is the need to specify the high impedance (or tristate) state on signals. With this in mind, SystemC provides *sc_logic*, a four state logic data type, the states being '0' (low or false), '1' (high or true), 'X' (unknown), and 'Z' (high impedance or tristate). SystemC also provides data types to represent resolved logic signals which is useful in modeling wires and buses with multiple drivers.

- **Bit and Bit Vector**: The *sc_bit* and *sc_bv* types can be used to model bits and bit vectors for which only two states, '0' and '1' are sufficient, and on which logical operations such as logical AND, logical OR, etc are performed. Useful operations for these types include the reduction (*and* reduce, *or* reduce, and *xor* reduce) and part-select (range).

- **Fixed and Arbitrary Precision**: The integer data types provided in C++, such as int and unsigned have an implementation dependent bit width. However, the designer may wish to fix the precision of a data item if the range of values it takes is known in advance. SystemC provides two data type families for achieving this: fixed precision and arbitrary precision. The fixed precision types *sc_int* and *sc_uint* can be used to model data that is up to 64 bits wide.

These data types are implemented with a 64 bit integer. The usual operations associated with C++ integers can be applied to the fixed precision types, one useful addition being the bit-select operation.

- **Fixed Point Representation**: while the float data type can be used to model real numbers in the early stages of simulation, the hardware designer may have in mind an exact representation of such data in terms of the precision used for integral and fractional parts. The *sc_fixed* and *sc_ufixed* data types, which are used to represent such fixed point numbers in SystemC, are accompanied by the standard characteristics of fixed point arithmetic, such as quantization mode, overflow mode, and saturation bits.

The choice of data types has a significant impact on the simulation speed, and care must be taken to use the correct data types during modeling. The reader is referred to [81] for an exhaustive list of all the SystemC data types and the relevant operations.

2.3.3 Functionality and Concurrency

The functionality of a system is described in processes in SystemC. Analogous to VHDL processes, the SystemC processes are used to represent concurrent behavior – multiple processes within a module represent hardware or software blocks executing in parallel. Processes have an associated sensitivity list – a list of signals that trigger the execution of the process. There are two important types of processes.

Methods

A method process behaves like a function call and can be used to model simple combinational behavior. It does not have its own thread of execution, and hence, cannot be suspended. This characteristic allows for high simulation efficiency.

Threads

A thread process can be used to model sequential behavior. It is associated with its own thread of execution, and can be suspended and re-activated.

2.3.4 Time and Clocks

Since the concepts of time and clocks are very important in modeling hardware, SystemC provides a mechanism to specify them. A clock with a period of 10ns can be specified as:

```
sc_clock clk ("clk", 10, SC_NS);
```

The *sensitive*, *sensitive pos*, and *sensitive neg* keywords can be used to specify synchronization of a process to a clock.

```
SC_THREAD (x);
sensitive_pos << clk;
```

ensures that process x is activated on the positive edge of clock signal *clk*.

2.3.5 SystemC Update

The method *read* is trivial, and method *write* was faithfully described in [81]. The update method as described in SystemC documentation ensures deterministic behavior in the case of simultaneous read and write actions. If the new value of the signal is equal to the current value, then no update is needed. After that, we add it to the pending events set, set its time to next SystemC delta cycle, and finally change its type to event type.

2.3.6 SystemC MUTEX

This channel is part of SystemC 2.0 only [80]. It performs FIFO queuing of pending requests and issues a warning if multiple requests are issued during the same delta cycle. The MUTEX (mutual exclusion) channel is owned by only one process during any delta cycle at simulation time. If the channel is not locked, it is given to the first process that issues a request. Only the process that locked the MUTEX is allowed to unlock it. Dynamic sensitivity is used to suspend processes that request locking the channel when it is already locked, and and later resume them. The SystemC MUTEX primitive channel can be used to model shared variables, either through inheritance by deriving a shared variable channel from the MUTEX channel or by convention, where access to a certain variable is protected by a MUTEX.

The *lock* method keeps trying to take ownership of the channel while it is in use by another, the process will wait on freeing MUTEX channel event, and then check if the target channel is still in use. When it is freed, the process takes ownership of the channel, and it can unlock it later. The method, *trylock* tries one time to take ownership of the channel, it either fails or succeeds. The *unlock* method frees the channel (if process is the current owner of the channel) and triggers other processes that are suspended on freeing channel event in the next delta cycle.

2.3.7 Request Grant Protocol

This protocol deals with two SystemC channels, *master* and *slave*, that are sharing one port. Only one master and one slave can be connected to the port at one time. During a *WriteMaster* operation, the method verifies that the channel is not already requested, otherwise, it waits on the norequest event.

2.3.8 Design Rules

Each SystemC channel requires a specific number of ports to be connected to it, or arbitrarily unlimited in some cases. When a channel is created, it has to pass a design rules check in order to make sure that the number of ports connected is allowed. This is called static design rules checking. On the other hand, a channel may impose that only one process can perform an I/O operation at one time, so the channel has to pass design rules checking when processes access the channel. This is called dynamic design rules checking.

The SystemC signal channel demonstrates how to do dynamic design rule checking in addition to static design rule checking. During static design rule checking, the channel makes sure that only one writer port is attached.

2.3.9 Language Design

The overall architecture of the SystemC class library is summarized in Figure 2.2. The simulation kernel, i.e., the lightweight scheduler that is responsible for activating and suspending the SystemC processes is at the heart of the implementation, and forms the base layer. With these layers as the foundation, the communication elements – interfaces, channels, and ports are defined in the next layer. The design is based on the interface-method-call (IMC) scheme: essentially, the ports access the channels only through the interfaces. The example primitive channels supplied by SystemC is built on this layer. Finally, the hierarchical and other user-defined channels are built on the top layer. The most important observation is that the upper layers are built cleanly on the lower ones, and the designer can use the modeling mechanisms at any of the levels.

The simulation kernel for SystemC follows the evaluate-update paradigm that is common in HDLs. The concept of delta cycles, where multiple evaluate-update phases can occur at the same simulation time, is supported. A simplified version of the simulation algorithm is as follows:

Layer 4	Methodology-specific and User-defined Channels
Layer 3	Primitive Channels (signals, FIFOs, etc.)
Layer 2	Channels, Interfaces, Ports
Layer 1	Events, Dynamic Sensitivity
Layer 0	SystemC Scheduler

Figure 2.2: SystemC Language Architecture [84].

1. Initialization: Execute all processes to initialize the system.

2. Evaluate: Execute a process that is ready to run. Iterate until all ready processes are executed. Events occurring during the execution could add new processes to the ready list.

3. Update: Execute any update calls made during step

4. If delayed notifications are pending, determine list of ready processes and proceed to Evaluate phase (step 2).

5. Advance the simulation time to the earliest pending timed notification. If no such event exists, simulation is finished, else determine ready processes and proceed to step 2.

2.4 Transaction Level Modeling

Complexity management, particularly at the highest level of design, has led to the emergence of Transaction Level Modeling (TLM) [13]. The primary goal of TLM is to dramatically increase simulation speeds, while offering enough accuracy for the design task at hand. The increase in speed is achieved by the TLM abstracting away the number of events and amount of information that have to be processed during simulation to the minimum required. For example, instead of driving the individual signals of a bus protocol, the goal is to exchange only what is really necessary, i.e., the data payload. TLM also reduces the amount of detail the designer must handle, therefore making modeling easier.

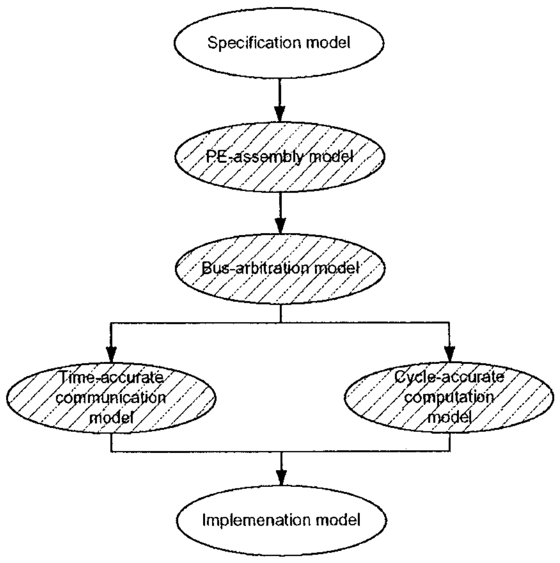


Figure 2.3: Defined System Models at Different Abstraction Levels [13].

Several proposals have been introduced to define TLM (in particular, [38], [49] and [85]). Among them, the model defined in [13] is the most accurate and complete. For instance, Cai *et al.* [13] proposed six system models at different abstraction levels according to Figure 2.3, where the TLM computation models are represented as shaded ellipses. These models are defined according to the following:

- *PE-assembly model*: The entities at the top level of the model represents concurrently executing processing elements(PEs), which communicates through message passing channels. The communication part of the model(channel) is untimed, while computation part of the model(PE) is timed through estimation. In comparison to specification model, PE-assembly model explicitly specifies the allocated PE in the system architecture and process-PE mapping decision.

- *Bus-arbitration model*: In comparison to PE-assembly model, channels between PEs in bus arbitration model represent buses, which are called abstract bus channels. The channels still implement data transfer through message passing, while bus protocols can be simplified as blocking and nonblocking I/O. No cycle-accurate and pin-accurate protocol details are specified. The abstract bus channels have estimated approximate time, which is specified in the channels by one wait statement per transaction.

- *Time-accurate communication model*: This model contains time/cycle accurate communication and approximate timed computation. (Rather than specifying the communication time, time-accurate communication model specifies the time constraint of communication, which is determined by the time diagram of components protocol.
- *Cycle-accurate computation model*: This model contains contains cycle accurate computation and approximate-timed communication generated from the *Bus-arbitration model.* computation components(PEs) are pin accurate and execute cycle-accurately.

Chapter 3

Soundness of the SystemC to AsmL Transformation

3.1 Introduction

In this Chapter, we establish the correctness of the transformation from SystemC to AsmL and vice-versa. Such a result represents a formal link between the two verification approaches we proposed for SoC verification framework previously introduced in Figure 1.2.

The work of Patrick and Radhia Cousot in [27] is the essence for any program transformation using abstract interpretation. The tactical choice of using semantics to link the subject program to the transformed program is very smart in the sense that it enables proving the soundness proof of the transformation, related to an observational semantics. The transformation from SystemC to AsmL, and vice-versa, represents an online program transformation which corresponds to the approach described in Section 3.9 of [27]. Figure 3.1 displays a projection of that generic methodology on a SystemC subject program and an AsmL transformed program. The same figure can be used to perform the soundness of a transformation and also to construct it. In both cases, we need to define the syntax, semantics and observation functions for both AsmL and SystemC.

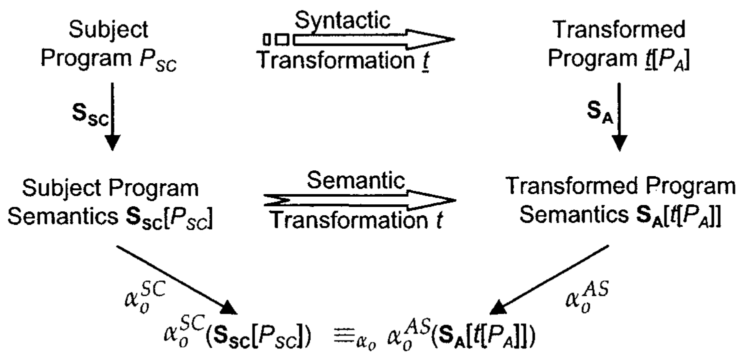

Subject Program P_{SC}
Syntactic Transformation t
Transformed Program $t[P_A]$
$\mathbf{S}_{SC}$
$\mathbf{S}_A$
Subject Program Semantics $\mathbf{S}_{SC}[P_{SC}]$
Semantic Transformation t
Transformed Program Semantics $\mathbf{S}_A[t[P_A]]$
α_o^{SC}
α_o^{AS}
$\alpha_o^{SC}(\mathbf{S}_{SC}[P_{SC}]) \equiv_{\alpha_o} \alpha_o^{AS}(\mathbf{S}_A[t[P_A]])$

Figure 3.1: Online Program Transformation.

Several approaches have been used to write the SystemC semantics. In particular, in [75], ASM based SystemC semantics have been defined. However, the use of ASM as a concrete semantics has two main drawbacks. First, the ASM notation has the tendency to hide low-level details, by making wide use of macros. While this may be an advantage to the casual reader, it is a drawback for the design of precise yet sound static analyses. Second, the program computation is hidden in the ASM transition relation, and the fixpoint computation is not explicitly stated. As a consequence, this formalism is inadequate to express, for example, program-wide invariant properties.

In this respect, denotational semantics [77] is found to be most effective since objects can be expressed as fixpoints on suitable domains [63]. Moreover, it is straightforward to consider a domain composed by an environment (a map from variables to addresses) and a store (a map from addresses to values). Hence, object aliasing can be naturally expressed. It was shown in [20] that generally denotational semantics is an abstraction of a trace-based operational semantics in the sense that it abstracts away the history of computations, by considering input-output functions. Salem in [90] proposed a denotational semantics for SystemC. However, the proposal in [90] was very shallow and does not relate the semantics of the whole SystemC program to the semantics of its classes. In [71] Logozzo presented a complete and sound denotational semantics for a generic OO language. However, this work cannot be directly applied to SystemC due to the specific environment, store, modules structure and simulation semantics of SystemC.

In this chapter, we provide a formalization of the SystemC and AsmL semantics in fixpoint. We will first list the syntactical domains. Then, we will provide the semantics of whole program and the semantics of modules for SystemC and classes for AsmL. Finally, we will establish the proofs for the completeness and soundness of the whole semantics. We expend the generic OO semantics provided by Logozzo in [71] by: (1) modifying the syntactical domains to support SystemC specific domains such as modules and signals; (2) upgrading the default environment and store to include the values of the signals in the previous, current and next simulation cycles; (3) adding to the generic class semantics (in particular to the class constructor) the information related to initializing the processes and threads involved in the simulation; and (4) re-establishing the soundness and completeness proofs considering the extensions and modifications of (1), (2) and (3).

3.2 SystemC Denotational Semantics

3.2.1 Syntactical Domains

The SystemC language has a large number of syntactical domains. These, however, are based on the single `SC_Module` domain. Hence, the minimum representation for a general SystemC program is as set of modules.

Definition 3.2.1. (SystemC Module: `SC_Module`)
A SystemC Module is a set ⟨`DMem`, `Ports`, `Chan`, `Mth`, `SC_Ctr`⟩, where `DMem` is a set of the module data members, `Ports` is a set of ports, `Chan` a set of SystemC Chan, `Mth` is a set of methods (functions) definition and `SC_Ctr` the module constructor.

Definition 3.2.2. (SystemC Port: `SC_Port`)
A SystemC port is a set ⟨`IF`, `N`, `SC_In`, `SC_Out`, `SC_InOut`⟩, where `IF` is a set of the virtual methods declarations, `N` is the number of interfaces that may be connected to the port, `SC_In` is an input port (provides only a `Read` method), `SC_Out` is an output port (provides only a `Write` method) and `SC_InOut` is an input/output port (provides `Read` and `Write` functions).

Definition 3.2.3. (SystemC Channel: `SC_Chan`)
A SystemC channel is a set ⟨`SigMeth`, `CurrVal`, `PrevVal`, `NewVal`, `SC_Mutex`, `SC_Semaph`⟩, where `SigMeth` is a set of basic channel virtual methods (including in particular the `Update` method), `CurrVal` the current value of the signal, `PrevVal` its previous value, `NewVal` its value in the next simulation cycle, `Mutex` is a mutex channel (including additional methods such as `Lock` and `UnLock`) and `SC_Semaph` is a semaphore interface (including in particular the number of concurrent accesses to the interface).

In contrast to default class constructors for OO languages, the SystemC module constructor `SC_Ctr` contains the information about the processes and threads that will be executed during simulation, and their sensitivity lists, SC_SL, specifying which events can affect their states.

Definition 3.2.4. (SystemC Constructor: `SC_Ctr`)
A SystemC constructor is a set ⟨`Name`, `Init`, `SC_Pr`, `SC_SSt`⟩, where `Name` is a string specifying the module name, `Init` is a default class constructor, `SC_Pr` a set of processes and `SC_SSt` is a set of sensitivity statements (to set the process sensitivity list `SC_SL`).

Definition 3.2.5. (SystemC Process: `SC_Pr`)
A SystemC process is a set ⟨`PMth`, `PTh`, `PCTh`⟩, where `PMth` is a method process (defined as a set ⟨`Mth`, `SC_SL`⟩ including the method and its sensitivity list), `PTh` is a thread process (accepts a wait statement in comparison to the method process), `PTh` is a clocked thread process (sensitive to the clock event).

Definition 3.2.6. (SystemC Process Sensitivity List: SC_SL)

A SystemC sensitivity list is a set $\langle \mathtt{SL_S}, \mathtt{SL_D} \rangle$, where $\mathtt{SL_S}$ is a static sensitivity list and $\mathtt{SL_D}$ is a dynamic list. Both lists contain a set of events `SC_Event` but are different in the sense that one can be updated during the simulation while the other is not changeable.

Definition 3.2.7. (SystemC Event: `SC_Event`)

A SystemC event is a set $\langle$`t`, `notify`, `cancel`$\rangle$, where `t` specifies (in simulation cycles) when the notification is supposed to be sent, `notify` the method used to notify the owner module and `cancel` is the method used to cancel an event.

Definition 3.2.8. (SystemC Program: SC_Pg)

A SystemC program is a set $\langle \mathtt{L_{SC\_Mod}}, \mathtt{SC\_main} \rangle$, where $\mathtt{L_{SC\_Mod}}$ is a set of SystemC modules and `SC_main` is the main function in the program that performs the simulator initialization and contains the modules declarations.

Note that restricting our model to modules does not affect the validity of the results since modules are the default syntactical domain for SystemC. All other domains are built on top of it.

3.2.2 Fixpoint Semantics

Semantic Domains

In this section, we define the semantics of the whole SystemC program, $\mathbb{W}\ [\![\mathtt{SC\_Pg}]\!]$, and the SystemC module, $\mathbb{M}_{SC}[\![\mathtt{m\_sc}]\!]$. Then, present the proofs (or proof sketches) of the soundness and completeness of $\mathbb{M}_{SC}[\![\mathtt{m\_sc}]\!]$.

Definition 3.2.9. (Delta Delay: δ_d)

The SystemC simulator considers two phases *evaluate* and *update*. The separation between these two phases is called *delta delay*.

Definition 3.2.10. (SystemC Environment: SC_Env)

The SystemC environment is the summation of the default C++ environment (Env) as defined in [71] and the signal environment (Sig_Store) specific to SystemC: SC_Env = Env + Sig_Env = [Var $\rightarrow$ Addr]+ [SC_Sig $\rightarrow$ Addr,Addr], where Var is a set of variables, SC_Sig is a set of SystemC signals and Addr $\subseteq \mathbb{N}$ is a set of addresses.

Definition 3.2.11. (SystemC Store: SC_Store)

The SystemC store is the summation of the default C++ store (Store) as defined in [71] and the signal store (Sig_Store): SC_Store = Store + Sig_Store = [Addr $\rightarrow$ Val]+ [(Addr, Addr) $\rightarrow$ (Val,Val)], where Val is a set of values such that SC_Env $\subseteq$ Val.

We denote by $R_0 \in \mathcal{P}(\texttt{SC\_Env}\times\texttt{SC\_Store})$ the set of initial states, $\texttt{pc}_{\texttt{in}}$ the entry point of the main function sc_main and $\rightarrow\subseteq$: (SC_Env $\times$ SC_Store) $\times$ (SC_Env$\times$SC_Store) a transition relation.

Whole SystemC Program Semantics

The whole SystemC program semantics can be defined as the traces of the executions of the program starting from a set of initial states R_0. It can be expressed in fixpoint semantics as follows:

Definition 3.2.12. (Whole Program Semantics: $\mathbb{W}$ [[SC_Pg]])

Let SC_Pg = $\langle L_{\texttt{SC\_Mod}}, \texttt{SC\_main}\rangle$ be a SystemC program. Then, the semantics of SC_Pg, $\mathbb{W}$ [[SC_Pg]]$\in \mathcal{P}(\texttt{SC\_Env}\times\texttt{SC\_Store}) \rightarrow \mathcal{P}(\mathcal{T}(\texttt{SC\_Env}\times \texttt{SC\_Store}))$ is:

$$\mathbb{W}[\![\texttt{SC\_Pg}]\!](R_0) =$$

$$\text{lfp}\,{}^{\subseteq}_{\emptyset}\lambda X.\quad (R_0) \cup \{\rho_0 \rightarrow \ \dots\ \rho_n \rightarrow \rho_{n+1} |\ \rho_{n+1} \in (\texttt{SC\_Env}\times \texttt{SC\_Store}) \wedge \{\rho_0 \rightarrow \ \dots\ \rho_n\} \in X \wedge \rho_n \rightarrow \rho_{n+1}\}$$

Module Semantics

A SystemC module is a particular C++ class where the constructor declares a set of processes and thread that will executed during simulation according to a set of events (timed or non-timed). The module semantics can be defined as the set of all its instances. While and object module semantics reflects the evolution of the object internal state.

Module Constructor Semantics

Definition 3.2.13. (Process Declaration: $\mathbb{P}_R$ $[\![\texttt{SC\_Pr}]\!]$))
Let SC_Pr = ⟨PMth,PTh, PCTh⟩ be a SystemC process. Then, the semantics of SC_Pr, $\mathbb{P}_R$ $[\![\texttt{SC\_Pr}]\!]) \in \mathcal{P}(\texttt{SC\_Env}\times\texttt{SC\_Store}) \rightarrow \mathcal{P}(\mathcal{T}(\texttt{SC\_Env}\times\texttt{SC\_Store}))$ is

$$\mathbb{P}_R\ [\![\texttt{SC\_Pr}]\!](\mathrm{R}_0, M, SL) =$$
$$\mathrm{lfp}\,\overset{\subseteq}{\underset{\emptyset}{}}\ \lambda X,\ m,\ sl.\ (\mathrm{R}_0) \cup \{\rho_0 \rightarrow\ \ldots\ \rho_n \rightarrow \rho_{n+1} |\ \rho_{n+1} \in$$
$$(\texttt{SC\_Env}\times\ \texttt{SC\_Store}) \wedge \{\rho_0 \rightarrow\ \ldots\ \rho_n\} \in X$$
$$\wedge\ \rho_n \rightarrow \rho_{n+1} \wedge \rho_{n+1}(X) = (m, sl)\}$$

Definition 3.2.14. (Module Constructor: $\mathbb{P}_{Ctr}$ $[\![\texttt{SC\_Ctr}]\!]$))
Let SC_Ctr = ⟨Name, Init, SC_Pr, SC_SSt⟩ be a constructor of a SystemC module. Then, the semantics of SC_Ctr, $\mathbb{P}_{Ctr}[\![\texttt{SC\_Ctr}]\!] \in \mathcal{P}(\texttt{SC\_Env} \times \texttt{SC\_Store}) \rightarrow \mathcal{P}(\mathcal{T}(\texttt{SC\_Env} \times \texttt{SC\_Store}))$ is:

$$\mathbb{P}_{Ctr}[\![\texttt{SC\_Ctr}]\!](\mathrm{L}_{P,M,SL}) =$$
$$\mathrm{lfp}\,\overset{\subseteq}{\underset{\emptyset}{}}\ \lambda\ \{(p_1, m_1, sl_1), \ldots, (p_i, m_i, sl_i), \ldots, (p_n, m_n,$$
$$sl_n)\}.\ \cup_{(p_i,m_i,sl_i)\in L_{p,m,sl}}\ \mathbb{P}_R\ [\![\texttt{SC\_Pr}]\!](\mathrm{R}_0, M, SL)\}$$

Module Object Semantics

In a general OO context, such as C++, an object can be defined as a set of states including a first (initial) state representing the object just after its creation and a set of states resulting from the interaction of the object with its context [71]. In this case, the interaction can happen in two ways: (1) the context invokes an object's method, or (2) the context modifies a memory location reachable from the object's environment. In [71], this interaction was very well defined using two functions $\texttt{next}_{\texttt{d}}$, for direct interactions, and $\texttt{next}_{\texttt{ind}}$ for indirect interactions. The object semantics, $\mathbb{O}$ $[\![\text{o}]\!]$, was defined as [71]:

$$\mathbb{O}[\![\text{o}]\!](\texttt{v},\texttt{s}) = \text{lfp}_{\emptyset}^{\subseteq} \lambda T. \\ \text{S}_0\langle v,s\rangle \cup \{\sigma_0 \xrightarrow{l_0} \dots \xrightarrow{l_{n-1}} \sigma_n \xrightarrow{l_n} \sigma' | \\ \{\sigma_0 \xrightarrow{l_0} \dots \xrightarrow{l_{n-1}} \sigma_n \in T,\ \texttt{next}(\sigma_n) \ni \langle \sigma', l'\rangle\}$$

where $\text{next}(\sigma) = \texttt{next}_{\texttt{d}}(\sigma) \cup \texttt{next}_{\texttt{ind}}(\sigma)$, l is a transition label and $\text{S}_0\langle v,s\rangle$ is a set of initial states.

In addition to the semantics definition of an OO object in [71], a SystemC method can be activated by the SystemC simulator through the sensitivity list of the process. This interaction is a hybrid direct/indirect interaction because the SystemC simulator will, according to the state of the program events (that may be external to the module), invoke directly the concerned methods. First, we will define the interaction states, then, we will provide the complete definition for the direct, indirect and SystemC simulator based interaction functions.

Definition 3.2.15. (Interaction States)
The set of interaction states is $\sum = \texttt{SC\_Env} \times \texttt{SC\_Store} \times \texttt{D}_{\texttt{out}} \times \mathcal{P}(\texttt{Addr})$

After the creation of the module object, the reached states represent the initial states defined as follows:

Definition 3.2.16. (Initial States $\text{S}_0\langle v_{sc}, s_{sc}\rangle$)
Let $v_{sc} \in \texttt{D}_{\texttt{in}}$ be a SystemC object input value, $s_{sc} \in \texttt{SC\_Store}$ a store at object creation time and SC_Obj a SystemC module object. The set of initial states of SC_Obj is:

$$\text{S}_0\langle v_{sc}, s_{sc}\rangle = \{\langle e'_{sc}, s'_{sc}, \phi, \emptyset\rangle \mid \mathbb{P}_{Ctr}[\![\texttt{SC\_Ctr}]\!](\text{L}_{P,M,SL}) \\ \ni \langle e'_{sc}, s'_{sc}\rangle\}$$

where: $\text{L}_{P,M,SL} = \{(p_1, m_1, sl_1), \dots, (p_i, m_i, sl_i), \dots,$
$(p_n, m_n, sl_n)\}$ is a list of all the module processes, methods, and sensitivity lists.

In Definition 3.2.16, ϕ is a void value ($\in \mathtt{D_{out}}$) meaning that the constructor does not return any value and therefore does not expose any address to the context.

Definition 3.2.17. (Transition Labels: Label_SC)
The set of transition labels is $\mathtt{Label\_SC} = (\mathtt{Mth} \times \mathtt{D_{in}}) \cup (\mathtt{SC\_Pr} \times \mathtt{D_{in}}) \cup \{k\}$.

In Definition 3.2.17, we distinguish three types of interactions corresponding, respectively, to: (1) invoking a C++ method (direct interaction); (2) invoking a SystemC process (interaction through the SystemC simulator); and (3) modifying the memory location that is reachable from the the object environment (indirect interaction). The transition function next_SC is made up of three functions: $\mathtt{next\_SC_{dir}}$, $\mathtt{next\_SC_{pr}}$ and $\mathtt{next\_SC_{ind}}$.

Definition 3.2.18. (Direct Interactions: $\mathtt{next\_SC_{dir}}$)
Let $\langle e_{sc}, s_{sc}, v_{sc}, \mathtt{Esc}\rangle \in \sum$ be an interaction state. Then, the direct interaction function $\mathtt{next\_SC_{dir}} \in [\sum \to \mathcal{P}(\sum \times \mathtt{Label\_SC})]$ is defined as:

$$\begin{aligned}
&\mathtt{next\_SC_{dir}}(\langle e_{sc}, s_{sc}, v_{sc}, \mathtt{Esc}\rangle) = \\
&\quad \{(\langle e'_{sc}, s'_{sc}, v'_{sc}, \mathtt{Esc'}\rangle, \langle mth, v_{in}\rangle) \mid mth \in \mathtt{Mth},\ v_{in} \in \mathtt{Din}, \\
&\quad \mathbb{M}[\![\mathtt{mth}]\!](v_{in}, e_{sc}, s_{sc}) \ni (v'_{sc}, e'_{sc}, s'_{sc}), \\
&\quad \mathtt{Esc'} = \mathtt{Esc} \cup \mathit{reachable}(v'_{sc}, e'_{sc})\}.
\end{aligned}$$

where $\mathbb{M}[\![\mathtt{mth}]\!]$ is the semantics of generic OO method as defined in [71].

The function *reachable* is an extension of the *helper function* defined in [71]. For instance, given an address v_{sc} and a store s_{sc}, *reachable* determines all the addresses that are reachable from v_{sc}. In the SystemC context, this function acts only on the data members of the module according to the following recursive definition:

Definition 3.2.19. (The Function *reachable*)
The function *reachable* $\in [\mathtt{D_{out}} \times \mathtt{SC\_Store}] \to \mathcal{P}(\mathtt{Addr})$ is defined as follows:
$\mathit{reachable}(v_{sc}, s_{sc}) =$

if $v_{sc} \in \texttt{Addr}$ then

$\{\texttt{Addr}\} \cup \{\mathit{reachable}(e'_{sc}(d_{mem}),s'_{sc}) \mid$

$\exists\ \texttt{sc\_module} = \langle\texttt{DMem, Ports, Chan, Mth, SC\_Ctr}\rangle,$

$d_{mem} \in \texttt{DMem},\ s_{sc}(v_{sc})$ is an instance of $\texttt{sc\_module},$

$s_{sc}(s_{sc}(v_{sc})) = e'_{sc}\}$

else $\emptyset$.

In the case of interactions related to changing the sensitivity list of a processor, the function $\texttt{next\_SC}_{\texttt{pr}}$ considers the method that was affected to the process in the module constructor. Then, the invocation of the method is similar to the direct interaction.

Definition 3.2.20. (Process Interactions: $\texttt{next\_SC}_{\texttt{pr}}$)
Let $\langle e_{sc}, s_{sc}, v_{sc}, \texttt{Esc}\rangle \in \sum$ be an interaction state. Then, the process interaction function $\texttt{next\_SC}_{\texttt{pr}} \in [\sum \to \mathcal{P}(\sum \times$
$\texttt{Label\_SC})]$ is defined as:

$\texttt{next\_SC}_{\texttt{pr}}(\langle e_{sc}, s_{sc}, v_{sc}, \texttt{Esc}\rangle) =$

$\{\langle\langle e''_{sc}, s''_{sc}, v''_{sc}, \texttt{Esc}''\rangle, \langle pr, m, sl, v_{in}\rangle\rangle \mid pr \in \texttt{SC\_Pr},$

$v_{in} \in \texttt{Din},\ \mathbb{P}_{Ctr}[\![\texttt{SC\_Ctr}]\!](pr, m, sl) \ni (v'_{sc}, e'_{sc}, s'_{sc}),$

$\mathbb{M}[\![\texttt{m}]\!](v_{in}, e'_{sc}, s'_{sc}) \ni (v''_{sc}, e''_{sc}, s''_{sc}),$

$\texttt{Esc}'' = \texttt{Esc} \cup \mathit{reachable}(v''_{sc}, e''_{sc})\}.$

The third possible interaction corresponds to indirect interaction which may happen when an address escapes from an object. In that case, the context can modify the content of this address with any value. The function $\texttt{next\_SC}_{\texttt{ind}}$ defines this type of interaction:

Definition 3.2.21. (Indirect Interactions: $\texttt{next\_SC}_{\texttt{ind}}$)
Let $\langle e_{sc}, s_{sc}, v_{sc}, \texttt{Esc}\rangle \in \sum$be an interaction state. Then, the indirect interaction function $\texttt{next\_SC}_{\texttt{ind}} \in [\sum \to \mathcal{P}(\sum \times$
$\texttt{Label\_SC})]$ is defined as:

$$\begin{aligned}&\texttt{next\_SC}_{\texttt{ind}}(\langle e_{sc}, s_{sc}, v_{sc}, \texttt{Esc}\rangle) = \\ &\quad \{\langle\langle e_{sc}, s'_{sc}, \phi, \texttt{Esc}\rangle, k\rangle \mid \exists\ \alpha \in \texttt{Esc}. \\ &\quad s'_{sc} \in \mathit{update\_sc}(\alpha, s'_{sc})\}.\end{aligned}$$

The *update_sc* function is an extension of the *update* function defined in [71] in the sense that it considers SystemC signals in addition to C++ variables. It is defined in following:

Definition 3.2.22. (The Function *update_sc*)
The function *update_sc* $\in$ [Addr $\times$ SC_Store $\rightarrow \mathcal{P}$(SC_Store)] is defined as follows:

$$\mathit{update\_sc}(\alpha, s_{sc}) = \{s'_{sc} \mid \exists\ v \in \texttt{Val}.\ s'_{sc} = s_{sc}[\alpha \mapsto v]\ \}.$$

update_sc returns all the possible stores where $s_{sc}(\alpha)$ takes all the possible values in values domain Val.

Using the definitions of $\texttt{next\_SC}_{\texttt{dir}}$, $\texttt{next\_SC}_{\texttt{pr}}$ and $\texttt{next\_SC}_{\texttt{ind}}$, we define the global transition function next_SC as:

Definition 3.2.23. (Transition Function: next_SC)
Let st $= \langle e_{sc}, s_{sc}, v_{sc}, \texttt{Esc}\rangle \in \sum$ be an interaction state. Then, the transition function next_SC $\in [\sum \rightarrow \mathcal{P}(\sum \times$
Label_SC)] is defined as:

$$\begin{aligned}\texttt{next\_SC}(\texttt{st}) = &\ \texttt{next\_SC}_{\texttt{dir}}(\texttt{st}) \cup \texttt{next\_SC}_{\texttt{pr}}(\texttt{st}) \\ &\cup \texttt{next\_SC}_{\texttt{ind}}(\texttt{st})\end{aligned}$$

Using the transition function, a SystemC module object's semantics is defined as follows:

Definition 3.2.24. (SystemC Module Object: $\mathbb{O}_{SC}[\![\text{o\_sc}]\!]$)
Let $v_{sc} \in$ Val be a SystemC object input value and $s_{sc} \in$ SC_Store a store at object creation time. Then the SystemC object semantics, $\mathbb{O}_{SC}[\![\text{o\_sc}]\!] \in [\text{D}_{\text{in}} \times \text{Store}] \rightarrow \mathcal{P}(\mathcal{T}(\Sigma))$ is defined as:

$\mathbb{O}_{SC}[\![\texttt{o\_sc}]\!])(v_{sc}, s_{sc}) = \text{lfp}\,{}^{\subseteq}_{\emptyset}\ \lambda T.$
$\quad \texttt{S}_0\langle v, s\rangle \cup \{\sigma_0 \xrightarrow{l_0} \ldots \xrightarrow{l_{n-1}} \sigma_n \xrightarrow{l_n} \sigma' |$
$\quad \{\sigma_0 \xrightarrow{l_0} \ldots \xrightarrow{l_{n-1}} \sigma_n \in T,\ \texttt{next\_SC}(\sigma_n) \ni \langle \sigma', l'\rangle\}\}$

where $\sum = \texttt{SC\_Env} \times \texttt{SC\_Store} \times \texttt{D}_{\texttt{out}} \times \mathcal{P}(\texttt{Addr})$ is a set of interaction states, $\texttt{D}_{\texttt{in}}$ and $\texttt{D}_{\texttt{out}}$ are, respectively, the semantic domains for the input and output values.

Theorem 3.2.1 *(SystemC Module Object Semantics in Fixpoint)*
Let $F_{sc} = \lambda T.$
$\quad S_0\langle v, s\rangle \cup \{\sigma_0 \xrightarrow{l_0} \ldots \xrightarrow{l_{n-1}} \sigma_n \xrightarrow{l_n} \sigma' |$
$\quad \{\sigma_0 \xrightarrow{l_0} \ldots \xrightarrow{l_{n-1}} \sigma_n \in T,\ \textit{next\_SC}(\sigma_n) \ni \langle \sigma', l'\rangle\}$
Then $\mathbb{O}_{SC}[\![o\_sc]\!])(v_{sc}, s_{sc}) = \cup_{n=0}^{\omega} {F_{sc}}^{n}(\emptyset)$

Proof 3.2.1. The proof is immediate from the fixpoint theorem in [25].

Definition 3.2.25. (Module Semantics: $\mathbb{M}_{SC}[\![\texttt{m\_sc}]\!])$)
Let $\texttt{m\_sc} = \langle \texttt{DMem, Ports, Chan, Mth, SC\_Ctr}\rangle$ be a SystemC module. Then its semantics $\mathbb{M}_{SC}[\![\texttt{m\_sc}]\!]) \in \mathcal{P}(\mathcal{T}(\Sigma))$ is:
$\mathbb{M}_{SC}[\![\texttt{m\_sc}]\!] = \{\mathbb{O}_{SC}[\![\texttt{o\_sc}]\!](v_{sc}, s_{sc}) \mid \texttt{o\_sc}$ is an instance of $\texttt{m\_sc},\ \texttt{v\_sc} \in \texttt{D\_in},\ \texttt{s\_sc} \in \texttt{SC\_Store}\}$

Theorem 3.2.2 *(SystemC Module Semantics in Fixpoint)*
Let $G_{sc}\langle S\rangle = \lambda T.$
$\quad \{S_0\langle v, s\rangle \mid \langle v, s\rangle \in S\ \} \cup \{\sigma_0 \xrightarrow{l_0} \ldots \xrightarrow{l_{n-1}} \sigma_n \xrightarrow{l'} \sigma' |$
$\quad \{\sigma_0 \xrightarrow{l_0} \ldots \xrightarrow{l_{n-1}} \sigma_n \in T,\ \textit{next}_{sc}(\sigma_n) \ni \langle \sigma', l'\rangle\}$
Then $\mathbb{M}_{SC}[\![m\_sc]\!](v_{sc}, s_{sc}) = \textit{lfp}\,{}^{\subseteq}_{\emptyset}\ G_{sc}\langle\ D_{in} \times \textit{Store}\rangle$

Proof 3.2.2. Although the SystemC model presents some additional functionalities on top of C++, the proof of this theorem is similar to the proof of Theorem 3.2 in [71]. For instance, considering the definition of $\mathbb{M}_{SC}$ and applying in order Definition of a SystemC module object, Theorem 3.2.1 and the fixpoint theorem in [25], the proof is straightforward.

Soundness and Completeness of the Module Semantics

The last step in the SystemC fixpoint semantics is to relate the module semantics to the whole SystemC program semantics. For this purpose, we consider an extended version of the functions *split* ($\alpha^{\circ}_{\times}$), *project* ($\alpha^{\circ}_{\uparrow}$) and *abstract* (α°) as defined in [71]. The new functions are upgraded to support the SystemC simulation semantics, environment and store. For example, *split_SC* (α_SC$^{\circ}_{\times}$) can drop the memory reached by the environment for a method that was previously executed in the current simulation cycle because a method cannot be executed again until the next cycle starts.

The basic concept behind defining the module object semantics is to cut all the instances not involving the object. For this purpose, two helper functions are required: (1) α_SC$^{\circ}_{\times}$ cuts all the traces involving the object instances; and (2) α_SC$^{\circ}_{\uparrow}$ maps all the cut instances to interaction states.

First we define the helper function `split_SC` that given a trace τ and an object `o_sc` returns a pair consisting of the last state of the prefix of τ made up of the last state of the execution of a method or process of `o_sc` and the remaining suffix of the prefix of τ. In contrast to the general case of OO programs, we consider both default C++ methods and processes according to the following definition:

Definition 3.2.26. (The Split Helper Function `split_SC`)
Let `o_sc` be a SystemC module object, $\tau \in \mathcal{T}(\texttt{SC\_Env} \times \texttt{SC\_Store})$, `CurProcess` $\in$ `SC_Pr`, `CurMethod` $\in$ `Mth` and $\texttt{pc}_{exit}$ be the exit point of $\tau(0)(\texttt{CurMethod})$. Then `split_SC` $\in [(\mathcal{T}(\texttt{SC\_Env} \times \texttt{SC\_Store}) \rightarrow (\texttt{SC\_Env} \times \texttt{SC\_Store}) \times \mathcal{T}(\texttt{SC\_Env} \times \texttt{SC\_Store})]$ is defined as:

$\texttt{split\_SC}(\tau)=$

$\quad$ let $n = \min\{i \in \mathbb{N} \mid \tau(i)(\texttt{CurProcess})=\langle\texttt{CurMethod}, \texttt{SL}\rangle$

$\quad \wedge\ \tau(i)(\texttt{SL})=\texttt{true} \wedge \tau(i)(\texttt{pc})=\texttt{pc}_{exit} \wedge$

$\quad \tau(i)(\texttt{this})= \texttt{o\_sc} \wedge \tau(i)(\texttt{StackHeight})=$

$\quad \tau(0)(\texttt{StackHeight})\}$

$\quad$ in $\langle\tau(n), \tau(n+1) \rightarrow \ldots \rightarrow \tau(\texttt{Len}(\tau)-1)\rangle$

The cut function $\alpha\_\mathrm{SC}^{\circ}_{\times}$ considers four different cases: (1) for the empty trace, ϵ, it returns an empty trace; (2) if the trace is part of the object trace, then we split it recursively keeping only the last state of the execution of a method or process. The rest of the trace is removed; (3) If this is not the current object and the store is not changed, then we continue with the rest of the trace; and (4) If this is not the current object and the store is changed, then we keep the current trace and we continue with the rest of the traces.

Definition 3.2.27. (Cut Function: $\alpha\_\mathrm{SC}^{\circ}_{\times}$)
Let o_sc be a SystemC module object, $\tau \in \mathcal{T}(\texttt{SC\_Env}\times\ \texttt{SC\_Store})$. Then $\alpha\_\mathrm{SC}^{\circ}_{\times} \in [(\mathcal{T}(\texttt{SC\_Env}\times\ \texttt{SC\_Store}) \times \texttt{SC\_Store}) \rightarrow \mathcal{T}(\texttt{SC\_Env} \times \texttt{SC\_Store})]$ is defined as:

$$\alpha\_\mathrm{SC}^{\circ}_{\times} = \lambda(\tau, \mathtt{S_{last}}).$$

$$\begin{cases} \epsilon & \texttt{if } \tau = \epsilon \\ \texttt{let } \langle \rho', \tau' \rangle = \texttt{split\_SC}(\tau) & \texttt{if } \tau = \langle e_{sc}, s_{sc} \rangle \rightarrow \tau'', \\ \texttt{in let } \langle e'_{sc}, s'_{sc} \rangle = \rho' & e_{sc}(\texttt{this}) = \texttt{o\_sc} \\ \texttt{in } \rho' \rightarrow \alpha\_\mathrm{SC}^{\circ}_{\times}(\tau', s'_{sc}) & \\ & \texttt{if } \tau = \langle e_{sc}, s_{sc} \rangle \rightarrow \tau'', \\ \alpha\_\mathrm{SC}^{\circ}_{\times}(\tau'', \mathtt{S_{last}}) & e_{sc}(\texttt{this}) \neq \texttt{o\_sc}, \\ & \mathtt{S}_{/\mathtt{S(o\_sc)}} = \mathtt{S}_{\mathtt{last}/\mathtt{S(o\_sc)}} \\ & \texttt{if } \tau = \langle e_{sc}, s_{sc} \rangle \rightarrow \tau'', \\ \langle e_{sc}, s_{sc} \rangle \rightarrow \alpha\_\mathrm{SC}^{\circ}_{\times}(\tau'', \mathtt{S}) & e_{sc}(\texttt{this}) \neq \texttt{o\_sc}, \\ & \mathtt{S}_{/\mathtt{S(o\_sc)}} \neq \mathtt{S}_{\mathtt{last}/\mathtt{S(o\_sc)}} \end{cases}$$

The second part of the abstraction includes the $\alpha\_\mathrm{SC}^{\circ}_{\uparrow}$ function which maps the states of a trace to interaction states.

Definition 3.2.28. (Map Function: $\alpha\_\mathrm{SC}^{\circ}_{\uparrow}$)
Let o_sc be a SystemC module object, $\tau \in \mathcal{T}(\texttt{SC\_Env} \times$
$\texttt{SC\_Store})$. Then $\alpha\_\mathrm{SC}^{\circ}_{\uparrow} \in [(\mathcal{T}(\texttt{SC\_Env}\times\ \texttt{SC\_Store}) \times \mathcal{P}(\texttt{Addr})) \rightarrow \mathcal{T}(\sum)]$ is defined as:

$\alpha\_\mathtt{SC}^{\circ}_{\uparrow} = \lambda(\tau, \mathtt{Esc}).$

$$\begin{cases} \epsilon & \text{if } \tau = \epsilon \\ \mathtt{let}\ \langle e_{sc}, s_{sc}\rangle = \rho & \\ \mathtt{in\ let}\ \mathtt{Esc}' = \mathtt{Esc}\cup & \\ \quad \mathtt{reachable\_SC}(\rho(\mathtt{retVal}), s_{sc}) & \text{if } \tau = \rho \rightarrow \tau', \\ \mathtt{in}\ \langle\langle e_{sc}, s_{sc}, \rho(\mathtt{retVal}), \mathtt{Esc}\rangle, & e_{sc}(\mathtt{this}) = \mathtt{o\_sc} \\ \langle \rho(\mathtt{curMethod}), \rho(\mathtt{inVal})\rangle\rangle & \\ \rightarrow \alpha\_\mathtt{SC}^{\circ}_{\uparrow}(\tau', \mathtt{Esc}') & \\ \mathtt{let}\ \langle e_{sc}, s_{sc}\rangle = \rho & \\ \mathtt{in}\ \langle\langle e_{sc}, s_{sc}, \phi, \mathtt{Esc}\rangle, k\rangle & \text{if } \tau = \rho \rightarrow \tau', \\ \rightarrow \alpha\_\mathtt{SC}^{\circ}_{\uparrow}(\tau', \mathtt{Esc}) & e_{sc}(\mathtt{this}) \neq \mathtt{o\_sc} \end{cases}$$

The abstraction function $\alpha\_\mathtt{SC}^{\circ}$ projects from the traces of an execution the set of relevant states to a specific object.

Definition 3.2.29. (Abstract Function: $\alpha\_\mathtt{SC}^{\circ}$)

Let o_sc be a SystemC module object, $\mathtt{T} \subseteq \mathcal{T}(\mathtt{SC\_Env} \times \mathtt{SC\_Store})$ a set of execution traces and $\mathtt{s}_{\emptyset}$ the empty store. The the abstraction function $\alpha\_\mathtt{SC}^{\circ} \in [(\mathcal{T}(\mathtt{SC\_Env} \times \mathtt{SC\_Store}) \rightarrow \mathcal{P}(\mathcal{T}(\sum))]$ is defined as:

$$\alpha\_\mathtt{SC}^{\circ}(\mathtt{T}) = \{\alpha\_\mathtt{SC}^{\circ}_{\uparrow}(\alpha\_\mathtt{SC}^{\circ}_{\times}\ (\tau,\mathtt{s}_{\emptyset}),\emptyset) \mid \tau \in \mathtt{T}\ \}$$

Theorem 3.2.3 *(Soundness of* $\mathbb{M}_{SC}[\![m\_sc]\!]$*)*

Let M_{SC} be a whole SystemC program and let $m_{SC} \in M_{SC}$ be a SystemC module. Then $\forall\ R_0 \in SC\_Env \times\ SC\_Store.\ \forall\ \tau \in \mathcal{T}(SC\_Env \times\ SC\_Store).$

$$\tau \in \mathbb{W}[\![SC\_Pg]\!](R_0) : \exists \tau' \in \mathbb{M}_{SC}[\![m_{SC}\ ]\!].\ \alpha\_SC^{\circ}(\{\tau\}) = \{\tau'\}$$

Proof 3.2.3. We have to consider both cases when τ contains an object $\mathtt{o}_{SC}$, instantiation of $\mathtt{m}_{SC}$, and when it does not include any $\mathtt{o}_{SC}$. For the second situation, the proof of the theorem is trivial considering that τ will be an empty trace. In the first case, the trace is not empty (let it be τ''). Since SystemC modules are initialized in the main program `sc_main` before the simulation starts, there exist an initial environment, store and set of variables that define the initial trace $\sigma_0 \in \tau''$. The rest of the traces in τ'' are interaction states of o_{SC} because they are obtained by applying α_SC° on τ. Therefore, $\tau'' \in \mathbb{M}_{SC}[\![\mathtt{m}_{SC}]\!]$.

Theorem 3.2.4 *(Completeness of* $\mathbb{M}_{SC}[\![]\!]$*)*
Let m_{SC} *be a SystemC module. Then*

$$\forall \tau \in \mathcal{T}(\Sigma).\ \tau \in \mathbb{M}_{SC}[\![m_{SC}]\!]:\ \exists\ SC\_P \in \langle L_{SC\_Pg}\rangle.$$
$$\exists \rho_0 \in SC\_Env \times SC\_Store.\ \exists\ o_{SC}\ \textit{instance of}\ m_{SC}.$$
$$\exists\ \tau' \in \mathcal{T}(SC\_Env \times SC\_Store).\ \tau' \in$$
$$\mathbb{W}[\![\rho_0]\!] \wedge \alpha\_SC^\circ(\{\tau'\}) = \{\tau\}$$

Proof 3.2.4. A SystemC program satisfying the previous theorem can be constructed by creating an instance of $\mathtt{m}_{SC}$ in the `sc_main` function, the initial state corresponds to the state when the module's constructor, `SC_Ctr`, was executed. An execution of a method of $\mathtt{m}_{SC}$ corresponds to executing a method thread (setting of the events in its sensitivity list to *Active*) and a change of a port corresponds to updating its internal signal by the new values. Hence, it is always possible to construct both `SC_P` and ρ_0. For instance, there exist many other possible constructions involving SystemC threads, clocked threads, etc.

3.3 AsmL Denotational Semantics

3.3.1 Syntactical Domains

We will present the basic syntactical domains that are required for the semantics section. These include: classes, methods, constraints and programs.

Definition 3.3.1. (AsmL Class: `AS_C`)
An AsmL class is a set ⟨`AS_DMem`, `AS_Mth`, `AS_Ctr`⟩, where `AS_DMem` is a set of the class

data members, `AS_Mth` a set of methods (functions) definition and `AS_Ctr` is the class constructor.

One of the important AsmL features corresponds to the methods pre-conditions (Boolean proposition verified before the execution of the method).

Definition 3.3.2. (AsmL Method: `AS_Mth`)
An AsmL method is a set ⟨`AS_M`, `AS_Pre`, `AS_Pos`, `AS_Cst`⟩, where `AS_M` is a the core of the method, `AS_Pre` is a set of pre-conditions, `AS_Pos` is a set of post–conditions and `AS_Cst` is a set of constraints.

Note that `AS_Pre`, `AS_Pos` and `AS_Cst` share the same structure. They are differentiated in the methods by using a specific keyword for each of them (e.g., *require* for pre-conditions).

Definition 3.3.3. (AsmL Method Precondition: `AS_Pre`)
An AsmL method pre-condition is a set ⟨`AS_B`⟩, where `AS_B` is a Boolean proposition.

Definition 3.3.4. (AsmL Program: `AS_Pg`)
An AsmL Program is a set $\langle \mathtt{L}_{\mathtt{AS\_C}}, \mathtt{INIT}\rangle$, where $\mathtt{L}_{\mathtt{AS\_C}}$ is a set of AsmL classes and `INIT` is the main function in the program.

3.3.2 Semantical Domains

AsmL considers two phases: *evaluate* and *update*. The program will be always running in the *evaluate* mode except if an update is requested. There are two types of updates, total and partial.

Definition 3.3.5. (Total Update: `Step`)
A total update is performed using the `Step` instruction and affects all the programs variables.

Definition 3.3.6. (AsmL Environment: $\texttt{AS\_Env}$)
The AsmL environment is a modified OO environment $\texttt{AS\_Env} = [\texttt{Var} \rightarrow \texttt{Addr,Addr}]$, where $\texttt{Var}$ is a set of variables and $\texttt{Addr} \subseteq \mathbb{N}$ is a set of addresses.

For every variable correspond two addresses storing its current and the new values.

Definition 3.3.7. (AsmL Store: $\texttt{AS\_Store}$)
The AsmL store is $\texttt{AS\_Store} = [(\texttt{Addr, Addr}) \rightarrow (\texttt{Val,Val})]$, where $\texttt{Val}$ is a set of values such that $\texttt{AS\_Env} \subseteq \texttt{Val}$.

Let $\texttt{R}_0 \in \mathcal{P}(\texttt{AS\_Env}\times\texttt{AS\_Store})$ be a set of initial states, $\texttt{pc}_{\texttt{in}}$ be the entry point of the main function $\texttt{Main}$ and $\rightarrow\subseteq$: $(\texttt{AS\_Env} \times \texttt{AS\_Store}) \times (\texttt{AS\_Env}\times \texttt{AS\_Store})$ be a transition relation.

3.3.3 Fixpoint Semantics

Whole AsmL Program Semantics

The whole AsmL program semantics can be defined as the traces of the executions of the program starting from a set of initial states $\texttt{R}_0$. It can be expressed in fixpoint semantics as follows:

Definition 3.3.8. (Whole AsmL Program Semantics: $\mathbb{W}_{AS}$ $[\![\texttt{AS\_Pg}]\!]$)
Let $\texttt{AS\_Pg} = \langle \texttt{L}_{\texttt{AS\_C}}, \texttt{Main}\rangle$ be an AsmL program. Then, the semantics of $\texttt{AS\_Pg}$, $\mathbb{W}_{AS}$ $[\![\texttt{AS\_Pg}\ ]\!] \in \mathcal{P}(\texttt{AS\_Env}\times\texttt{AS\_Store}) \rightarrow \mathcal{P}(\mathcal{T}(\texttt{AS\_Env}\times \texttt{AS\_Store}))$ is

$$\mathbb{W}_{AS}[\![\texttt{AS\_Pg}]\!](\texttt{R}_0) = \text{lfp}\,{}^{\subseteq}_{\emptyset}\lambda X.\quad (\texttt{R}_0) \cup \{\rho_0 \rightarrow \ \dots\ \rho_n \rightarrow \rho_{n+1} \mid \rho_{n+1} \in (\texttt{AS\_Env}\times \texttt{AS\_Store}) \wedge \{\rho_0 \rightarrow\ \dots\ \rho_n\} \in X \wedge \rho_n \rightarrow \rho_{n+1}\}$$

Definition 3.3.9. (Method Semantics: $\mathbb{M}_{AS}$ $[\![.\ ]\!]$)
Let $\texttt{AS\_Mth} = \langle \texttt{AS\_M}, \texttt{AS\_Pre}, \texttt{AS\_Pos}, \texttt{AS\_Cst}\rangle$ be an AsmL method. Then, the semantics of $\texttt{AS\_Mth}$, $\mathbb{M}_{AS}$ $[\![\texttt{AS\_m}\ ]\!]) \in \mathcal{P}(\texttt{AS\_Env}\times\texttt{AS\_Store}) \rightarrow \mathcal{P}(\mathcal{T}(\texttt{AS\_Env}\times \texttt{AS\_Store}))$ is

$$\mathbb{M}_{AS}\ [\![\texttt{AS\_m}]\!](\texttt{R}_0, \texttt{M}, \texttt{Pre}, \texttt{Pos}, \texttt{Cst}) =$$

$$\text{lfp}\,_{\emptyset}^{\subseteq}\ \lambda X,\ m,\ spre,\ spos,\ scst.\ (\texttt{R}_0) \cup \{\rho_0 \rightarrow\ \ldots\ \rho_n \rightarrow \rho_{n+1} |\ \rho_{n+1} \in (\texttt{AS\_Env}\times\ \texttt{AS\_Store}) \wedge \{\rho_0 \rightarrow\ \ldots\ \rho_n\} \in X \wedge \rho_n \rightarrow \rho_{n+1} \wedge \rho_{n+1}(X) = (m, spre,\ spos, scst) \wedge spre = spos = scst = true\}$$

AsmL Class Semantics

The AsmL class constructor is a default OO constructor. It can be defined according to the Definition 3.8 in [71].

In a general OO context, such as Java, an object can be defined as a set of states including a first (initial) state representing the object just after its creation and a set of states resulting from the interaction of the object with its context [71]. In this case, the interaction can happen in two ways: (1) the context invokes an object's method, or (2) the context modifies a memory location reachable from the object's environment. In [71], this interaction was very well defined using two functions $\texttt{next}_{\texttt{d}}$, for direct interactions, and $\texttt{next}_{\texttt{ind}}$ for indirect interactions and the object semantics, $\mathbb{O}\ [\![\text{o}]\!]$, was defined as:

$$\mathbb{O}[\![\text{o}]\!](\texttt{v,s}) = \text{lfp}\,_{\emptyset}^{\subseteq}\quad \lambda T.\ \texttt{S}_0\langle v, s\rangle \cup \{\sigma_0 \xrightarrow{l_0} \ldots \xrightarrow{l_{n-1}} \sigma_n \xrightarrow{l_n} \sigma' | \{\sigma_0 \xrightarrow{l_0} \ldots \xrightarrow{l_{n-1}} \sigma_n \in T,\ \texttt{next}(\sigma_n)\ \ni\ \langle\sigma', l'\rangle\}$$

where $\text{next}(\sigma) = \texttt{next}_{\texttt{d}}(\sigma) \cup \texttt{next}_{\texttt{ind}}(\sigma)$, l is a transition label and $\texttt{S}_0\langle v, s\rangle$ is a set of initial states.

In addition to the semantics definition of an OO object in [71], an AsmL method can be activated by an update instruction. This interaction is a hybrid direct/indirect interaction because, according the state of the program events (that maybe external to the object), invoke directly the concerned methods. In following, we will define the interaction states, then, we will provide the complete definition for the direct, indirect and AsmL specific interaction functions.

Definition 3.3.10. (Interaction States)

The set of interaction states is $\sum = \texttt{AS\_Env}\ \times\ \texttt{AS\_Store}\ \times\ \texttt{D}_{\texttt{out}}\ \times\ \mathcal{P}(\texttt{Addr})$

After the creation of the object, the reached states represent the initial states defined as follows:

Definition 3.3.11. (Initial States $\mathtt{S}_0\langle v_{as}, s_{as}\rangle$)
Let $v_{as} \in \mathtt{D_{in}}$ be an AsmL object input value, $s_{as} \in$ `AS_Store` a store at object creation time and `AS_Obj` an AsmL object. The set of initial states of `AS_Obj` is:

$$\mathtt{S}_0\langle v_{as}, s_{as}\rangle = \{\langle e'_{as}, s'_{as}, \phi, \emptyset\rangle \mid \mathbb{P}_{Ctr}[\![\mathtt{AS\_Ctr}]\!](\mathtt{L}_M) \ni \langle e'_{as}, s'_{as}\rangle\}$$

where: $\mathtt{L}_M = \{m_1, \ldots, m_i, \ldots, m_n\}$ is a list of the methods.

In Definition 3.3.11 ϕ is a void value ($\in \mathtt{D_{out}}$) meaning that the constructor does not return any value and therefore does not expose any address to the context.

Definition 3.3.12. (Transition Labels: `Label_AS`)
The set of transition labels is $\mathtt{Label\_AS} = (\mathtt{Mth} \times \mathtt{D_{in}}) \cup \{k\}$.

In Definition 3.3.12 we distinguish two types of interactions corresponding respectively to: (1) invoking a method (direct interaction); and (2) modifying the memory location that is reachable from the object environment (indirect interaction). The transition function `next_AS` is made up of two functions: $\mathtt{next\_AS_{dir}}$ and $\mathtt{next\_AS_{ind}}$.

Definition 3.3.13. (Direct Interactions: $\mathtt{next\_AS_{dir}}$)
Let $\langle e_{as}, s_{as}, v_{as}, \mathtt{Esc}\rangle \in \sum$ be an interaction state. Then, the direct interaction function $\mathtt{next\_AS_{dir}} \in [\sum \rightarrow \mathcal{P}(\sum \times \mathtt{Label\_AS})]$ is defined as:

$$\begin{aligned}&\mathtt{next\_AS_{dir}}(\langle e_{as}, s_{as}, v_{as}, \mathtt{Esc}\rangle) = \\ &\quad \{\langle\langle e'_{as}, s'_{as}, v'_{as}, \mathtt{Esc'}\rangle, \langle mth, v_{in}\rangle\rangle \mid mth \in \mathtt{Mth},\ v_{in} \in \mathtt{Din}, \\ &\quad \mathbb{C}[\![\mathtt{mth}]\!](v_{in}, e_{as}, s_{as}) \ni (v'_{as}, e'_{as}, s'_{as}),\ \mathtt{Esc'} = \mathtt{Esc} \cup reachable(v'_{as}, e'_{as})\}.\end{aligned}$$

where $\mathbb{C}[\![\mathtt{mth}]\!]$is the semantics of a generic OO method as defined in [71].

The function *reachable* is an extension of the helper function of the one defined in [71]. For instance, given an address v_{as} and a store s_{as}, *reachable* determines all the addresses that are reachable from v_{as}. In the AsmL context, this function acts only on the data members of the class according to the following recursive definition:

Definition 3.3.14. (The function *reachable*)
The function *reachable* $\in [\mathsf{D}_{\mathsf{out}} \times \texttt{AS\_Store}] \rightarrow \mathcal{P}(\texttt{Addr})$ is defined as follows:

$$
\begin{aligned}
&\mathit{reachable}(v_{as},s_{as}) = \\
&\quad \text{if } v_{as} \in \texttt{Addr} \text{ then} \\
&\qquad\qquad \{\texttt{Addr}\} \cup \{\mathit{reachable}(e'_{as}(d_{mem}),s'_{as}) \mid \exists\ \texttt{as\_class} = \\
&\qquad\qquad \langle \texttt{AS\_DMem}, \texttt{AS\_Mth}, \texttt{AS\_Ctr}\rangle,\ d_{mem} \in \texttt{AS\_DMem},\ s_{as}(v_{as}) \\
&\qquad\qquad \text{is an instance of } \texttt{as\_class},\ s_{as}(s_{as}(v_{as})) = e'_{as}\} \\
&\quad \text{else } \emptyset.
\end{aligned}
$$

The second possible interaction corresponds to indirect interaction, which may happen when an address escapes from an object. In that case, the context can modify the content of this address with any value. The function $\texttt{next\_AS}_{\texttt{ind}}$ defines this type of interaction:

Definition 3.3.15. (Indirect Interactions: $\texttt{next\_AS}_{\texttt{ind}}$)
Let $\langle e_{as}, s_{as}, v_{as}, \texttt{Esc}\rangle \in \sum$ be an interaction state. Then, the indirect interaction function $\texttt{next\_AS}_{\texttt{ind}} \in [\sum \rightarrow \mathcal{P}(\sum \times \texttt{Label\_AS})]$ is defined as:

$$
\begin{aligned}
&\texttt{next\_AS}_{\texttt{ind}}(\langle e_{as}, s_{as}, v_{as}, \texttt{Esc}\rangle) = \\
&\quad \{\langle\langle e_{as}, s'_{as}, \phi, \texttt{Esc}\rangle, k\rangle \mid \exists\ \alpha \in \texttt{Esc}.\ s'_{as} \in \mathit{update\_as}(\alpha, s'_{as})\}.
\end{aligned}
$$

The *update_as* function is an extension of the *update* function defined in [71] in the sense that it considers AsmL updates in addition to variables. It is defined in following:

Definition 3.3.16. (The function *update_as*)
The function *update_as* $\in [\texttt{Addr} \times \texttt{AS\_Store} \rightarrow \mathcal{P}(\texttt{AS\_Store})]$ is defined as follows:

$$\mathit{update\_as}(\alpha,s_{as}) = \{s'_{as} \mid \exists\ v \in \texttt{Val}.\ s'_{as} = s_{as}[\alpha \mapsto v]\ \}.$$

where: *update_as* returns all the possible stores and $s_{as}(\alpha)$ takes all the possible values in the values domain Val.

Using the definitions of $\mathtt{next\_AS_{dir}}$ and $\mathtt{next\_AS_{ind}}$, we define the global transition function next_AS as:

Definition 3.3.17. (Transition Function: next_AS)
Let $\mathtt{st} = \langle e_{as}, s_{as}, v_{as}, \mathtt{Esc}\rangle \in \sum$ be an interaction state. Then, the transition function $\mathtt{next\_AS} \in [\sum \rightarrow \mathcal{P}(\sum \times \mathtt{Label\_AS})]$ is defined as:

$$\mathtt{next\_AS(st)} = \mathtt{next\_AS_{dir}(st)} \cup \mathtt{next\_AS_{ind}(st)}$$

Using the transition function, an AsmL object semantics is defined as follows:

Definition 3.3.18. (AsmL Object: $\mathbb{O}_{AS}[\![\mathtt{o\_AS}]\!]$)
Let $v_{as} \in \mathtt{Val}$ be an AsmL object input value and $s_{as} \in \mathtt{AS\_Store}$ be a store at object creation time. Then the AsmL object semantics, $\mathbb{O}_{AS}[\![\mathtt{o\_as}]\!] \in [\mathtt{D_{in}} \times \mathtt{AS\_Store}] \rightarrow \mathcal{P}(\mathcal{T}(\Sigma))$ is defined as:

$$\begin{aligned}\mathbb{O}_{AS}[\![\mathtt{o\_as}]\!])(v_{as}, s_{as}) = \;& \mathrm{lfp}\, ^{\subseteq}_{\emptyset}\, \lambda T.\; \mathrm{S_0}\langle v_{as}, s_{as}\rangle \cup \{\sigma_0 \xrightarrow{l_0} \dots \xrightarrow{l_{n-1}} \sigma_n \xrightarrow{l_n} \sigma' \mid \\ & \{\sigma_0 \xrightarrow{l_0} \dots \xrightarrow{l_{n-1}} \sigma_n \in T,\ \mathtt{next_{as}}(\sigma_n) \ni \langle\sigma', l'\rangle\}\end{aligned}$$

where $\mathtt{D_{in}}$ and $\mathtt{D_{out}}$ is the semantic domain for the input and output values, $\sum = \mathtt{AS\_Env} \times \mathtt{AS\_Store} \times \mathtt{D_{out}} \times \mathcal{P}(\mathtt{Addr})$ is a set of interaction states, $\mathtt{next_{as}}(\sigma)$.

Theorem 3.3.1 *Let*

$$\begin{aligned}F_{as} = \lambda T.\;\; & S_0\langle v_{as}, s_{as}\rangle \cup \{\sigma_0 \xrightarrow{l_0} \dots \xrightarrow{l_{n-1}} \sigma_n \xrightarrow{l_n} \sigma' \mid \\ & \{\sigma_0 \xrightarrow{l_0} \dots \xrightarrow{l_{n-1}} \sigma_n \in T,\ next_{as}(\sigma_n) \ni \langle\sigma', l'\rangle\}\end{aligned}$$

Then $\mathbb{O}_{AS}[\![o\_as]\!](v_{as}, s_{as}) = \cup_{n=0}^{\omega} {F_{as}}^n(\emptyset)$

Proof 3.3.1. The proof is immediate from the fixpoint theorem in [25].

Definition 3.3.19. (AsmL Class Semantics: $\mathbb{C}_{AS}[\![\mathtt{c\_as}]\!]$)
Let $\mathtt{c\_as} = \langle \mathtt{as\_dmem}, \mathtt{as\_mth}, \mathtt{as\_ctr}\rangle$ be an AsmL class. The semantics of $\mathbb{C}_{AS}[\![\mathtt{c\_as}]\!] \in \mathcal{P}(\mathcal{T}(\Sigma))$ is:

$\mathbb{C}_{as}[\![\texttt{c\_as}]\!]=\ \{\mathbb{O}_{AS}[\![\texttt{o\_as}]\!](v_{as}, s_{as}) \mid$ o_as is an instance of c_as, v_as $\in$ D_in, s_as $\in$ AS_Store$\}$

Theorem 3.3.2 *(AsmL Class Semantics in Fixpoint) Let*

$$H_{as}\langle S\rangle = \lambda T.\ \ \{S_0\langle v, s\rangle \mid \langle v, s\rangle \in S\ \} \cup \{\sigma_0 \xrightarrow{l_0} \ldots \xrightarrow{l_{n-1}} \sigma_n \xrightarrow{l'} \sigma' \mid \{\sigma_0 \xrightarrow{l_0} \ldots \xrightarrow{l_{n-1}} \sigma_n \in T,\ \textit{next}_{as}(\sigma_n) \ni \langle\sigma', l'\rangle\}$$

Then $\mathbb{C}_{AS}[\![c\_as]\!](v_{as}, s_{as}) = \mathit{lfp}_{\emptyset}^{\subseteq}\ H_{as}\langle\ D_{in}\times\textit{Store}\rangle$

Proof 3.3.2. Although the AsmL model presents some additional functionalities on top of generic OO languages, the proof of this theorem is similar to the proof of Theorem 3.2 in [71]. For instance, considering the definition of $\mathbb{C}_{AS}$ and applying in order Definition 3.3.18, Theorem 3.3.1 and the fixpoint theorem in [25], the proof is straightforward.

3.3.4 Soundness and Correctness of the Class Semantics

The last step in the AsmL fixpoint semantics is to relate the class semantics to the whole AsmL program semantics. For this purpose, we consider updated versions of the functions *split* ($\alpha^{\circ}_{\curlyvee}$), *project* ($\alpha^{\circ}_{\uparrow}$) and *abstract* (α°) as defined in [71]. The new functions are upgraded to support the AsmL update semantics, environment and store.

The basic concept behind defining the object semantics is to cut all the instances not involving the object. For this purpose, two helper functions are required: (1) α_AS$^{\circ}_{\curlyvee}$ that cuts all the traces involving the object instances; and (2) α_AS$^{\circ}_{\uparrow}$ that maps all the cut instances to interaction states.

Let us first define the helper function split_AS, which given a trace τ and an object o_as, it returns a pair consisting of the last state of the prefix of τ made up of the last state of the execution of a method or process of o_as and the remaining suffix of prefix of τ.

Definition 3.3.20. (The Split Helper Function $\texttt{split\_AS}$)
Let $\texttt{o\_as}$ be an AsmL object, $\tau \in \mathcal{T}(\texttt{AS\_Env}\times \texttt{AS\_Store})$, $\texttt{CurMethod} \in \texttt{Mth}$ and $\texttt{pc}_{exit}$ be the exit point of $\tau(0)(\texttt{CurMethod})$. Then $\texttt{split\_AS} \in [(\mathcal{T}(\texttt{AS\_Env}\times \texttt{AS\_Store}) \rightarrow (\texttt{AS\_Env}\times \texttt{AS\_Store}) \times \mathcal{T}(\texttt{AS\_Env} \times \texttt{AS\_Store})]$ is defined as:

$$\begin{array}{lll}
\texttt{split\_AS}(\tau)= & \text{let } n = & \min\{i \in \mathbb{N} \mid \tau(i)=\langle\texttt{CurMethod}\rangle \\
 & & \wedge\ \tau(i)(\texttt{SL})=\texttt{true} \wedge \tau(i)(\texttt{pc})=\texttt{pc}_{exit} \wedge \\
 & & \tau(i)(\texttt{this})= \texttt{o\_as} \wedge \tau(i)(\texttt{StackHeight})= \\
 & & \tau(0)(\texttt{StackHeight})\} \\
 & \text{in} & \langle\tau(n), \tau(n+1) \rightarrow \ldots \rightarrow \tau(\texttt{Len}(\tau)-1)\rangle
\end{array}$$

The cut function $\alpha\_\texttt{AS}^{\circ}_{\times}$ considers four different cases:

1. for empty trace, ϵ, it returns an empty trace.
2. if trace is part of the object trace, then we split it recursively keeping only the last state of the execution of a method or process. The rest of the trace is are removed.
3. If this is not the current object and the store is not changed, then we continue with the rest of the trace.
4. If this is not the current object and the store is changed, then we keep the current trace and we continue with the rest of the traces.

Definition 3.3.21. (Cut Function: $\alpha\_\texttt{AS}^{\circ}_{\times}$)
Let $\texttt{o\_as}$ be an AsmL object, $\tau \in \mathcal{T}(\texttt{AS\_Env}\times \texttt{AS\_Store})$. Then $\alpha\_\texttt{AS}^{\circ}_{\times} \in [(\mathcal{T}(\texttt{AS\_Env}\times \texttt{AS\_Store}) \times \texttt{AS\_Store}) \rightarrow \mathcal{T}(\texttt{AS\_Env}\times \texttt{AS\_Store})]$ is defined as:

$\alpha\_\mathtt{AS}^{\circ}_{\times} = \lambda(\tau, \mathtt{S_{last}}).$

$$\begin{cases} \epsilon & \text{if } \tau = \epsilon \\ \mathtt{let}\ \langle \rho', \tau' \rangle = \mathtt{split\_AS}(\tau) & \\ \mathtt{in\ let}\ \langle e'_{as}, s'_{as} \rangle = \rho' & \text{if } \tau = \langle e_{as}, s_{as} \rangle \to \tau'', e_{as}(\mathtt{this}) = \mathtt{o\_as} \\ \mathtt{in}\ \rho' \to \alpha\_\mathtt{AS}^{\circ}_{\times}(\tau', s'_{as}) & \\ & \text{if } \tau = \langle e_{as}, s_{as} \rangle \to \tau'', \\ \alpha\_\mathtt{AS}^{\circ}_{\times}(\tau'', \mathtt{S_{last}}) & e_{as}(\mathtt{this}) \neq \mathtt{o\_as}, \\ & \mathtt{S}_{/\mathtt{S(o\_as)}} = \mathtt{S}_{\mathtt{last}/\mathtt{S(o\_as)}} \\ & \text{if } \tau = \langle e_{as}, s_{as} \rangle \to \tau'', \\ \langle e_{as}, s_{as} \rangle \to \alpha\_\mathtt{AS}^{\circ}_{\times}(\tau'', \mathtt{S}) & e_{as}(\mathtt{this}) \neq \mathtt{o\_as}, \\ & \mathtt{S}_{/\mathtt{S(o\_as)}} \neq \mathtt{S}_{\mathtt{last}/\mathtt{S(o\_as)}} \end{cases}$$

The second part of the abstraction includes the $\alpha\_\mathtt{AS}^{\circ}_{\uparrow}$ function which maps the states of a trace to interaction states.

Definition 3.3.22. (Map Function: $\alpha\_\mathtt{AS}^{\circ}_{\uparrow}$)
Let o_as be an AsmL object, $\tau \in \mathcal{T}(\mathtt{AS\_Env} \times \mathtt{AS\_Store})$. Then $\alpha\_\mathtt{AS}^{\circ}_{\uparrow} \in [(\mathcal{T}(\mathtt{AS\_Env} \times \mathtt{AS\_Store}) \times \mathcal{P}(\mathtt{Addr})) \to \mathcal{T}(\sum)]$ is defined as:

$$\alpha\_\mathtt{AS}^{\circ}_{\uparrow} = \lambda(\tau, \mathtt{Esc}).$$

$$\begin{cases} \epsilon & \text{if } \tau = \epsilon \\ \mathtt{let}\ \langle e_{as}, s_{as}\rangle = \rho \\ \mathtt{in\ let\ Esc}' = \mathtt{Esc} \cup \\ \quad \mathtt{reachable\_AS}(\rho(\mathtt{retVal}), s_{as}) & \text{if } \tau = \rho \to \tau', e_{as}(\mathtt{this}) = \mathtt{o\_as} \\ \mathtt{in}\ \langle\langle e_{as}, s_{as}, \rho(\mathtt{retVal}), \mathtt{Esc}\rangle, \\ \langle \rho(\mathtt{curMethod}), \rho(\mathtt{inVal})\rangle\rangle \\ \to \alpha\_\mathtt{AS}^{\circ}_{\uparrow}(\tau', \mathtt{Esc}') \\ \mathtt{let}\ \langle e_{as}, s_{as}\rangle = \rho \\ \mathtt{in}\ \langle\langle e_{as}, s_{as}, \phi, \mathtt{Esc}\rangle, k\rangle & \text{if } \tau = \rho \to \tau', e_{as}(\mathtt{this}) \neq \mathtt{o\_as} \\ \to \alpha\_\mathtt{AS}^{\circ}_{\uparrow}(\tau', \mathtt{Esc}) \end{cases}$$

The abstraction function $\alpha\_\mathtt{AS}^{\circ}$ projects from the traces of an execution of the set of relevant states to a specific object.

Definition 3.3.23. (Abstract Function: $\alpha\_\mathtt{AS}^{\circ}$)
Let o_as be an AsmL object, $\mathtt{T} \subseteq \mathcal{T}(\mathtt{AS\_Env} \times \mathtt{AS\_Store})$ a set of execution traces and $\mathtt{s}_{\emptyset}$ the empty store. The abstraction function $\alpha\_\mathtt{AS}^{\circ} \in [(\mathcal{T}(\mathtt{AS\_Env} \times \mathtt{AS\_Store}) \to \mathcal{P}(\mathcal{T}(\sum))]$ is defined as:

$$\alpha\_\mathtt{AS}^{\circ}(\mathtt{T}) = \{\alpha\_\mathtt{AS}^{\circ}_{\uparrow}(\alpha\_\mathtt{AS}^{\circ}_{\times}\ (\tau, \mathtt{s}_{\emptyset}), \emptyset) \mid \tau \in \mathtt{T}\ \}$$

Theorem 3.3.3 *(Soundness of $\mathbb{C}_{AS}[\![c\_as]\!]$)*
Let P_{AS} be a whole AsmL program and let $c_{AS} \in C_{AS}$. Then

$$\forall\ R_0 \in \mathit{AS\_Env} \times \mathit{AS\_Store}.\ \forall\ \tau \in \mathcal{T}(\mathit{AS\_Env} \times \mathit{AS\_Store}).\ \tau \in \mathbb{W}[\![\mathit{AS\_Pg}]\!](R_0)$$
$$:\ \exists \tau' \in \mathbb{C}_{AS}[\![c_{AS}\ ]\!].\ \alpha\_\mathit{AS}^{\circ}(\{\tau\}) = \{\tau'\}$$

Proof 3.3.3. (Sketch) We have to consider both cases when τ contains an object $\mathtt{o}_{AS}$, instantiation of $\mathtt{m}_{AS}$, and when it does not include any $\mathtt{o}_{AS}$. For the second situation, the proof of the theorem is trivial considering that τ will be an empty trace. In the first case, the trace is not empty (let it be τ''). Since AsmL classes are initialized in the main program `Main` before the execution starts, there exists an initial environment, store and set of variables that define the initial trace $\sigma_0 \in \tau''$. The rest of the traces in τ'' are interaction states of o_{AS} because they are obtained by applying α_AS° on τ. Therefore, $\tau'' \in \mathbb{C}_{AS}[\![m_{AS}]\!]$.

Theorem 3.3.4 *(Completeness of* $\mathbb{C}_{AS}[\![]\!]$*)*
Let c_{AS} *be an AsmL class. Then*

$$\forall \tau \in \mathcal{T}(\Sigma).\quad \tau \in \mathbb{C}_{AS}[\![c_{AS}]\!]\text{: } \exists\ AS\_P \in \langle L_{AS\_Pg}\rangle.\ \exists \rho_0 \in AS\_Env \times AS\_Store.\ \exists\ o_{AS} \textit{ instance of } c_{AS}.\ \textit{exists } \tau' \in \mathcal{T}(AS\_Env \times AS\_Store).\ \tau' \in \mathbb{W}[\![\rho_0]\!] \wedge \alpha\_AS^\circ(\{\tau'\}) = \{\tau\}$$

Proof 3.3.4. An AsmL program satisfying the previous theorem can be constructed by creating and instance of c_{AS} in the `Main` function, the initial state corresponds to the state when the class constructor, `AS_Ctr`, was executed. It is always possible to construct both `AS_P` and ρ_0. For instance, there exist many other possible constructions involving AsmL methods pre-conditions and post-conditions.

3.4 Program Transformation

The equivalence in behavior, with respect to the observation α_o, between the source SystemC program and the target AsmL program is required to ensure the soundness of any verification result at the AsmL level. Our objective is to define a relation between the SystemC processes active for certain delta cycle and the set of methods allowed to be executed in the AsmL model. Hence, we will map every thread (method, sensitivity list) in the SystemC program by a method (method core, pre-condition) in the AsmL program to ensure having set of variables in both programs updated in the same time with the same values.

The SystemC observation function needs to see all the active processes at the beginning of a delta-cycle by checking for the end of the update phase.

Definition 3.4.1. (SystemC Observation Function: α_o^{SC})
Let SC_Pg= $\langle$L$_{\texttt{SC\_Mod}}$, SC_main$\rangle$ be a SystemC program, the observation function α_o^{SC} $\in \mathcal{P}$(SC_Env×SC_Store) $\rightarrow \mathcal{P}(\mathcal{T}$(SC_Env× SC_Store)) is

$\alpha_o^{SC}[\![$SC_Pg$]\!]$(R$_0$) =

$$\text{lfp}\ {}^{\subseteq}_{\emptyset}\lambda X.(\text{R}_0) \cup \{\tilde{\rho}_0 \rightarrow \ldots \tilde{\rho}_n | \ \forall \tilde{\rho}_i \in (\texttt{SC\_Env}\times\ \texttt{SC\_Store})\ \exists\ \{\rho_0^i \rightarrow \ldots \rho_m^i\} \in X \ \wedge\ \rho_m^i \rightarrow \tilde{\rho}_i \ \wedge \ \{\ \texttt{m\_sc}\ \text{in}\ \mathbb{M}_{SC}\ |\ \exists \texttt{o\_sc} \in \mathbb{M}_{SC}.\ \texttt{o\_sc}(\rho_m^i()) \neq \{\epsilon\}\ \} = \emptyset\}$$

In the previous definition, α_o^{SC} is only tracing the initial states of a simulation cycle. For instance, the third condition confirms that in the last simulation cycle there was no single process ready to run. Similarly, we define an observation function α_o^{AS} for an AsmL program.

Definition 3.4.2. (AsmL Observation Function: α_o^{AS})
Let AS_Pg= $\langle$L$_{\texttt{AS\_C}}$, INIT$\rangle$ be an AsmL program, the observation function α_o^{AS} $\in$ $\mathcal{P}$(AS_Env×AS_Store) $\rightarrow \mathcal{P}(\mathcal{T}$(AS_Env× AS_Store)) is

$\alpha_o^{AS}[\![$AS_Pg$]\!]$(R$_0$) =

$$\text{lfp}\ {}^{\subseteq}_{\emptyset}\lambda X.(\text{R}_0) \cup \{\tilde{\rho}_0 \rightarrow \ldots\ \tilde{\rho}_n | \ \forall \tilde{\rho}_i \in (\texttt{SC\_Env}\times\texttt{AS\_Store})\ \exists\ \{\rho_0^i \rightarrow\ \ldots\ \rho_m^i\} \in X \ \wedge\ \rho_m^i \rightarrow \tilde{\rho}_i \ \wedge \ \{\ \texttt{m\_as}\ \text{in}\ \mathbb{C}_{AS}\ |\ \exists \texttt{o\_as} \in \mathbb{C}_{AS}.\ \texttt{o\_as}(\rho_m^i()) \neq \{\epsilon\}\ \} = \emptyset\ \}$$

Next, we define the notion of equivalence between the two observations. Although, SystemC and AsmL have different environment and store structures, it is possible to ensure that they contain the same information.

Definition 3.4.3. (Equivalence w.r.t. α_o: $\equiv_{\alpha_o}$)
Let SC_Pg be a SystemC program, V_sc a set of its variables, AS_Pg be an AsmL program and Dout_as a set of its output variables.

prog_sc $\equiv_{\alpha_o}$ prog_as if

$\forall$R$_0^{SC}$ set of initial states of SC_Pg. $\forall$R$_0^{AS}$ set of initial states of AS_Pg.

$\forall\tilde{\rho} \in \{\tilde{\rho}_0 \rightarrow\ \ldots\ \rightarrow \tilde{\rho}_n\} \in \alpha_o^{SC}[\![SC_Pg]\!]$(R$_0^{SC}$).

$\exists\hat{\rho} \in \{\hat{\rho}_0 \rightarrow \ \dots \ \rightarrow \hat{\rho}_n\} \in \alpha_o^{AS}[\![\texttt{AS\_Pg}]\!](\mathrm{R}_0^{AS})$ |

$\forall$ vsc $\in$ V_sc. $\exists$ vas $\in$ V_as such that

if vsc $\in$ SC_Sig then $(\tilde{\rho}(\texttt{vsc}) = (\texttt{vl1},\texttt{vl2})) \wedge (\hat{\rho}(\texttt{vas}) = (\texttt{vl1},\texttt{vl2}))$

if vsc $\in$ AS_DMem then $(\tilde{\rho}(\texttt{vsc}) = \texttt{vl1}) \wedge (\hat{\rho}(\texttt{vas}) = (\texttt{vl1},\texttt{vl1}))$

The observation function ensures that the AsmL program is mimicking the *evaluate* and *update* phases (same length n of the ρ sets). The first if condition takes care of the SystemC signals while the second one concerns basic C++ variables.

Theorem 3.4.1 *(Existence of Transformed AsmL Program w.r.t. α_o^{SC})*
Let SC_Pg be a whole SystemC program, SC_Din a set of inputs and SC_Dout a set of outputs. Then

$\exists$ *AS_Pg, an AsmL program, such that SC_Pg $\equiv_{\alpha_o}$ AS_Pg*

Proof 3.4.1. (Sketch) The proof is done by constructing the AsmL program. For instance, for every SystemC module we affect an AsmL class having the same data members and methods. We set the pre-conditions, AS_Ctr, for the AsmL methods as a conjunction of the state of the events present in the sensitivity list, SC_SL, of the SystemC program processes. The tricky point in the construction is when to make the updates in the AsmL program. We have two possibilities: (1) C++ variables update: whenever a C++ variable is involved in an instruction, a partial update can be applied using the notion of *binders* in AsmL; and (2) SystemC signals: all signals are updated when all methods pre-conditions are false. Once the set of AsmL classes defined, Theorem 3.3.4 ensures the existent of the AsmL program.

Theorem 3.4.2 *(Soundness of the transformation)*
Let SC_Pg be a whole SystemC program and let AS_Pg be a whole AsmL program. Then

SC_Pg $\equiv_{\alpha_o}$ AS_Pg : $\quad \forall$ *Prop(V_sc,$\tilde{\rho}$) | $\tilde{\rho} \in \alpha_o^{SC}[\![SC\_Pg]\!]$.*
SC_Pg $\vdash$ Prop(V_sc,$\tilde{\rho}$) : AS_Pg $\vdash$ Prop(V_as,$\hat{\rho}$) | $\hat{\rho} \in \alpha_o^{AS}[\![AS\_Pg]\!]$.

where Prop is a program's property, V_sc is a set of variables of the SystemC program, V_as are their corresponding variables in the AsmL program.

Proof 3.4.2. The proof is straightforward from the construction of equivalence relation $\equiv_{\alpha_o}$ in Definition 3.4.3.

3.5 Summary

We presented in this Chapter the fixpoint semantics of the SystemC library including, in particular, the semantics of a SystemC module that we proved to be sound and complete w.r.t. a trace semantics of a SystemC program. We provided also the denotational semantics of a subset of AsmL and we proved the soundness and completeness of an AsmL class w.r.t. to a trace semantics of the AsmL program. Then, we proved the existence, for every SystemC program, of an AsmL program having similar behavior w.r.t. an observation function that we set to consider the traces of the system just after the update phase of the SystemC simulator. In the next four Chapters we are going to present the techniques we used in our proposed SoC framework for both direct and Asml based approaches.

Chapter 4

Static Code Analysis

4.1 Introduction

In this chapter we introduce an abstraction framework for object-oriented languages. Then, we provide the abstract environment for SystemC. Finally, we illustrate our proposed methodology to statically analyze SystemC design by considering a Packet Switch design from the SystemC library.

4.2 Abstraction Framework for OO Languages

Program analysis by abstract interpretation (also called semantical analysis) is a method that computes, automatically, an approximative description of the behavior of a program when executed. This description can be a property like "a variable x is always negative" or "at the entry of a function, the variable x is equal to the value of the variable y". These properties are considered as invariant properties because they are always true for every program execution. Let us consider the following example program:

```
read(x0)
x := x0
```

```
  while (x < 100) do
    begin
      write(T[i])
      x := x + 1
    end
```

which requires the user to enter a value $x0$ then prints the elements of the table $T[x]$ having their indexes x between $x0$ and $x0 + 100$. The following two properties:

- $x0 \leq x \leq 100$ at the entry of the loop.
- $x0 \geq 101$ at the end of the program.

are invariant properties, in other terms they are always true for any value of $x0$ entered by the user. Determining these properties automatically is important for program analysis and in particular bug detection.

Abstract interpretation was first introduced by Patrick Cousot [21] in 1977 to analyze flow chart based languages (not procedural languages). Then, both Patrick and Radhia Cousot [22] upgraded their proposal to support recursive procedures. The objective of the technique was to formally design approximate semantics of programs which can be used to gather information about programs in order to provide sound answers to questions about their run-time behaviors. These semantics can be used to design manual proof methods or to specify automatic program analyzers.

Formally, given a concrete domain C and a semantic function $[\![.]\!] : \mathit{Program} \rightarrow C$ associating to each program $P \in \mathit{Program}$ its semantics $[\![P]\!]$, an abstract interpretation can be formulated by defining a corresponding abstract domain A and an abstract semantic function $[\![.]\!]^{\#} : \mathit{Program} \rightarrow A$ which approximates the concrete one. The notion of approximation is encoded in abstract interpretation by suitable partial order on domain's objects. Both C and A are assumed to be complete lattices with respect to this approximation order. The ordering on the concrete and abstract domains describe the relative precision of domains values, somehow in a dual fashion with respect to the standard domains for denotational semantics.

The relationship between the concrete and abstract setting is given by a pair of functions, namely the abstraction map $\alpha : C \rightarrow A$ and the concretization map $\gamma : A \rightarrow C$, where, $\forall a \in A$, $\alpha(c) \leq_A a$, or, equivalently, $\forall c \in C$, $c \leq_C \gamma(a)$. This means that a is a concrete approximation of c.

The classical verification methodology, as defined by Cousot [21], defines an adjunct couple of functions between concrete and abstract semantics. However, this approach can be applied only for semantics based abstractions. For syntax based abstraction, where the program is represented in the collecting semantics as a set of properties, there is no Galois connection to abstract semantics. In fact, several properties can have the same meaning and may orient the analysis of the program differently. Therefore, the link between the abstract and collecting semantics is no more defined by an adjunct Galois Connection but using a *meaning* function γ [10]. The general schematic of the approach is given in Figure 4.1.

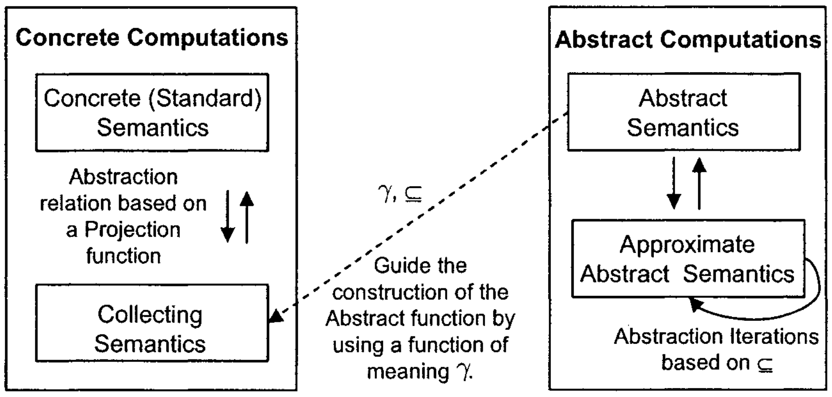

Concrete Computations
Concrete (Standard) Semantics
Abstraction relation based on a Projection function
Collecting Semantics
γ, ⊆
Guide the construction of the Abstract function by using a function of meaning γ.
Abstract Computations
Abstract Semantics
Approximate Abstract Semantics
Abstraction Iterations based on ⊆

Figure 4.1: Abstraction Methodology.

The general methodology of Figure 4.1 requires the definition of a concrete semantics then a collecting semantics to collect the properties to analyze.

The projection of the concrete semantics to the collecting semantics defines the properties associated to the concrete semantics. The collecting semantics is constructed by induction. The main constructions cover basic types, pairs, tuples, functions and sets.

The link between the abstract semantics and the collecting semantics is performed using a *meaning* function γ. In general there is no abstraction function corresponding to this function because the abstract domain corresponds to a number of properties and a set of properties may have the same semantical meaning and orients the analysis differently.

4.2.1 Constructing Collecting Semantics

A fundamental step in our abstraction methodology is to define the collecting semantics which will allow us to perform total and precise analysis. It defines statically the future domains that will serve for the analysis and their specific manipulations. It is not possible to change the domain during the analysis which presents a drawback for the technique.

In our proposed collecting semantics we distinguish two notions: the syntax and the semantics. For basic domains, natural numbers for example, both concepts are confused. Nevertheless, for complex domains they are separated. Syntactical constructions are algorithmic. Semantical constructions, on the other hand, are "meaning" constructions because they correspond directly to mathematical constructions.

As shown in Figure 4.2, we define a collecting semantics associated to X by:

- A domain X_c augmented by syntactical operations.
- A number of theorems (rewriting rules) valid in the concrete semantics.

We define an element from the collecting semantics as a combination of:

- An element $x^c \in X_c$ representing a property.

- A link to the environment of properties: for every symbol in the x^c corresponds a number of properties that must be verified by this symbol.

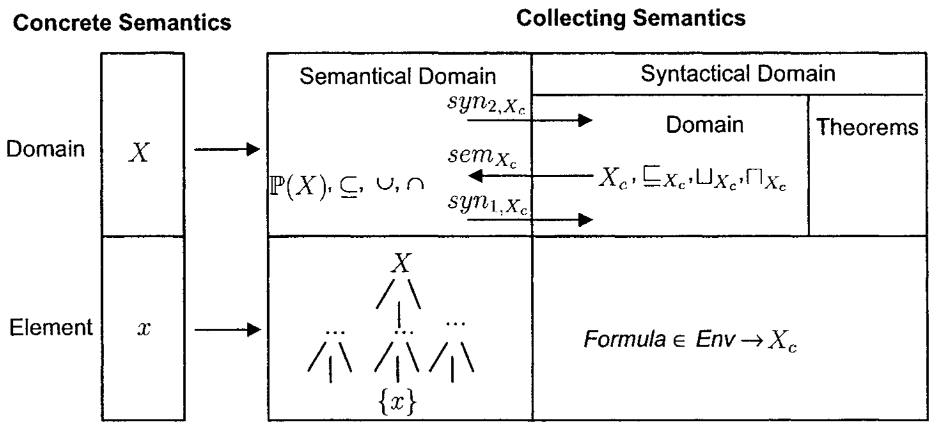


Figure 4.2: Collecting Semantics.

The environment corresponds to all the collecting semantics objects that are introduced by the program. In the rest of this Section, we will focus on the construction of the domain X_c but we should always take to consideration the existence of theorems and rewriting rules. These theorems describe syntactical properties of the programs functions. For example, the analysis of a function performing the summation of two elements supposes the addition operator is implemented by a total order that does not modify the memory. The properties of total order are described then as syntactical theorems.

The general methodology of construction is done by induction on the concrete domains X. Syntactical operators are algorithmic operators that take their arguments x^c in a language of formulae defined by X_c. Semantical operations are descriptive; their arguments are concrete elements $x \in X$ that verify the property x_c. This construction was first introduced by [76] for general types.

For every collecting domain X_c, correspond two basic operations: a projection function p_X and and order of approximation $\sqsubseteq_{X_c}$. $p_X : X \rightarrow X_c$ computes for every concrete element $x \in X$ the most precise property verified by x. The syntactical order $\sqsubseteq_{X_c}$ defines the relative precision between two properties.

As we previously noticed, considering complex systems induces the separation of both semantical and syntactical domains. Therefore, there exist another order $\sqsubseteq_{sem,X_c}$ which is a semantical order that describes the real precision of the objects. In order to define a set of operations on the collecting semantics, the construction methodology must verify certain properties. These latter define the validity of the syntactical order and define the semantical order.

- $\sqsubseteq_{X_c}$ is the syntactical order associated to the domain X_c.

- $\sqcup_{X_c}$ and $\sqcap_{X_c}$ are the union and intersection operations associated to $\sqsubseteq_{X_c}$.

- The image by p_X is partially atomic (for every element from the concrete semantics, there exists at least one corresponding property in the collecting semantics), which means: $p_X(x) \sqsubseteq_{X_c} \sqcup_{X_c} X^c : \exists x^c \in X_c . p_X(x) \sqsubseteq_{X_c} x^c$. This property is less powerful than total atomicity since it does not prove that there is no contradiction. It ensures however that the union of non-coherent states will be non-coherent.

As we will prove, the previous properties define a double Galois Connection between the collecting domain X_c and the natural domain of properties, $\mathbb{P}$, set of parts of X:

$$X_c \underset{syn_{1,X_c}}{\overset{sem_{X_c}}{\rightleftarrows}} \mathbb{P}(X) \underset{sem_{X_c}}{\overset{syn_{2,X_c}}{\rightleftarrows}} X_c \text{ and } sem_{X_c} \text{ surjective.}$$

where:

$$syn_{2,X_c} \overset{def}{=} \sqcup_{X_c}\{p_X(x) \;/\; x \in S\}$$

$$sem_{X_c} \overset{def}{=} \{x \;/\; p_X(x) \sqsubseteq_{X_c} x^c\}$$

$$syn_{1,X_c} \overset{def}{=} \sqcup_{X_c}\{x^c \;/\; sem_{X_c}(x^c) \subseteq S\}$$

The proof of soundness of the collecting semantics is equivalent to prove that $(sem_{X_c}\,,\, syn_{1,X_c})$ and $(sem_{X_c}\,,\, syn_{2,X_c})$ form two couples of joint Galois functions.

Proof that: $X_c \underset{syn_{1,X_c}}{\overset{sem_{X_c}}{\rightleftarrows}} \mathbb{P}(X)$.

The proof is done in two steps:

Step 1: Proof that $sem_{X_c} \subseteq S$:$x_c \sqsubseteq_{X_c} syn_{1,X_c}(S)$
Since $sem_{X_c} \subseteq S$, then, $x^c \in \{y^c \mathbin{/} sem_{X_c}(y^c) \subseteq S\}$, therefore, $x^c \sqsubseteq_{X_c} syn_{1,X_c}(S)$

Step 2: Poof that $x^c \sqsubseteq_{X_c} syn_{1,X_c}(S)$:$sem_{X_c}(x^c) \subseteq S$
Let us consider $x \in X$ verifying $p_X(x) \sqsubseteq_{X_c} x^c$. The objective is to prove that $x \in S$.
From the definition of p_X, $p_X(x) \sqsubseteq_{X_c} \sqcup_{X_c}\{y^c/y^c \subseteq S\}$.
Or, p_X is atomic, therefore, $\exists x^{c'}/p_X(x) \sqsubseteq_{X_c} x^{c'}$ and $sem_{X_c}(x^{c'}) \subseteq S$.
sem_{X_c} is monotone, meaning: $sem_{X_c}(p_X(x) \subseteq S$
Or, if there exists $y/p_X(x)p_X(y)$, then, $x = y$, which implies for our case that:
$sem_{X_c}(p_X(x) = \{x\}$.
Therefore, $x \in S$.

Proof that: sem_{X_c} is surjective.

Consider $S \subseteq X$,

$S = \cup\{\{x\}/x \in S\} = \cup\{sem_{X_c}(p_X(x))/x \in S\} = sem_{X_c}(\sqcup_{X_c}(p_X(x))/x \in S)$.

The last equality is true because sem_{X_c} is an abstraction operation.

Therefore, sem_{X_c} is surjective.

Proof that: $\mathbb{P}(X) \underset{sem_{X_c}}{\overset{syn_{2,X_c}}{\rightleftarrows}} X_c$ and sem_{X_c} surjective.

The proof is done in two steps:

Step 1: $x^c \sqsubseteq_{X_c} syn_{2,X_c}(S)$:$S \subseteq sem_{X_c}(x^c)$

Let $S \subseteq X$ and $x^c \in X_c$ verifying $x^c \sqsubseteq_{X_c} syn_{2,X_c}(S)$

We have to prove that: $S \subseteq \{x/p_X(x) \sqsubseteq_{X_c} x^c\}$

Consider $x \in S$. From the definition of syn_{2,X_c}, $p_X(x) \sqsubseteq_{X_c} x^c$.

Therefore, $x \in \{y/p_X(y) \sqsubseteq_{X_c} y^c\}$

Step 2: Poof that $S \subseteq sem_{X_c}(x^c)$:$syn_{2,X_c}(S) \sqsubseteq_{X_c} x^c$

Let $x \in S$, we need to prove that $p_X(x) \sqsubseteq_{X_c} x^c$.

Or, $x \in sem_{X_c}(x^c)$ (because of the inclusion), hence, $p_X(x) \sqsubseteq_{X_c} x^c$

The semantical order, $\sqsubseteq_{sem,X_c}$ is then defined as: $X_1^c \sqsubseteq_{sem,X_c} X_2^c \overset{def}{\Leftrightarrow} sem_{X_c}(X_1^c) \subseteq sem_{X_c}(X_2^c)$. The semantical function sem_{X_c} gives the meaning of an element from the collecting semantics, it was firstly defined by [10] as a *meaning* function. The first Galois connection proves the validity of the construction while the second one ensures that every property can be described by an element from the collecting semantics.

Collecting Basic Domains:

Basic domains correspond to bits, Boolean, bytes, natural numbers and reals. Considering such a domain X then the collecting semantics over X is defined by the set of parts of X. The projection function is defined as:

$$\begin{array}{rccl} p_X\colon & X & \rightarrow & X_c \\ & x & \mapsto & \{x\} \end{array}$$

The syntactical order $\sqsubseteq_{X_c}$ is defined as $\subseteq$. Union and intersection operations ($\sqcup_{X_c}$ and $\sqcap_{X_c}$) are defined as $\cap$ and $\cup$ respectively. Syntactical equality is simply defined as:

$$x_1^c =_c x_2^c \overset{def}{=} (x_1^c = \emptyset) \vee (x_2^c = \emptyset) \vee (x_1^c = x_2^c)$$

There are many orders that can be defined on basic domains. In general, they are used accordingly to the requirements of the application. The simplest way to extend the natural order $\leqslant$ to an order $\leqslant_c$ defined as:

$$x_1^c \leqslant_c x_2^c \Leftrightarrow \forall x_1 \in x_1^c \, . \, \forall x_2 \in x_2^c \, . \, x_1 \leqslant x_2$$

The previous order can be eventually extended to an Egli Milner order [34] that represents the classical notion of intervals.

Collecting Pairs:

Pairs are fundamental in all programming languages. For C++ for examples, they are the first step in order to represent tables and lists in general. The direct way to collect pairs is to consider the following projection function:

$$\begin{array}{rccl} p_{X\times Y}\colon & X \times Y & \rightarrow & \mathbb{P}((X_c \setminus \emptyset) \times (Y_c \setminus \emptyset)) \\ & (x,y) & \mapsto & \{p_X(x), p_Y(y)\} \end{array}$$

Removing the empty set from the definition of the projection function ensures that the pairs are defined on consistent property. In other terms, non-consistent properties will be represented by the empty set.

The syntactical order $\sqsubseteq_{(X\times Y)_c}$ over the domain $(X \times Y)_c$ is defined as:

$$P_1^c \sqsubseteq_{(X\times Y)_c} P_2^c \Leftrightarrow \forall\, (x_1^c, y_1^c) \in P_1^c \,.\, \exists (x_2^c, y_2^c) \in P_2^c \,.\, (x_1^c \sqsubseteq_{X_c} x_2^c) \;\&\; (y_1^c \sqsubseteq_{Y_c} y_2^c)$$

Four main operations are usually defined over pairs: fst (for first) and snd (for second), constructing a pair from two components and equality of pairs. fst and snd are defined as:

$$fst\colon \begin{array}{ccc} X \times Y & \to & X \\ (x,y) & \mapsto & x \end{array} \quad \text{and} \quad snd\colon \begin{array}{ccc} X \times Y & \to & X \\ (x,y) & \mapsto & y \end{array}$$

The validity of the operations defined over pairs is ensured by:

$$sem_{X_c}(fst(P^c)) \supseteq fst(sem_{(X\times Y)_c}(P^c))$$

$$sem_{Y_c}(snd(P^c)) \supseteq snd(sem_{(X\times Y)_c}(P^c))$$

$$sem_{(X\times Y)_c}(x^c, y^c) \supseteq \{(x,y) /\ x \in sem_{X_c}(x^c) \ \&\ y \in sem_{Y_c}(y^c)\}$$

$P_1^c =_c P_2^c$:$sem_{(X\times Y)_c}(P_1^c) = sem_{(X\times Y)_c}(P_2^c)$ is a singleton.

The syntactical operations are defined as follows:

$$fst_c(P^c) \overset{def}{=} \sqcup_{X_c}\{x^c \ /\ (x^c, y^c) \in P^c\}$$

$$snd_c(P^c) \overset{def}{=} \sqcup_{Y_c}\{y^c \ /\ (x^c, y^c) \in P^c\}$$

$$(x^c, y^c)_c \overset{def}{=} \{(x^c, y^c)\}$$

$$P_1^c =_c P_2^c \overset{def}{\Leftrightarrow} \forall (x_1^c, y_1^c) \in P_1^c \ .\ \forall (x_2^c, y_2^c) \in P_2^c \ .\ x_1^c =_c x_2^c \ \&\ y_1^c =_c y_2^c$$

Example:

The pairs (x, y) verifying $x \geqslant 5y$ is described as:

$P^c = \exists t \in \mathbb{Z} \,.\, (] - \infty, 4t], t_c)_c \overset{def}{=} \{(] - \infty, 4t], \{t\})/t \in \mathbb{Z}\}$

4.2.2 Constructing Abstract Semantics

Collecting semantics associates a domain of properties to a programming language. It is possible to stop the analysis at this level and perform all the analysis on collecting semantics. Nevertheless, this approach has many drawbacks in particular algorithmic problems. In fact, an analysis program can only handle finite sets which is not the case for elements based on semantical polymorphism. On the other hand, even for relatively simple domains, such as sets, the algorithmic construction may cause state explosion.

This is the reason why abstract semantics was introduced. It maps a property to a finite representation of the property more suitable for the analysis. The connection between the abstract and collecting semantics is performed using the meaning function $\gamma_{abc} \in X_{abs} \rightarrow X_c$. This function is not a concretization function as it is not possible to define a correspondent abstraction function because the concept of the best abstraction cannot be applied to collecting semantics. Therefore, this construction does not fall under the classical concept of the abstract interpretation. This fact will require a complex algorithms for the analysis [72], however, it will offer more flexibility.

Informally, the link between the abstract and collecting semantics can be seen as compilation. In fact, the collecting semantics can be interpreted as a low level language with no loops. However, the abstract semantics is an advanced language.

Semantical domains corresponds to decidable theories over which we can define efficient algorithms. It was very deeply studied in the literature. In general, these are represented as lattices [65]. In contrast, the syntactical abstract domains corresponds to non-decidable theories having theories elements as logical formulae. They are usually defined as programming languages trying to compromise complexity and formal analysis.

Abstract syntax domains allow the syntactical manipulation of expressions in order to perform the analysis of the program. In order to allow both the interaction with the user and abstract debugging we selected the so-called *hypergraph* structure firstly introduced by [105] to represent the abstract environment. This graph is illustrated in Figure 4.3 where we can notice a structural layered representation.

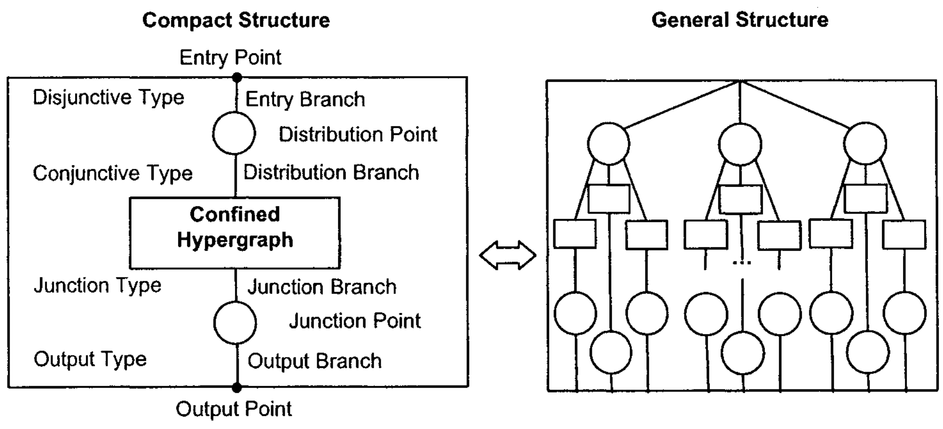


Figure 4.3: Hypergraph General Structure.

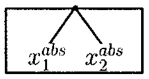
The hypergraph structure can be seen as a general automata connecting its states by branches (also called hyper-branches). Theses branches can be seen as an extension to Binary Decision Diagrams (BDDs) more adapted to programs representation. In other terms, they offer a higher level of abstraction and flexibility by introducing the notion of confined hypergraph. This encapsulation property of the hypergraphs is very suitable to SoC where a system is a connection of modules (IPs) using its input and output ports.

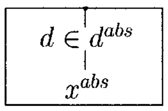
The hypergraphs define a logical language and a control language for both simulation and program proofs. It may offer good solution to control the state explosion problem of model checking. In fact, over the hypergraphs is defined a number of reduction and analysis techniques reducing (abstracting) the concrete system to get to a simplified abstract structure more adequate for model checking and formal techniques in general.

Abstract concrete domains are atomic domains. They usually include natural numbers, real, Boolean, etc. Their collecting domain corresponds to their domains' properties. The main elements of the abstract domain are:

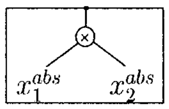
$$\begin{aligned}
\gamma_{X_{abs}}(\boxed{a}) &\overset{def}{=} \{a\} \\
\gamma_{X_{abs}}(\boxed{\top_X}) &\overset{def}{=} X \\
\gamma_{\mathbb{N}_{abs}}(\boxed{[a,b]_{\leq}}) &\overset{def}{=} [a,b]_{\leq_c}
\end{aligned}$$

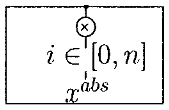
Just like collecting semantics, abstract semantics domains support both syntactical and semantical disjunctions accordingly to the following:

$\gamma_{X_{abs}}($  $)$ $\overset{def}{=}$ $\gamma_{X_{abs}}(\boxed{x_1^{abs}}) \cup \gamma_{X_{abs}}(\boxed{x_2^{abs}})$

$\gamma_{X_{abs}}($  $)$ $\overset{def}{=}$ $\exists d \in \gamma_{D_{abs}}(d^{abs}) \,.\, \gamma_{X_{abs}}(\boxed{x^{abs}})$

where: $\gamma_{X_{abs}}(\boxed{x_1^{abs}})$ and $\gamma_{X_{abs}}(\boxed{x_2^{abs}})$ represent the hypergraphs associated to X_{abs}. To complete the construction of the abstract semantics we need to represent all the operations present in the concrete domain. These operations will be introduced as separation points. The operation of multiplication is, for example, represented as follows:

$\gamma_{\mathbb{N}_{abs}}($  $)$ $\overset{def}{=}$ $\gamma_{X_{\mathbb{N}_{abs}}}(\boxed{x_1^{abs}}) \times_c \gamma_{X_{abs}}(\boxed{x_2^{abs}})$

$\gamma_{\mathbb{N}_{abs}}($  $)$ $\overset{def}{=}$ $\prod_{i=0}^{n} \gamma_{\mathbb{N}_{abs}}(\boxed{x^{abs}})$

Over natural numbers, for example, the basic operations such as the addition $\textcircled{+}$ and multiplication $\textcircled{\times}$ are defined. We also add an operator to represent interval construction $\textcircled{[,]_{\leq}}$.

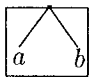
In order to use the previously defined graphical structures, it is required to introduce abstract operations. These will include in particular operations related to the degradation and fusion of hypergraphs. But, let us first introduce the abstract order for these basic domains. For every a and b elements of the concrete domain X

and any confined hypergraph H_x defined from elements in X, we define the following.

$$\begin{aligned}
\boxed{a} \sqsubseteq_{abs} \boxed{b} &\overset{def}{\Leftrightarrow} a = b\\
\boxed{H_x} \sqsubseteq_{abs} \boxed{\top_X} &\overset{def}{=} true\\
\boxed{\top_x} \sqsubseteq_{abs} \boxed{a} &\overset{def}{=} false
\end{aligned}$$

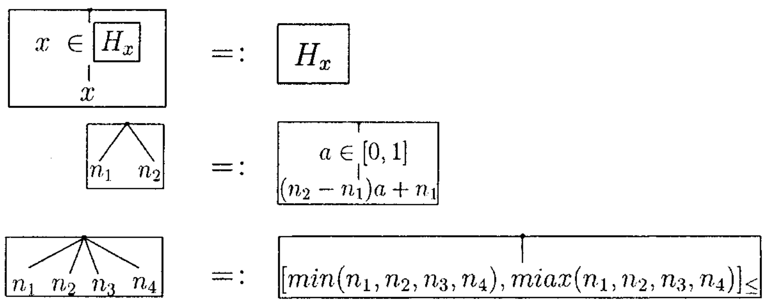
The abstract union and intersection operation are defined as follows:

$$\boxed{a} \sqcup_{abs} \boxed{b} \overset{def}{=}$$



$$\begin{aligned}
\boxed{H_x} \sqcup_{abs} \boxed{\top_X} &\overset{def}{=} \boxed{\top_X}\\
\boxed{a} \sqcap_{abs} \boxed{b} &\overset{def}{=} \begin{cases} \boxed{a} & \text{if } \mathrm{a} = \mathrm{b}\\ \emptyset & otherwise \end{cases}\\
\boxed{H_x} \sqcap_{abs} \boxed{\top_X} &\overset{def}{=} \boxed{H_x}
\end{aligned}$$

Simplifying the hypergraph structure is very important in order to analyze the abstract system. By contrast to the usage of tactics in theorem proving, we define a number of possible simplifications. Eventually, when performing a simplification, a loss of information will occur. Nevertheless, simplification is mandatory in order to analyze the hypergraph. Several simplifications can be defined over the basic domains. We present in following three simple illustrative cases. We consider: x, n_1, n_2 *and* $n_3 \in X$. The symbol $=:$ will be used to represent a simplification operation.



Analyzing a program does not mean to optimize it. In other words, the hypergraph simplification techniques are not defined to reduce the representation but to extract the program properties. In fact, two different simplifications may get to two totally different hypergraph representations. To illustrate this fact, let us consider the following small C++ code:

```
int main() {
  int output = 1;
  int index;
  for(index=0; index<4;index++)
    output *= 2;
  cout << output << endl;
}
```

Figure 4.4: C++ Illustrative Program.

Depending on the reduction technique used, we may get one of the three hypergraphs represented in 4.5. The first representation is very suitable to analyze analytically the value of n. The second, can ensure that the result is always positive. The third one proves that the result is always even. The user will define the most suitable algorithm to apply to his/her program.

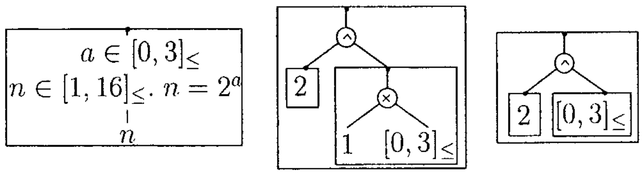


Figure 4.5: Possible Generated Hypergraphs

4.3 SystemC Static Analysis

The previous construction of the collecting and abstract semantics are generic for C++ (and Object-Oriented languages in general). However, when it comes to the verification of SystemC designs, it is required to define a specific abstraction for the SystemC designs. This abstraction will include the SystemC simulation semantics and basic classes.

Static code analysis has its own limitations in particular with respect to the completeness of the analysis [40] (for large-scale programs for instance). There are already some attempts, which try to define abstract environments for C++ [105] that might be used to analyze SystemC programs. However, in order to succeed, such an approach will have to analyze the whole SystemC simulation manager (responsible for the management of process and threads in particular), the SystemC events stack (that contains the list of all the events and their status) and all the classes involved in the program. Even considering the best case when the approach succeeds in handling the complexity of the SystemC structure, it will not extract many properties from the program because it would neglect totally the design structure and the SystemC library semantics. Thus, an efficient SystemC analysis environment has to consider the abstraction of the SystemC library itself in addition to the abstraction of the C++ language. The abstraction has also to target, in addition to the static analysis of the program, the eventual link with other verification techniques, in particular, abstract debugging, model checking and simulation.

To illustrate what is specific to SystemC designs, we will consider the example of Figure 4.6 (which illustrates a general structure of a SystemC design). SystemC programs include, in general, the function *sc_main()* which calls the default *main()* function. The core of this function consists of instructions related to variables declaration, modules configuration, binding establishment, simulation and cleanup.
The Event Manager is defined as a function that associates variables to events in order to identify their status. For instance, this function provides a link between the event status and its identifier of the event inside the program.

4.3.1 Constructing Concrete Semantics

Concrete Semantics

In this paragraph we will define the concrete semantics for the elements that we consider in the abstraction of both C++ language and the SystemC library. Some of the definitions exist already in the literature [105], so we will introduce them in a very short manner providing the required information for the abstraction section.

```
int sc_main(int ac, char * av[]){
  sc_signal < float > in;
  sc_signal < float > sum;                  <== Signals Declaration
  sc_clockclk("CLOCK1", 10, 0.5, 0.0);      <== Clock Definition
  NumGen N("NumGen");
  N(in, clk);                               <== Number Generator Declaration
  TransformUnit TU("Transform");
  TU.in1(in); TU.sum(sum); TU.clk(clk);     <== Tansformation Unit Declaration
  display D("display");
  D(sum);                                   <== Display Declaration
  sc_initialize();                          <== Simulation Initialization
  sc_clock :: start(-1);                    <== Start Simulation
  return 0;                                 <== Exit point of the program
}
```

Figure 4.6: Example of a Generic SystemC Program Structure.

Memory Semantics

Allocation blocks are atomic entities associated with the memory. Every block is defined by its size and a function extracting its bytes. The concrete semantic of such a block is given by the following:

$$\begin{aligned} sem_{con} : AllocBlock &\rightarrow \mathbb{N} \times (\mathbb{N} \rightarrow Byte \cup \{Undefined\}) \\ b &\mapsto (sizeof(b), Alloc(b)) \end{aligned}$$

where: $\begin{cases} \forall i < sizeof(b) & , \; Alloc(b)(i) \in Byte \\ \forall i \geq sizeof(b) & , \; Alloc(b)(i) = Undefined \end{cases}$

Undefined is a non-allocated memory byte. Otherwise, we know nothing about it. When a byte is allocated it belongs then to the type *Byte*.

The *stack* is defined as a table, of a variable size, of Allocation Blocks:

$$\begin{aligned} sem_{con} : Stack &\rightarrow \mathbb{N} \times (\mathbb{N} \rightarrow AllocBlock \cup \{Undefined\}) \\ p &\mapsto (sizeof(p), Alloc(p)) \end{aligned}$$

where: $$\begin{cases} \forall i < sizeof(p) & , \quad Alloc(p)(i) \in AllocBlock \\ \forall i \geq sizeof(p) & , \quad Alloc(p)(i) = Undefined \end{cases}$$

The *program environment* is defined as a function ρ, which provides a link between the memory location of variables and their names inside the program. For instance, this function returns an address l in the stack identifying the variable.

$$\begin{aligned} sem_{con} : ProgramEnvironment &\longrightarrow (Ident \times Stack) \\ \rho &\longmapsto \lambda id.l \end{aligned}$$

Declaration Semantics

When a variable is declared with a certain type, its corresponding allocation block is reserved in the memory. For classes, the reserved memory space will include the static objects, the inherited fields and the virtual methods [32] (if the definition of the class included some virtual methods).

Expression Semantics

Given a stack $Stack$ and a program environment $PEnv$, a program point corresponds to the position of the bullet "•" in the list of instructions.

For variables, for example, two cases are possible: either the variable is affected by value or by address. The following corresponds to the second case:

$Env(Id = a) < \bullet Id, Stack, PEnv > \longrightarrow < Id\bullet, Stack[Block(p, a)], PEnv >$

Regarding classes, the general rule is:

$< \bullet(Object.MemberName), Stack, PEnv > \longrightarrow$
$< (\bullet Object.MemberName), Stack, PEnv >$

Instructions Semantics

The semantic of the expression instructions is basically defined as a removal from the stack of the value that was computed by the expression:

$< \bullet Expr_1;, Stack, PEnv > \longrightarrow < Expr_1\bullet;, (Stack \to b), PEnv >$

$< Expr_1\bullet;, (Stack \to b), PEnv > \longrightarrow < Expr_1;\bullet, (Stack \to b), PEnv >$

The semantic of the selection instruction is:

$< \bullet(if\ (Expr)\ Instr), Stack, PEnv > \longrightarrow < if\ (\bullet Expr)\ Instr, Stack, PEnv >$

According to the if condition, two cases are possible:

- $AllocBloc(b) = true$

  $< if\ (\bullet Expr)\ Instr, Stack, PEnv > \longrightarrow < if\ (Expr)\ \bullet Instr, (Stack \rightarrow b), PEnv >$

  $< if\ (Expr)\ \bullet Instr, (Stack \rightarrow b), PEnv > \longrightarrow < (if\ (Expr)\ Instr)\bullet, Stack, PEnv >$

- $AllocBloc(b) = false$

  $< if\ (\bullet Expr)\ Instr, Stack, PEnv > \longrightarrow < (if\ (Expr)\ Instr)\bullet, Stack, PEnv >$

```
SC_MODULE(module_name){
 //ports, data members, member functions        ⟸  Base section
 //processes etc.
 SC_CTOR(module_name){
 //body of constructor, process registration,    ⟸  Construction section
 //sensitivity lists, module instantiations etc.
 };
}
```

Figure 4.7: SystemC Module.

SystemC Classes Semantics

Most of SystemC classes include two sections: a *base* section where the class members and methods are defined just like C++ classes and a *construction* section where the information related to the simulation execution is introduced (declarations of processes, threads, events, etc.). Both sections are illustrated in Figure 4.7 for the case of SystemC modules. The first part is general however the second part is specific to the SystemC library since it includes the list of processes, threads and their activation conditions.

SystemC Simulator Semantics

The Event Stack will keep the traces of all events and their status (active, idle, etc.), it is defined as:

$$\begin{aligned} sem_{con} : EventsStack &\rightarrow \mathbb{N} \times (\mathbb{N} \rightarrow EventsStatus \cup \{Undefined\}) \\ b &\mapsto (EventID, Status) \end{aligned}$$

The Event Manager is defined as a function that associates variables to events in order to identify their status. For instance, this function provides a link between the event status and its identifier of the event inside the program.

$$\begin{aligned} sem_{con} : EventManager &\rightarrow Ident \rightarrow EventsStack \\ \zeta &\mapsto \lambda id.l \end{aligned}$$

The semantics of the SystemC simulator can be seen as a succession of eight steps. A delta-cycle (evaluate and update phases) is defined by the steps 2, 3 and 4.

1. *Initialization Phase.*

2. *Evaluate Phase*: Select a process from the ones that are ready to run. The order in which processes are selected is unspecified. The execution of a process may cause immediate event notifications to occur, possibly resulting in additional processes becoming ready to run in the same evaluate phase.

3. Repeat Step 2 for any other processes that are ready to run.

4. *Update Phase*: Execute any pending updates resulting form the processs execution in the evaluate phase.

5. If there are pending notifications (inter-process messages or timed events like clocks for example), determine which processes are ready to run and go to step 2.

6. If there are no more timed event notifications, the simulation is done.

7. Else, advance the current simulation time to the time of the earliest (next) pending timed event notification.

8. Determine which processes become ready to run due to the events that have pending notifications at the current time. Go back to step 2.

In summary, the SystemC simulation semantics can be regarded as an execution of the algorithm shown in Figure 4.8 The first step in the procedure, which corresponds to the initialization phase, is defined syntactically in the compilation phase. The infinite while loop is executed until the simulation is exited otherwise it loops infinitely.

```
While(true) {
    While(NotEmpty(EventStack)) {
         Enumerate all active processes
         Select an active process
         Execute the active process
         Update the active processes list
     }
     Update the EventsList
}
```

Figure 4.8: SystemC Simulation Semantics.

4.3.2 Constructing Abstract Semantics

Abstraction can be applied to the memory, the environment and to the code itself. Since the environment is known statically at each program point, we can use the concrete program environment which is generated during the compilation phase. So, it is seen as a function that associates with every variable a list of abstracted pointers referring to some locations in the stack.

Memory Abstraction

Following the memory concrete semantics, the memory abstract semantics includes the allocation blocks and the stack.

Figure 4.9 shows the abstraction of an allocation block represented by a list of abstracted bytes. The concretization function $\gamma_{ByteAbs}$ generates for every block an integer b_i that can vary in the range 0 to $2^8 - 1 = 255$.

Three possible cases can describe how the data is organized inside the memory:

$a \textcircled{\&} b$: both a and b are in the same memory block, they do not overlap and their relative positions are unknown.

$a \textcircled{|} b$: a comes before b in the memory (some other blocks can be in between: case of structures for example).

$a \textcircled{\cdot} b$: a comes just before b in the memory (case of tables for example).

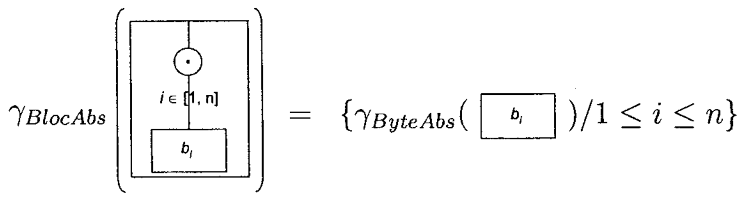


Figure 4.9: SystemC Example.

N.B.: $\gamma_{ByteAbs}(\boxed{b_i}) = \{(1, b_i)\}$

The abstraction of the memory includes also the definition of some operations over the defined graphical entities. The basic operations are: extraction of sub-blocks, conversion and determination of the size of a block.

The stack is represented as a sequence of allocation blocks. The access to the stack is always done using the address parameter. Two distribution points are defined to separate the blocks:

$\textcircled{[]}$: separates local variables for a block.

$\textcircled{()}$: separates function arguments and their local variables.

Code Abstraction

It is possible to abstract the code as a list of program points of the compiled code. However, without the abstraction of the code, it is quite impossible to reduce local variables (for example, unused variables). The code is seen as a list of instructions, where every two instructions are separated by a program point.

For every expression an abstract code is defined. To make the required transformation in the stack and the memory any time a program point is executed.

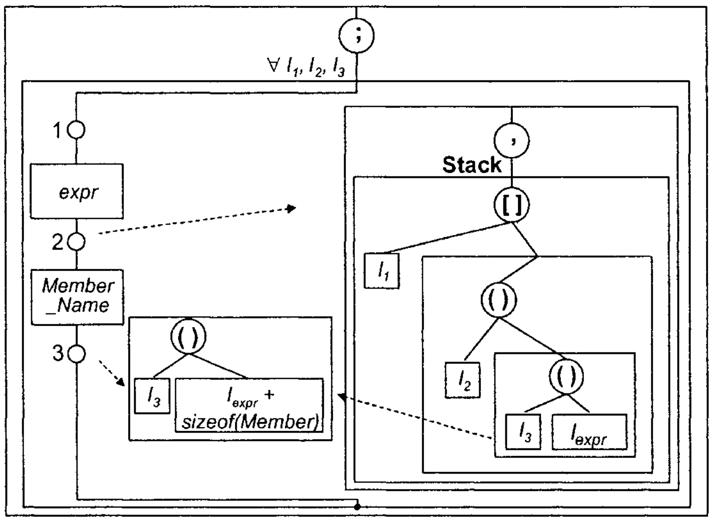


Figure 4.10: Class Members Abstraction.

Figure 4.10 shows the general case of a class members abstraction, where l_1 , l_2 and l_3 respectively represent a function call, a local variable and an expression local to an instruction. The member name is represented in the stack by a pointer to its relative address in the object. Considering l_{expr} as the address of the object *Expr* in the stack, the address of the member name is then: $l_{expr} + sizeof(Member)$.

Instructions abstraction concerns the basic instructions having the general format '*Expr;*', conditional instructions and declarations. The general effect of an instruction is shown in Figure 4.11(a), where the instruction acts as a memory transformer. A concrete example is given in Figure 4.11(b) for the conditional instruction: $if\ Expr\ then\ Instr_1\ else\ Instr_2$

SystemC Elements Abstraction

Figure 4.12 illustrates the abstraction of SystemC elements by the base module, where *Classname* refers to the module name, $string_{name}$ is the name of the module and (arg_1, arg_2, ... , arg_n) are the arguments of the module.

SystemC Simulation Manager Abstraction

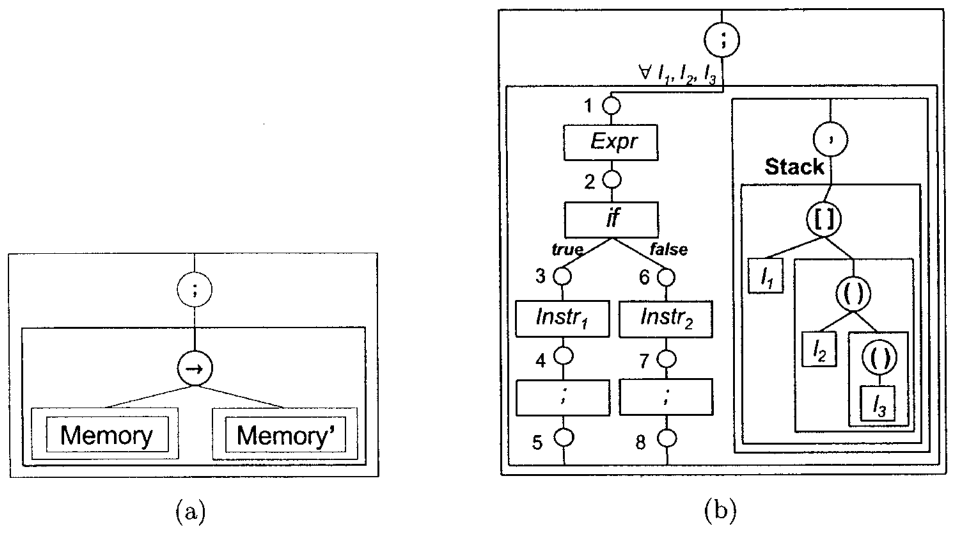


Figure 4.11: Instruction Abstraction.

The way events are organized can be summarized in three possible cases:

$e_1 \textcircled{\&} e_2$: both e_1 and e_2 will be executed in the same simulation cycle, however, their relative occurrence is unknown.

$e_1 \textcircled{|} e_2$: e_1 is always executed before e_2.

$e_1 \odot e_2$: e_1 is executed then e_2 is executed (no other process execution in between).

The events stack is represented as a sequence of events. The access to the stack is always done using the event identifier. We define two distribution points are defined to separate the blocks in the stack:

$\textcircled{[\,]}$: separates events general to the whole program.

$\textcircled{(\,)}$: separates events local to a module.

The main set of operations to manipulate the events stack is: adding a new event to the stack, removing an event from the stack and reading the state of an event considering its identifier.

The abstraction of the simulation environment of SystemC adds a new abstract layer over the default C++ abstraction. Figure 4.13 shows a general representation of the abstract environment (including the abstraction of the simulation manager).

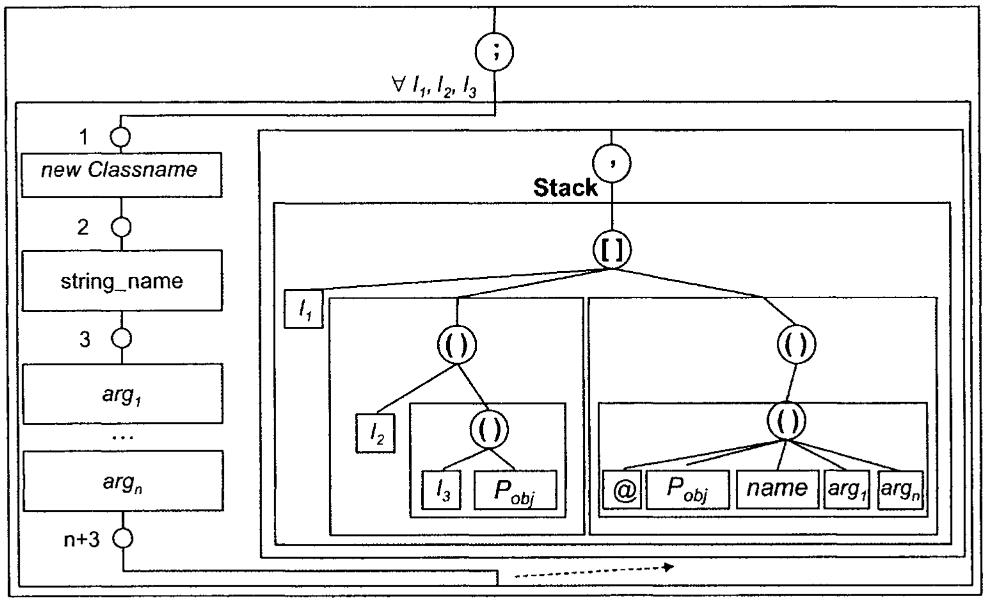


Figure 4.12: SystemC Base Module Abstraction.

The program is no more seen as a single copy performing updates over the memory at every program point. But, the abstracted program is represented by a number of entities (for instance, processes and threads) that are either active or inactive for a specific abstract cycle (which corresponds to a simulation cycle). In addition to that, the whole program is in the same time interacting with the program environment and the stack and dealing with the SystemC's event's stack and event's manager.

4.3.3 Analysis Tactics

The computation technique we propose combines ideas from both [14] and [19] and is based on the same principles as for the denotational semantics. The stack is supposed to be infinite. However, when the stack becomes full an exception will be processed. The general structure of the computation technique can be described by the following steps:

- Construct the hypergraph from the code fragments.
- Decorate the hypergraph by the abstract memory related information.

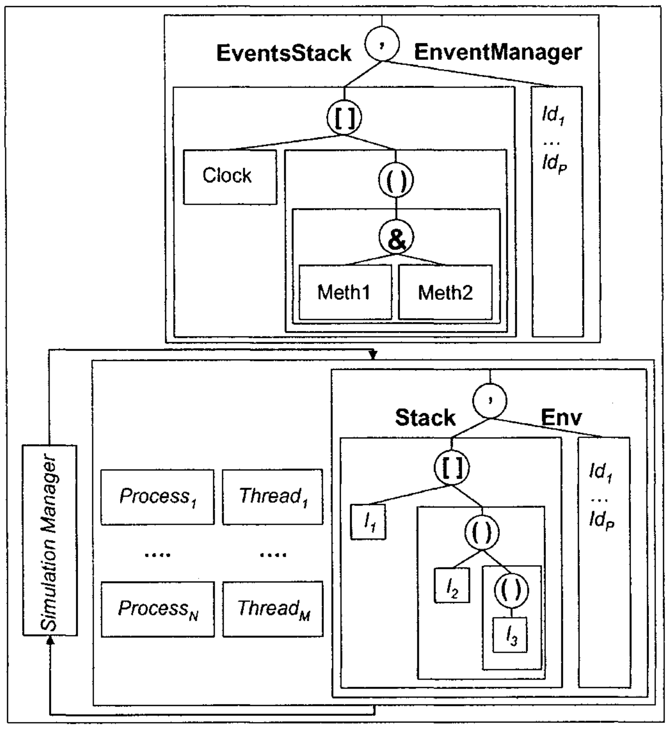


Figure 4.13: General Structure of the Abstract Environment.

- Any time a distribution point is found, an iteration counter is introduced. The analyzer will try to predict in the next steps how to relate the next state of the hypergraph to the iterations counter.
- After few duplications a merge tactic is applied. The analyzer will apply the merge after four to five iterations depending on the SystemC program.
- After few more iterations a new merge is applied but this time based on the iterations counter.
- The analyzer, at this level, must determine formally the shape of the code for the next iteration.
- Decompose the iterations counter in a union of a number of disjoint intervals.
- Given the initialization condition, we increment directly the iterations counter to the value of the actual interval targeting to get to the last state.

- Repeat previous tasks for every abstract simulation cycle in order to relate the simulation state to the cycles iterations counter.

To get more details about this computation approach and the abstract environment construction, we will consider in the next Section a small illustrative example.

4.3.4 Illustrative Example

We consider the relatively simple SystemC design given in Figure 4.14. It is composed from a random number generator, a transformation unit and a display. The transformation unit receives at every clock cycle a random input uniformly distributed in the interval [0,1] and outputs the summation *sum* of the last 1000 inputs (every 1000 cycles). The display shows *sum* every time a new value is available. The function of the transformation unit can be described by:

$$sum = \sum_{i=0}^{999} in_i, \text{ where } in_i \text{ specifies the input at the clock cycle } i + k/1000 \qquad (4.1)$$

(k is the actual cycle and $k/1000$ is an integer division).

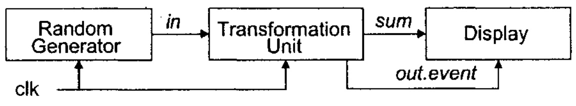


Figure 4.14: SystemC Example.

Each block of the design includes only one method:

- *U.Transform()*: responsible for performing the transformation on the input signal. This method is active on a clock event.
- *N.Generate()*: generates a random number. This method is active on a clock event.
- *D.Display()*: displays the number at its input (activated by the arrival of data).

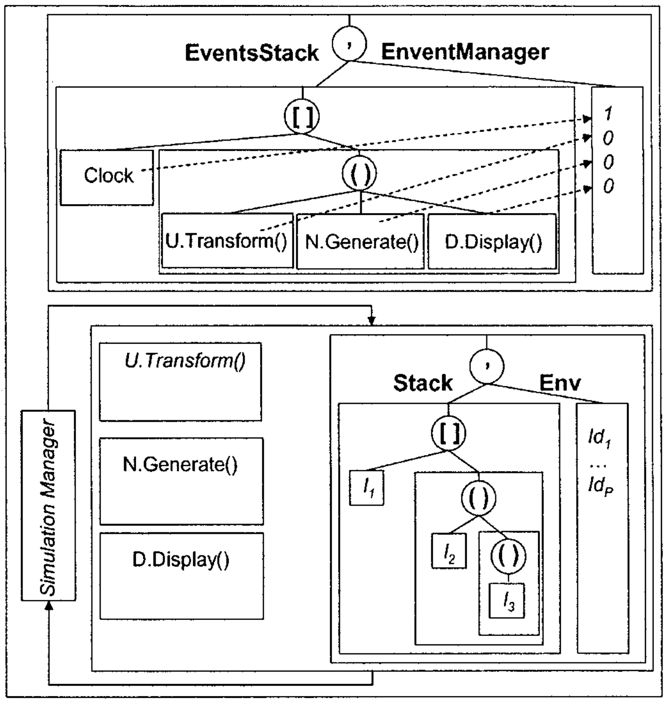


Figure 4.15: Initial Hypergraph.

Figure 4.15 shows the first hypergraph that is generated including four processes (the three methods and the clock process). Only the clock process is active at time $t = 0$.

The stack is empty and the program environment contains the list of the identifiers (without an explicit link with the program variables).

Figure 4.16 shows the new hypergraph after one iteration where the methods *N.Generate()* and *U.Transform()* are active and will be executed. The program environment is updated with the new variables (*in*, *sum*, T, N, D, a and N) coming from *sc_main*(). At the same time, the abstracted SystemC environment will be updated by the simulation manager. To get the hypergraph representation more compact and readable, we only included the new objects as pointers (P_N, P_T and P_D are pointers to the Number Generator (N), the Transformation Unit (T) and to the Display (D) respectively).

Figure 4.17 displays more details about the transformation method U.Transform().

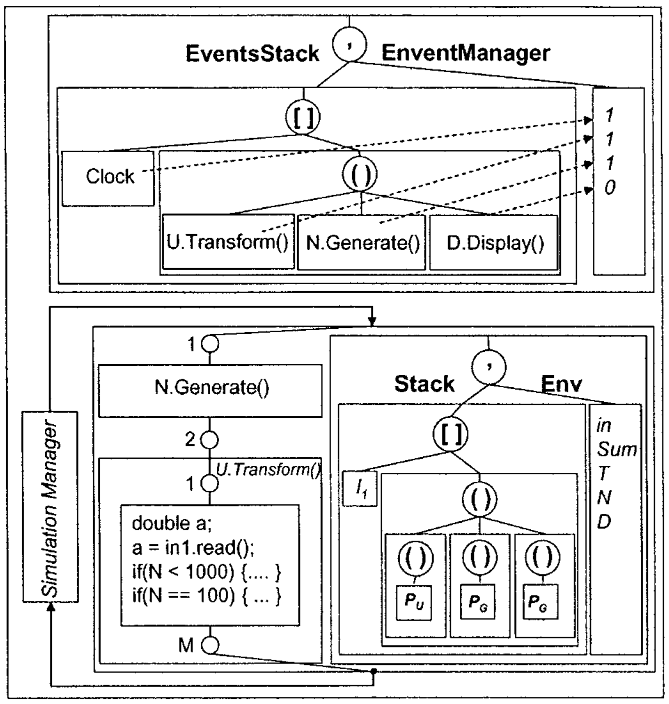


Figure 4.16: Iteration 1: SystemC Transformation Unit Hypergraph.

This compact representation will serve for iterations over this method. Then, after the link between the new copy of this method and the iterations counter is defined, we relate, using the reduction tactics, the new copy of the method to the cycles counter. We will proceed first by small step for the first iterations, then we move to big step for the last iteration. For instance, we can jump to the condition on the output $N = 1000$ because the condition on the output event to be triggered is activate by $N = 1000$.

After the iterations over the cycles counter are completed, the reduced version of the hypergraph is the one given in Figure 4.18. In this example, the simulation manager is reduced to an infinite while loop and an iterations counter (iter). The events are represented by if conditions and the whole system is reduced to a single procedure. Such a representation will eventually offer more information about the system in terms of order of execution of methods, relations between the program variables, the state of system at every iteration, etc.

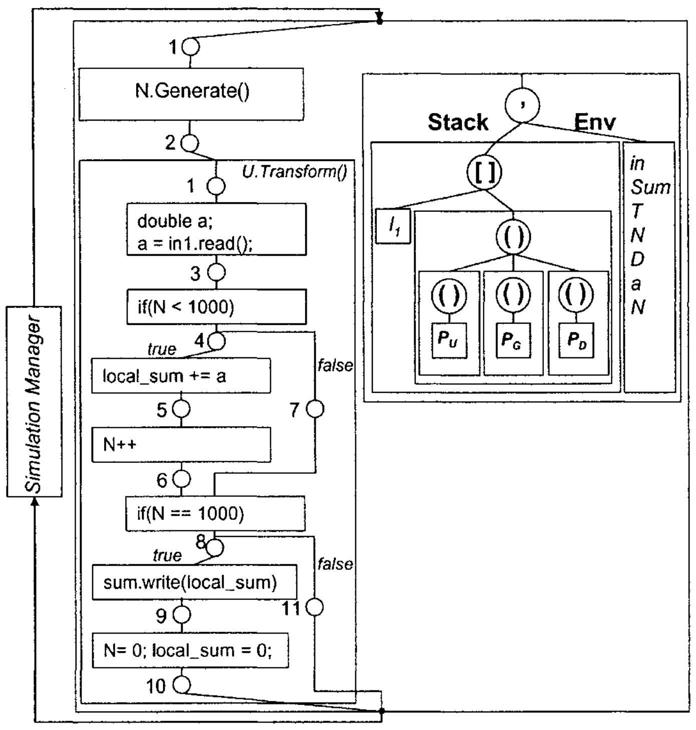


Figure 4.17: Hypergraph of the Active Methods.

4.4 Application: Packet Switch

In this Section, we show a more complex example of a 4x4 multi-cast packet switch taken from the SystemC library [80]. The switch uses a self routing ring of shift registers to transfer cells from one port to another in a pipelined fashion, resolving output contention and efficiently handling multi-cast cells. Input and output ports have FIFO buffers of depth four each [1].

Each input port is connected to a sender process. Each output port is connected to a receiver process. The sender and receiver processes are given distinguish *id* numbers during instantiations.

A sender process, writes a random value to data, and sends it to one or more of the four receivers. Each packet also contains a sender *id* field. Sender processes send packets at random intervals, varying from 1 to 4 units of its clock. A receiver

[1]The description of the switch is provided in Appendix C.1

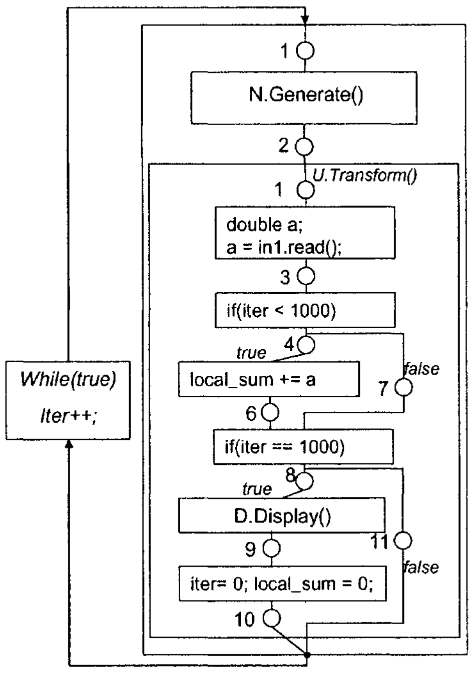


Figure 4.18: Final Reduced Hypergraph.

process is activated whenever a packet arrives. Then, it displays the content of the packet and the receiver *id*.

Figure 4.19 shows the first hypergraph generated from the packet switch code. It includes twelve processes: four senders, four receivers, two clock processes (first clock used for input and output operation and second clock used as internal switch clock), a process for the internal clock of the switch and process for the switch core itself. Only the clocks *clock*1 and *clock*2 are active when the switch starts. The other processes are activated on the reception of a packet or after sending a packet.

In parallel with the program environment, the events environment includes the list of all the system processes and their status. For simplification, we use only two status for each process: active (1) and not-active (0).

The simulation manager is presented as a box connected to the entries of the program hypergraph. It can be seen as a procedure that determines the structure of the system according to the list of active processes. For example, if the senders 1

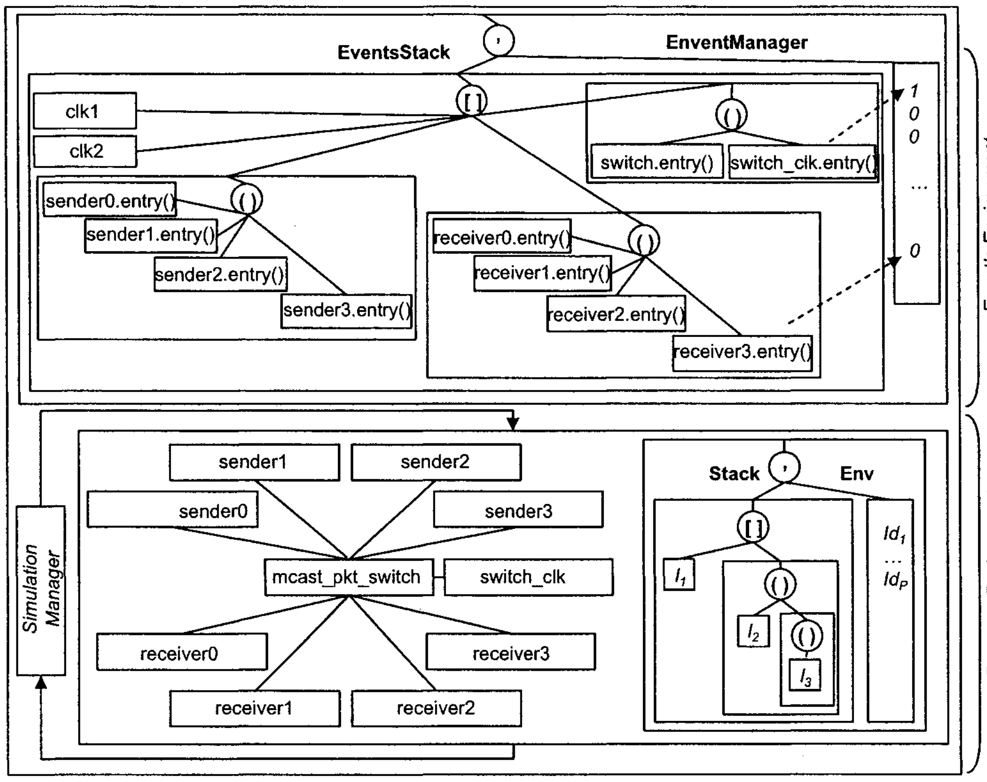
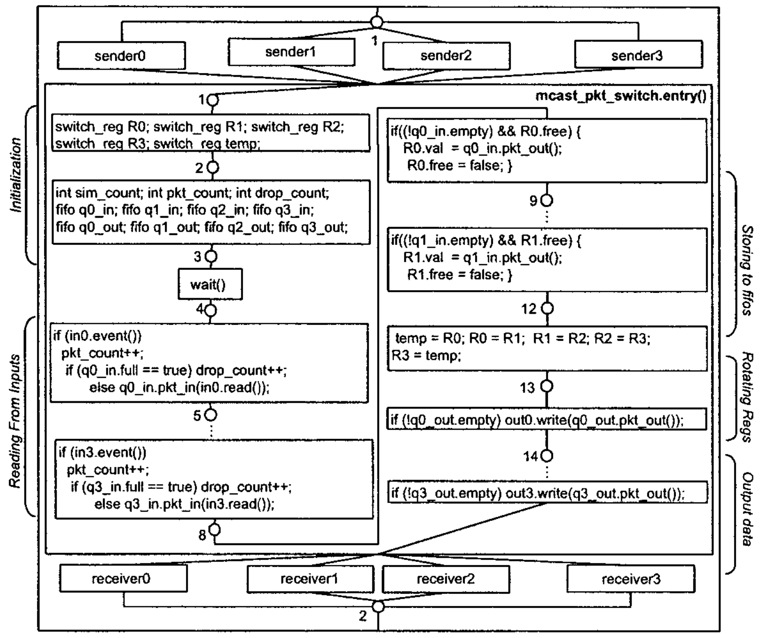


Figure 4.19: Packet Switch: Initial Hypergraph.

and 3 are active, then, only their relative code is analyzed. The general case when all the processes are active is presented in Figure 4.19. Each small box from the program environment, (e.g., sender0) presents a confined hypergraph that includes the correspondent object members and methods. For clarity, we will present the object hypergraph as its main method.

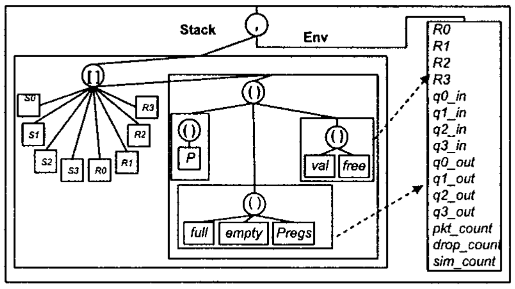
Note also that Figure 4.19 gives only an overview of the program environment. The variables l_1, l_2 and l_3 represents the general global and local variables. The status of the memory is not yet defined. In fact, during the analysis, when a new variable is analyzed, it will be added to the environment and will have its allocation space in the stack.

In this application we focus on the switch core itself. Figure 4.20-(a) presents the hypergraph of the switch entry method (main method). Figure 4.20-(b) shows an



```
switch_reg R0; switch_reg R1; switch_reg R2;
switch_reg R3; switch_reg temp;

int sim_count; int pkt_count; int drop_count;
fifo q0_in; fifo q1_in; fifo q2_in; fifo q3_in;
fifo q0_out; fifo q1_out; fifo q2_out; fifo q3_out;

wait()

if (in0.event())
  pkt_count++;
   if (q0_in.full == true) drop_count++;
        else q0_in.pkt_in(in0.read());

if (in3.event())
  pkt_count++;
   if (q3_in.full == true) drop_count++;
        else q3_in.pkt_in(in3.read());

if((!q0_in.empty) && R0.free) {
   R0.val  = q0_in.pkt_out();
    R0.free = false; }

if((!q1_in.empty) && R1.free) {
   R1.val  = q1_in.pkt_out();
    R1.free = false; }

 temp = R0; R0 = R1;  R1 = R2; R2 = R3;
R3 = temp;

if (!q0_out.empty) out0.write(q0_out.pkt_out());

if (!q3_out.empty) out3.write(q3_out.pkt_out());
```

(a)



(b)

Figure 4.20: Hypergraph of the Packet Switch Main Method.

overview of the state of the program environment and the stack. The stack includes global pointers to the senders and receiver objects. We detailed in particular the switch entry method parameters. This includes the local variables and the connection signals.

The analysis phase relates the elements of the previous hypergraph to a general iterators representing the simulation cycle. In other terms, we target to replace the whole SystemC simulator by a number of loops and iterators that define statically the order of execution of the processes.

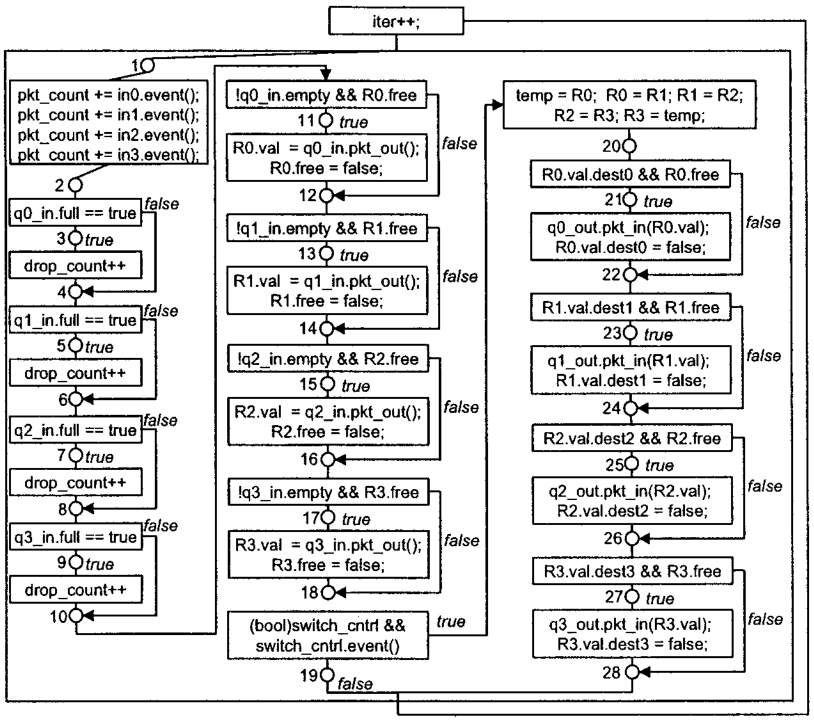


Figure 4.21: Reduced Hypergraph of the Switch Main Method.

By applying reduction (also called degradation) techniques on the basic hypergraph of the switch core we obtain the reduced version given in Figure 4.21. We notice that the SystemC simulator is reduced into a while loop and that most of the internal variables of the switch are defined as functions of the loop iterators. The switch main core is clearly divided into 3 main sections: reception and storage of the packets in the FIFOs, rotation of the data in the registers ring and output the packets to the receivers.

From the reduced hypergraph structure, a number of properties can be deduced. For example,

- The number of received packets is defined as *pkt_count*:

$$pkt\_count = \sum_{i=0}^{itr} \{in0.event()_i + in1.event()_i + in2.event()_i + in3.event()_i\}$$

where: $inX.event()_i$ is a Boolean flag (0 for false and 1 for true) set to true when a packet is received from the sender X.

- The number of dropped packet at the input FIFOs is defined by *drop_count*:

$$drop\_count = \sum_{i=0}^{itr}\{q0\_in.full()_i + q1\_in.full()_i + q2\_in.full()_i + q3\_in.full()_i\}$$

  where: $qX\_in.full()_i$ is a Boolean flag set to true when the input FIFO X is full.

- The number of packet received by the receiver X is $CountReceiver_X$:

$$CountReceiver_X = \sum_{i=0}^{itr}\{(RX.val.destX)_i \times (1-(RX.free)_i) \times (qX\_out.full)_i)\}$$

  where: $(RX.val.destX)_i$ is a Boolean flag set to true when the header destination flag of the packet contained in the register X is to be sent to the receiver X.

Although the previous properties may seem to be general, they can offer very precious information about the internal way the switch is working. In the same time they allow to detect behavioral errors. For example, the second property says that: "the count of dropped packets is the number of times the input queue (FIFO) is full", which is *not correct.* In fact, the correct property must set that: "the count of dropped packets is the number of *of received packet at the entry of the FIFO* when the input queue (FIFO) is full". In other words, we have to receive a packet when the FIFO is full to say that the packet was dropped. So, the condition to count the dropped packets must be changed from:

```
    if (q0_in.full == true)
        drop_count++;
```

to:

```
        if ((q0_in.full == true) && in0.event())
            drop_count++;
```

Note also that the properties obtained from the analysis phase can be used to validate properties related to the switch such as the maximum number of dropped packets. Also, according to the reduced hypergraph, the switch core only uses the packet's header to process the packet. Therefore, we can reduce the packet into its header (4 bits destination and 4 bits identifier) which may facilitate the use of model checking techniques to verify some particular properties of the switch.

This application can eventually give a general idea about our approach, nevertheless, it does not give a concrete evaluation of its performances. Trivially, real SoC designs are very complex systems; therefore, studying more complex systems is mandatory.

4.5 Summary

In this chapter, we presented a static code analysis framework for SystemC. We defined the basic concepts to construct both collecting and abstract semantics for OO languages. Then, we presented a graphical environment (extending an environment originally introduced by Vederine [105]) to represent and manipulate abstracted SystemC models. Finally, we illustrated the feasibility of our approach on some of SystemC basic designs.

Chapter 5

Model Checking

5.1 Introduction

In this Chapter, we will first illustrate the incapacity of classical model checkers to handle SystemC designs. For this purpose will conduct a *direct* model checking approach by translating the SystemC code to a language supported by existing commercial model checkers (Verilog for the case of FormalCheck [11]). Then, we propose an efficient model checking technique based on a state exploration algorithm provided for the AsmL language [51].

5.2 Direct Approach

We use the verification approach given in Figure 5.1, where the static code analyzer gets as input a SystemC design and a set of reduction tactics (called abstraction library). It then generates a reduced hypergraph representation of the design. This latter is fed into a hypergraph to Verilog converter. The conversion is seen as a concretization of the abstracted design (hypergraph) into the Verilog language. We did select Verilog because we will use the FormalCheck model checking tool [11].

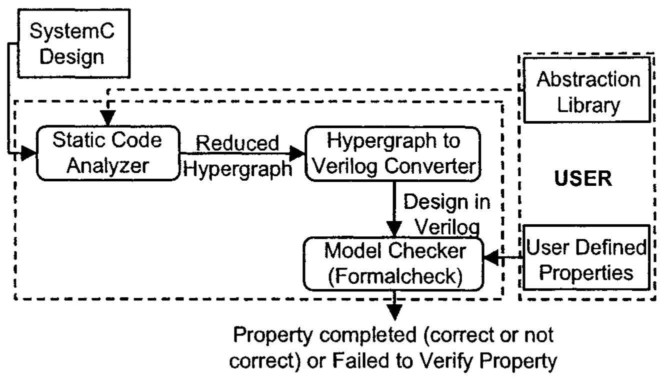

SystemC Design
Abstraction Library
Static Code Analyzer
Reduced Hypergraph
Hypergraph to Verilog Converter
USER
Design in Verilog
Model Checker (Formalcheck)
User Defined Properties
Property completed (correct or not correct) or Failed to Verify Property

Figure 5.1: Cascading Abstract Interpretation with Model Checking

We will illustrate the previously described verification approach on a bus structure offered as part of the SystemC distribution [80] [1]. In fact, this structure includes several SystemC components and showed the principles of using SystemC at the transactional level. Besides some of the sample properties, e.g. liveness and safety, cannot be verified using simulation. They require the usage of formal techniques such as model checking.

5.2.1 Abstraction

A partial representation of the bus's hypergraph is given in Figure 5.2. It shows the first hypergraph generated from the bus code. It includes an events' environment containing several processes: masters, slaves, clocks, arbiter, etc. In parallel with the program environment, the events environment includes the list of all the system processes and their status. For simplification, we use only two statuses for each process: active (1) and not-active (0).

The simulation manager is presented as a box connected to the entries of the program hypergraph. It can be seen as a procedure that determines the structure of the system according to the list of active processes. For example, if the Master 1 is sending active, then only its correspondent code is analyzed. Each small box from the

[1]The description of the bus structure is provided in the Appendix C.2.

program environment (e.g., arbiter()) presents a confined hypergraph that includes the corresponding object members and methods.

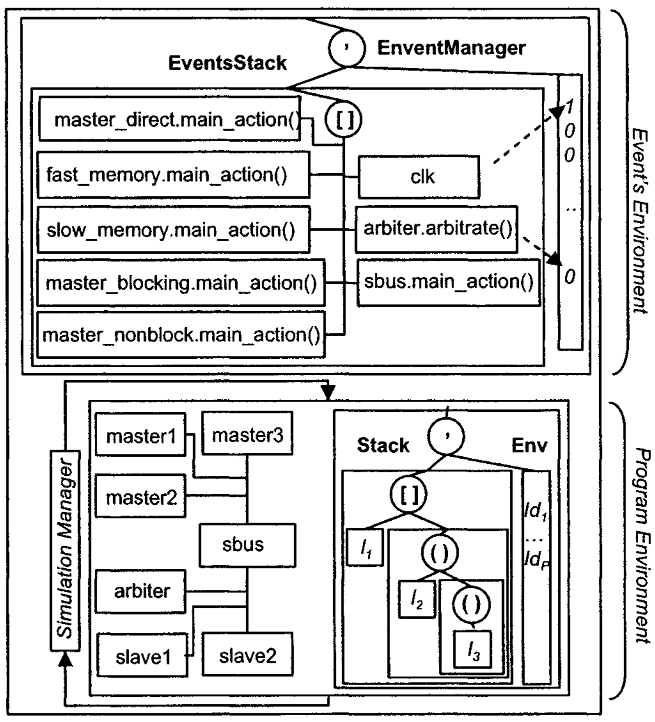


Figure 5.2: Hypergraph of the Simple Bus Structure.

5.2.2 Model Checking

After applying reductions tactics on the hypergraph of Figure 5.2, the generated reduced hypergraph is concretized into a Verilog code. This latter is fed into the FormalCheck tool [11] in order to verify some of the design's properties. In fact, FormalCheck verifies that a design model exhibits specific behaviors (*properties*) that are required by the design specification. Properties that form the basis of a model checker's query fall into two categories: *safety* and *liveness*. Safety properties can be expressed using one of two formats: The *always* format and the *never* format. Liveness properties describe behaviors that are *eventually* exhibited.

For instance we considered the following properties:

```
Property 1:
  NEVER( (simple_bus.request==ture)
      &&(simple_bus.status!=BUS_OK))
Property 2:
  AFTER (simple_bus.request==true)
  && (simple_bus.request.block==true)
  EVENTUALLY (simple_bus.status==BUS_BLOCK)
Property 3:
   EVENTUALLY (simple_bus.status==BUS_OK)
```

Property 1 means that a master generates a request only when the bus is ready to handle new requests (i.e. bus status set to BUS_OK). Property 2 says that if the bus receives a new blocking request, then, in the future, its status will change to blocking (i.e. bus status set to BUS_BLOCK). Property 3 proves the bus status will always return to ready to receive new requests.

5.2.3 Results and Discussion

We first started by translating the bus code from SystemC to Verilog (without abstraction). We modelled the SystemC simulator as a new module. Although this simplification reduces effectively the complexity of the code, the verification of all the previous properties failed after few minutes with the same problem of "memory exceeded". Then, when using the abstracted code all the properties were verified as it can be seen in Table 5.1. The verification platform is described in Table 5.2.

Even though both the design and the considered properties were quite simple, the results of Table 5.1 illustrate the complexity of model checking SystemC designs. Furthermore, existing model checkers are restricted to RTL designs which restricts them from handling transactional SystemC models or general C++ code that could

Table 5.1: Model Checking Results.

Property	CPU Time (hh:mm:ss)	Memory (in MB)
P1	6:59:12	93.59
P2	15:23:02	183.91
P3	17:46:54	293.63

Table 5.2: Verification Platform.

FormalCheck version	3.2
Main Memory	4.0 GB
CPU	2 CPUs (Run 900MHz)
Architecture	Sparc
OS Version	5.8

be integrated as part of the SystemC design. Hence, providing a new approach to apply model checking to SystemC is a non-avoidable task.

5.3 Related Work

As a related work, we cite, in particular, the Bandera [33] project that aims at interfacing Java code to model checking tools like SMV [17] and SPIN [53] by applying program analysis, abstraction, and transformation techniques. In its actual status, Bandera cannot handle SystemC designs because any analysis of a SystemC code must go through the whole simulation environment as well as SystemC defined data-types and classes. Besides, considering SMV as internal model checking tool is a big handicap for Bandera to handle large state space systems. We are not aware of any related work using a sound syntactical transformation from SystemC to AsmL and vice-versa to perform either model checking or ABV.

5.4 AsmL based Approach

We propose to combine the concepts of abstract interpretation and hyperstate in order to be able to treat complex SystemC designs and fit them to the verification process. Figure 5.3 shows the flow of our approach, where we start by a SystemC design, apply abstract interpretation and generate its hypergraphs, we then translate the events and processes based hypergraphs into ASM based on our embedding for SystemC in ASM. In parallel, we embed PSL properties in the ASM model [2]. We then compile the ASM model, including both the design and the properties, using the AsmL tool and generate its FSM. This FSM is translated into the input language of the model checking tool, which will check the correctness of the model. Similarly, the AsmL compiler can generate test scenarios, .NET, or C# models for verification by simulation.

The generation of the FSM from ASM is performed using the algorithm given in [51]. Unfortunately, the algorithm that generates the FSM is not available as the AsmL tool is provided as a black–box. To solve this problem, we embed the state of every property (as Boolean) in every system's state. Therefore, once the FSM is generated, it will include, by construction, a Boolean state variable giving the state of the property. The last step in the verification process is to translate the FSM to a format supported by a model checker. Note that there is no restriction on the model checker as the final FSM is concrete and includes only Boolean variables to represent the state of the PSL properties.

5.4.1 Model Checking Technique

To enable the integration of both the model and the properties at the ASM level, we embedded the PSL semantics in AsmL. At this level, it is possible to verify these properties using model checking. For instance, we encode the properties evaluation

[2]The details of the PSL embedding in AsmL are provided in the Appendix B.

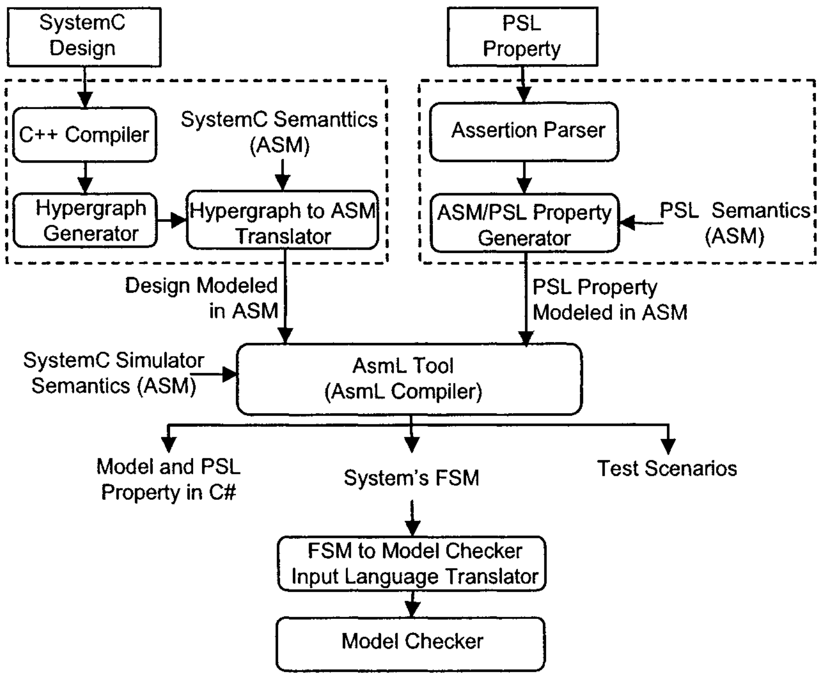

SystemC Design
C++ Compiler
SystemC Semanttics (ASM)
Hypergraph Generator
Hypergraph to ASM Translator
PSL Property
Assertion Parser
ASM/PSL Property Generator
PSL Semantics (ASM)
Design Modeled in ASM
PSL Property Modeled in ASM
SystemC Simulator Semantics (ASM)
AsmL Tool (AsmL Compiler)
Model and PSL Property in C#
System's FSM
Test Scenarios
FSM to Model Checker Input Language Translator
Model Checker

Figure 5.3: Verification of PSL for SystemC Designs in ASM.

in every state, which enables checking its correctness on-the-fly while executing the FSM generation algorithm (part of the AsmL tool). An incorrect property detection stops the reachability algorithms and outputs a sub-portion from the complete FSM, which represents a complete scenario for a counter-example.

Every property is embedded in every state in the FSM generated by the AsmL tool and is represented by two Boolean state variables P_{eval} and P_{value} (stating, respectively, if the property can be evaluated and the value of the property in the current state). A violated property is detected once $P_{eval} = true$ and $P_{value} = false$. The previous condition is a filter for the FSM generation algorithm stopping the generation when an error is detected. In this case, the generated portion of the state machine can be used to identify the problem through a scenario of a counter-example. For multiple properties, the filter is set as conjunction of all the conditions for the separate properties. This technique minimizes radically the number of the state variables (the FSM size and its generation time). A correct verification process results on the

generation of the system's FSM (according to the configuration file constraints). This approach may seem to be based on an ad-hoc model checking algorithm while more advanced techniques and approaches have been used in tools like SMV and VIS. We believe there are many reasons that make our approach more efficient, in particular:
(1) It is impossible to use these tools with AsmL considering the OO nature of the language. Therefore, a translation to the language supported by the tool (mostly a very low HDL) is mandatory. This operation will prohibit using some advanced features AsmL offers (e.g., data abstraction, pre-condition checking, etc.)
(2) Generating the counter-example as an FSM gives a strict and complete path of the error starting from the entry point to the state where the error took place [47].
(3) The configuration of the FSM generation algorithm can be set by the user in order to stress the verification only in some particular portions of the state space (through restricting some variables to have certain range for example) [47].

5.4.2 Illustrative Example: Packet Switch

In this section, we apply the above approach on a packet switch design from the SystemC library [80] [3].

For illustration purposes, we consider the following three PSL properties for the packet switch.

The first property, $P1$, is intended to verify that if there is only one recipient for the packet, and the output queue is not full, then the register that holds the packet should be free in the next internal clock, and the packet should be received at the output queue.

```
$Property P1:$
    $forall\ send\ in\ \{0,\ 1,\ 2,\ 3\}$
      $if\ Reg[send].free\ ==\ true\ and$
        $Packet.dest0\ and\ not\ OutQueue[send].full\ and$
```

[3]The description of the packet switch structure is provided in the Appendix C.1.

```
    $not\ Packet.dest1\ or\ Packet.dest2\ or\ Packet.dest3$
     $then\ at\ next\ SWCLK:$
      $Reg[send].free\ =\ true$
      $OutQueue[0]\ =\ Reg[send]$
```

The second property, $P2$, is intended to check the shortest path when sending from sender i to receiver i, where the input queue is not full. This operation should be performed in four internal clocks ($SWCLK$) or equivalently one external clock (CLK).

$PropertyP2:$

```
  $forall\ send\ in\ \{0,\ 1,\ 2,\ 3\},\ forall\ rec\ in\ \{0,\ 1,\ 2,\ 3\}$
    $if\ send\ ==\ rec\ and\ not\ InQueue[send].full\ and$
      $Reg[send].free\ ==\ true\ and$
      $OutQueue[rec].empty\ ==\ true\ then$
       $OutQueue[rec] = Reg[send]\ in\ 4\ SWCLK$
       $and\ OutQueue[rec] = Reg[send]\ in\ 1\ CLK$
```

The third property, $P3$, is intended to check the (worst) longest path when sender 0 transmits to receiver 3, input queue 3 has only one free slot, all other input queues are full, and the output queue 0 is full. This property is specified as follows:

$PropertyP3:$

```
  $ifsend\ ==\ 3\ and\ rec\ ==\ 0$
    $and\ InQueue[3].size\ ==\ 3$
    $and\ InQueue[0].full\ and\ InQueue[1].full$
    $and\ InQueue[2].full\ and\ OutQueue[0].full\ then$
      $OutQueue[0]\ ==\ Reg[3]\ in\ 8\ to\ 19\ SWCLK$
```

The AsmL tool is used in order to generate automatically the FSM of the packet switch and having the evaluation of the properties as a state variable. The model checking resulted in successfully verifying the correctness of properties $P2$ and $P3$. Property $P1$, however, was violated, indicating a bug in the SystemC packet switch model. This bug showed, after further inspection of the code, that the switch will free any packet coming from senders 0, 2 and 3 and having at least two destinations including port 1 before routing it to output port (different from port 1). The erroneous code is the following:

```
if (R1.val.dest1||R1.val.dest1||R1.val.dest2||R1.val.dest3)
    R1.free = true;
```

where the condition to free the register does not check if the packet is having as destination the port 0 and uses a double copy of the check about the port 1 ($R1.val.dest1$). The correct condition should be:

```
if (R1.val.dest0||R1.val.dest1||R1.val.dest2||R1.val.dest3).
```

5.5 Application: AGP Bus

We considered the AGP bus [4] [58] that was, as far as we know, only verified by simulation due to its complexity and very large state space. We will show that our technique combined with the abstraction features of AsmL and using an inductive proof, allows the model checking of a set of properties on the bus. These properties are also translated to a SystemC monitor that can be used as a separate IP to validate AGP compatible devices.

5.5.1 AGP Bus Properties Specification

In order to verify the bus properties, we first used a direct model checking approach by considering a set of properties to verify all the possible transactions scenarios. These cover two main classes: (1) PCI transactions and (2) AGP transactions including both

[4]A detailed description of the bus is provided in Appendix C.4

modes DMA and execute. We succeeded to prove the first class of properties with the approach of 5.4, while we failed to prove the second set due to state explosion. Therefore, we introduce a proof by induction.

Example properties are giving below:

Property P_1 :

```
    $forall\ Master\ in\ \{Master0,\ \ldots,\ Master4\}$
      $if\ (!Master.REQ\ ==\ true)\ then$
        $eventually\ (!Master.GNT\ ==\ true)$
```

meaning that if a master requests the bus $(!Master.REQ\ ==\ true)$, it will get access to it in the future $(!Master.GNT\ ==\ true)$, which guarantees that no master will use the bus indefinitely.

Property P_2 :

```
    $forall\ Master\ in\ \{Master0,\ \ldots,\ Master4\}$
    $forall\ Slave\ in\ \{Slave0,\ \ldots\ Slave4\}$
      $if\ (!Master.GNT\ ==\ true)\ and$
            $(!Master.DEST\ ==\ Slave.ID)$
      $then\ \{eventually\ (!Bus.FRAME\ ==\ true)\ and$
            $(!Master.TRDY\ ==\ true)\ and$
            $(!Slave[ID].TRDY\ ==\ true)\ and$
            $(!Master.GNT\ ==\ false)\}$
```

meaning that if a master is selected by the arbiter, then it will be able to get access to the bus by setting $!Bus.FRAME$. Thereafter, its destination slave will be activated by setting its $!Slave.TRDY$. Finally, the master will release the bus once $!Master.GNT$ is set to false.

5.5.2 Model Checking

Performing the verification of the whole model failed to complete due to a state explosion problem. The main reason for that is the huge size of the read, write and commands queues (each of width 256) present in both the AGP device and the corelogic. By reducing the queues width to three, however, we succeeded to verify all the properties. To generalize the verification, though, we defined an induction based approach taking advantage from the abstract data types supported by ASM.

Global Proof

We define DRQ: Device Read Queue, DWQ: Device Write Queue, $DReQ$: Device Request Queue, CRQ: Controller Read Queue, CWQ: Controller Write Queue and $CReQ$: Controller Request Queue. The maximum width of the queues is $Q.Wd$. The number of packets in each queue is $XXQ.Np$ (where $XX \in \{DR, DW, DReq, CR, CW, CReq\}$). The list of properties under verification is P.

- Step 1: Verify $P = true$, $\forall$ $DRQ.Np$, $DWQ.Np$, $DReQ.Np$, $CRQ.Np$, $CWQ.Np$, $CReQ.Np \in [0, 1]$.
- Step 2:
  - Hypothesis: Consider $N \in \mathbb{N}$ / $0 < N < Q.Wd$
    $\forall x \in \{DRQ.Np, DWQ.Np, DReQ.Np, CRQ.Np, CWQ.Np, CReQ.Np\}$, $x < N$: P is true.
  - Prove: $\forall x \in \{DRQ.Np, DWQ.Np, DReQ.Np, CRQ.Np, CWQ.Np, CReQ.Np\}$, $x < N + 1$: P is true.

Single Queue Proof

Considering the condition in hypothesis of Step 2, in order to prove the correctness of P, we need to consider all the possible states when at least one of the queues contains N+1 packets: $\exists x \in \{DRQ.Np, DWQ.Np, DReQ.Np, CRQ.Np, CRQ.Np, CReQ.Np\}$, where $x < N + 1$. For instance, this is still a quite huge number of states. So, we did divide this proof into six sub-proofs each considering a particular queue. We will illustrate next the case of the *DRQ* (other cases are similar). The new target is to prove that considering the hypothesis of Step 2, we do have the following:

$\forall y \in \{DWQ.Np, DReQ.Np, CRQ.Np, CWQ.Np, CReQ.Np\}$, $y < N$ and $DRQ < N + 1 : P$ is true.

The proof is as follows:

- Step 1: Verify $P = true$, $\forall$ $DWQ.Np$, $DReQ.Np$, $CRQ.Np$, $CWQ.Np$, $CReQ.Np \in [0, 1]$ and $DRQ.Np \in [0, 2]$.

- Step 2:

  - Hypothesis: Consider $N \in \mathbb{N}$ / $1 < N < Q.Wd$ $\forall x \in \{DWQ.Np, DReQ.Np, CRQ.Np, CWQ.Np, CReQ.Np\}$, $x < N - 1$ and $DRQ.Np < N + 1 : P$ is true.

  - Prove: $\forall x \in \{DWQ.Np, DReQ.Np, CRQ.Np, CWQ.Np, CReQ.Np\}$, $x < N$ $DRQ.Np < N + 1 : P$ is true.

This latter proof requires generating all the system's states where $DWQ.Np = DReQ.Np = CRQ.Np = CWQ.Np = CReQ.Np = N - 1$ and $DRQ < N + 1$.

Note that the proof is valid for any value of $N > 1$ (in particular for the value of 256 proposed by Intel in [58]). For instance, we abstracted the queue width to the domain `QueueWidth` = {`Empty`, `NotEmpty`, `Full`}. The `NotEmpty` case represents a non-full queue with at least one element. The `QueueWidth` abstract domain contains a set of helper functions (e.g., `GetWidth()` and `UpdateWidth()`) to enable tracking the queue state changes. Such an abstraction was possible thanks to the data abstraction feature offered by AsmL [51] and the user-defined configuration of the FSM generation algorithm [47].

Table 5.3: Validity of Initialization Conditions.

Queue	CPU	Number of FSM	
width	Time (s)	Nodes	Transitions
1	5.78	34	37
2	30.89	173	193
3	105.20	504	563
6	1758.78	4325	5223

5.5.3 Experimental Results and Discussion

The CPU time used for the generation of the model checking for queues widths in {1,2,3,6} is given in Table 5.3 [5]. The first three rows are required to ensure the correctness of the initialization conditions. The fourth row, queue width equal to six, is given to illustrate the effect of the numbers of states and transitions increase exponentially as function of the queue size. This clearly illustrates the impossibility of generating the complete FSM for a width of 256.

In Table 5.4 every row corresponds to the proof of a particular queue. Generally, the CPU time and the number of Nodes and transitions is close to the case when the queue width is equal to three (see Table 5.3). This is quite expected because in our approach the queue may be in one of the three states: empty, has some elements but not full or full.

Table 5.5 presents the verification information for the PCI mode which is optional for AGP. A direct verification for that mode was possible thanks to the relative simplicity of the PCI, which does not include any queue structure.

[5]All experiments presented in this section were conducted on a platform consisting of a 2.4 GHz Pentium IV and 512 MB of RAM (PC2700).

Table 5.4: Model Checking (AGP Mode).

Proof for	CPU	Number of FSM	
the Queue	Time (s)	Nodes	Trans.
DRQ	341.01	1156	1304
DWQ	345.25	1294	1325
DReQ	347.78	1302	1346
CRQ	457.89	1503	1425
CWQ	462.07	1653	1433
CReQ	487.01	1859	1481

Table 5.5: Model Checking (PCI mode).

Number of		CPU	Number of FSM	
Masters	Slaves	Time (s)	Nodes	Transitions
1	1	2.31	20	25
3	1	26.01	236	341
2	2	26.84	293	449
3	2	574.18	1881	3153

5.6 Summary

In this Chapter we proposed a model checking technique using a state exploration algorithm provided for the AsmL language. We illustrated the feasibility of our approach on a packet switch design part of the SystemC library. Then, we showed that our technique combined with the abstraction features of AsmL allows, using an inductive proof, the model checking of complex systems. For instance, we have been able to verify a set of properties of the AGP bus standard that was never been verified using formal techniques due to its huge state space.

Chapter 6

Assertion Based Verification

6.1 Introduction

Model checking is not always feasible in particular when dealing with very complex designs where exploring the whole state space results in a state explosion problem. For this reason, assertion based verification techniques are more suitable as they turn the property under verification into a monitor, checked by simulation and evaluated using coverage metrics. In this Chapter, we present two approaches to add assertions to SystemC: (1) directly adding assertions as external modules to the system; and (2) automatically generating assertion monitors from PSL properties.

In [48] an approach is presented to add assertion checkers to SystemC. Our work is different mainly in two aspects: (1) The properties in [48] are restricted to the notation of property checker from Infineon Technologies AG then translated to synthesizable SystemC instructions while we consider any PSL property; and (2) SystemC is considered in [48] as a low level HDL while we do not put any restriction on any subset of SystemC.

6.2 Extending SystemC by SVA

6.2.1 System Verilog Assertions

The SystemVerilog standard is the result of an industry-wide effort to extend the Verilog language in a consistent way to include enhanced modeling and verification features. A key feature of SystemVerilog is the SystemVerilog Assertion (SVA) [2], which unifies simulation and formal verification semantics to drive the design for verification methodology.

The semantics of SVA are defined such that the evaluation of the assertions is guaranteed to be equivalent between simulation and formal verification. This equivalence ensures that multiple tools will interpret the behaviors specified in SVA in the same way. Moreover, the unification of assertions with the design and verification code streamlines the interaction between the assertion and the testbench to augment the power of assertions. In particular, SystemVerilog allows assertions to communicate information to the testbench and allows the testbench to react to the status of assertions without requiring a separate application programming interface (API).

SystemVerilog provides two types of assertions: *immediate* and *concurrent*.

Immediate Assertions

Immediate assertions are procedural statements that can occur anywhere within always or initial blocks, and include a conditional expression to be tested and a set of statements to be executed depending on the result of the expression evaluation. The syntax of an immediate assertion is:

$immediate\_assert\_statement$ ::=

$assert(expression)[[pass\_statement]else[fail\_statement]]$

The expression is evaluated immediately when the statement is executed, exactly as it would be for an if statement. The *pass_statement* is executed if the expression evaluates to true, otherwise the *fail_statement* is executed.

Concurrent Assertions

The real power of SVA, both for simulation and formal verification, is the ability to specify behavior over time, which VHDL assertions cannot do. In fact, concurrent assertions provide the ability to specify sequential behavior concisely and to evaluate that behavior at discrete points in time, usually clock ticks (such as "posedge clk"). The syntax of concurrent assertions is:

concurrent_assert_statement ::=
assert(sequential_expr_or_prop)[[pass_statement] else
[fail_statement]]

The concepts and components that make up concurrent assertions can best be understood as a set of layers, each building on the layer as described in Figure 6.1.

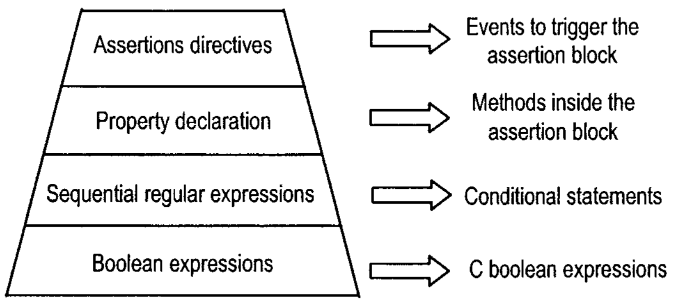


Figure 6.1: Mapping between SystemVerilog assertions and SystemC objects.

The basic function of an assertion is to specify a set of behaviors that is expected to hold true for a given design or component. The Boolean expressions layer is the most basic one, and specifies the values of elements at a particular point in time, while the sequential regular expressions layer builds on the Boolean layer to specify the temporal relationship between elements over a period of time. The property declarations layer builds on sequences to specify the actual behaviors of interest, and the assertion directives layer explicitly associates these behaviors with the design and guides verification tools about how to use them.

To ensure consistency between simulation and formal verification tools, which apply a cycle-based view of the design, concurrent assertions in SystemVerilog use sampled values of signals to evaluate expressions. The sampled value of signals is defined to be the value of the signal at the end (for instance, at read-only synchronization time as defined by the Programming Language Interface (PLI) [2]) of the last simulation time step before the clock occurs. This way, a predictable result can be obtained from the evaluation, regardless of the simulator's internal mechanism of ordering and evaluating events.

6.2.2 Extending SystemC by SVA

To add SVA to SystemC two options are possible: integrate the SVA as part of the library or on top of the library. The first case presents a radical change of SystemC requiring adding new constructors to the library (*assert* for example). Besides, the SystemC simulator and semantics must be updated in order to manage and verify correctly the assertions. This choice may seem to be the most efficient as assertions will be defined in SystemC the same way they are integrated in SystemVerilog. Nevertheless, considering the OO aspect of SystemC and its modular structure, it is easier yet probably more efficient to add assertions on top of SystemC. In fact, any assertion can be seen as a monitor having as input some of the design signals, performing a verification operation and giving as output a status flag. The open question facing this latter approach is how to update the design in order to connect the assertions monitors.

Figure 6.2 shows the proposed methodology to construct and integrate SVA into SystemC design. We first start by collecting the information about the environment from the SystemC compiled code. To do so we consider the symbol file generated from

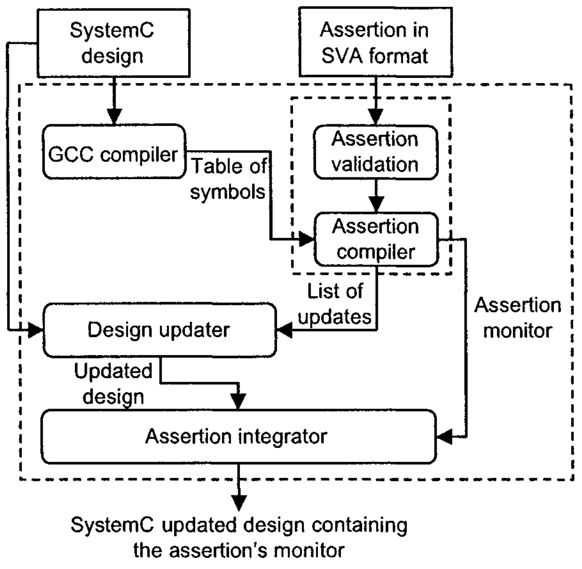


Figure 6.2: Extending SystemC by SVA.

the GNU-C-Compiler (GCC). This step is needed in order to localize which signal belongs to which module. Then, the assertion is validated and compiled. The validation phase verifies the syntax of the assertion while the compilation phase performs the link between the design variables and the assertion parameters.

In order to connect the assertion monitor to the design, this latter needs also to be updated. In fact, the signals involved in the assertion must be transformed to output signals in order to feed them to the assertion monitor. The list of signals required to extract from the design is generated by the assertion compiler and input to the design updater which performs the required modifications to the original SystemC design. These modification will not affect the behavior of the design since they will only get some signals connected to the assertion monitor as *read-only*.

The assertion module is then connected to the updated design. This module will be instantiated in the main function of the SystemC design (*sc_main*) and connected to the appropriate existent modules. The resulting code when executed will therefore consider the assertion monitor as part of the design.

The assertion compiler generates the SystemC code that corresponds to the input assertion which includes: Boolean expressions, sequential expressions and properties. We will consider Boolean variables as SystemC signals (*sc_signal*) in order to get benefit of the object nature of this module and to be able to integrate any variable as part of the monitor constructor section (containing the triggering conditions).

Sequential Expressions

SystemVerilog includes the ability to specify sequential expressions or sequences of Boolean expressions with temporal relationships between them. To determine a match of the sequence, the Boolean expressions are evaluated at each successive sample point, defined by a clock that gets associated with the sequence. If all expressions are true, then a match of the sequence occurs. The most basic sequential expression is something like *event1* followed by *event2* after three-clock cycles" which is represented in SystemVerilog syntax as: "*event1* # # 3 *event2*".

In this previous example, the "##3" indicates a three-clock delay between successive Boolean expressions in the sequence. Every sequence will be represented in SystemC as a list of members of the assertion monitor. The clock cycles will be represented as counters. The code corresponding to the previous example is given by:

```
sc_in<bool> event1; sc_in<bool> event2;
sc_in<int> counter = 0;
```

The *counter* variable is updated in the sequence validation method as follows:

```
 if(event1)
    counter = 1;
    if (counter > 0)
      counter++;
    if(counter == 3) {
      if(event2)
```

Table 6.1: SVA Sequence Operations.

Operation	Syntax	Meaning
Concatenation	seq1 ##1 seq2	seq2 begins on the clock after seq1 completes
Overlap	seq1 ##0 seq2	seq2 begins on the same clock seq1 completes
Ended Detection	seq1 ##1 seq2.ended	seq2 completes on the clock after seq1 completes (regardless when seq1 started)
Repetition	seq1 [*n:m]	repeat seq1 a minimum of n and maximum of m times.
First Match Detection	first_match(seq1)	if seq1 has multiple matches, consider the first one.
OR	seq1 or seq2	compound sequence that matches when seq1 or seq2 matches
End	seq1 and seq2	compound sequence that matches when both seq1 and seq2 match

```
  {
    counter = 0; return TRUE;
  }
  else
  {
    counter = 0; return ERROR;
  }
  return Pending;
}
```

The main operations defined over sequences are summarized in Table 6.1.

Property Declarations

The property layer allows for more general behaviors to be specified. In particular, properties allow users to invert the sense of a sequence (e.g., when the sequence should not happen), disable the sequence evaluation, or specify that a sequence be implied by some other occurrence. The property construct allows these capabilities using the following syntax:

property_declaration ::=
property_name[*formal_item*(, *formal_item*)];
    assertion varaibles declaration
    property_specification
endproperty

property_specfication ::=
property_name[*formal_item*(, *formal_item*)];
    [*clocking_events*]
    [*disable iff* (*expression*)] [*not*] *property_expr*

property_expr ::= *sequence_expr* | *implication_expr*

The important difference between sequences and properties resides in the fact that these latter are triggered by signals other than clocks. As a result, the representation of a property in the assertion monitor will contain two parts:

- Property verification method: A method that is responsible for the verification of the assertion.
- Triggering conditions: a list of conditions that activate the verification of the assertion (a signal update for example).

As an illustration consider the assertion: "as long as the *test* signal is low, check that the *abort_seq* sequence does not occur". This can be written in SVA as:

property p1 ;

@ (*posedge clk*) *disable iff* (*test*) *not abort_seq endproperty*

This assertion is represented in the SystemC assertion monitor as method triggered when the signal *test* is low and performing the following code:

```
if(test.in()) {
  if(abort_seq)
    return ERROR;
  else
    return TRUE;}
```

We note that in practice, most sequential assertions are expressed as some form of implication "when this happens, then that will happen", and thus require the assertion writer to specify the antecedent expression to trigger the assertion (for the previously given assertion the condition was *test.in()* set to TRUE). The object nature of the SystemC assertion monitor as a "*SystemC Module*" offers more flexibility to define this kind of assertions.

6.3 Extending SystemC by PSL Assertions

To support SystemC, PSL can be either integrated as part of SystemC, or put on top of the library. The first approach presents a radical change of SystemC requiring the addition of new constructors and functionalities to the library (like *assert* and *assume*). Besides, the SystemC simulator and semantics must be updated in order to manage, support and verify the assertions and their verification process. Although, this choice may seem to be very efficient, considering the object-oriented aspect of SystemC and its modular structure, it is easier, yet probably more efficient, to add assertions on top of SystemC. In fact, any assertion can be seen as a monitor keeping track of some of the design signals, performing a verification operation and giving a status flag as an output. The open question with the second approach is how to update the design in order to connect the assertion monitors.

The classical way to add PSL assertions to SystemC is to code them in C++. However, this option has many drawbacks especially that the C++ language is not adequate to write logical and sequential properties and formulas as defined in PSL. Besides, to make sure that the embedding of PSL in C++ is correct, we must put an important additional effort to validate the new assertion's layer. It will hence be more efficient to model the assertion in a language, like ASM, which offers two very important features: (1) it can model state machines, and (2) it can be translated to a C# code, which supports any other language in the .NET framework (in particular C++).

Figure 6.3 describes our approach to extend SystemC with PSL assertions, which consists of the following three main steps:

1. Updating the SystemC design to interface to the assertion monitor.

2. Generating the assertion as a C# code from its ASM description.

3. Integrating the C# assertion in the SystemC design.

Generating the table of symbols from the SystemC design is important in order to validate the variables (names and types) that are used in the assertion. In fact, while compiling the assertion, we are concerned with, first, its syntactical correctness, and second, its semantical validity. In this latter, we check the type and the naming of the assertion variables.

Once the assertion's structure verified, we translate it to its equivalent ASM code. In our embedding of the assertion in ASM, we defined a one to one mapping between the PSL assertion and their ASM embedding. Hence, the transformation is purely syntactical, which guarantees the correctness of the embedded assertion.

In the validation phase of the assertion structure, we also generate a list of updates required to prepare the design to integrate the assertion. For instance, the

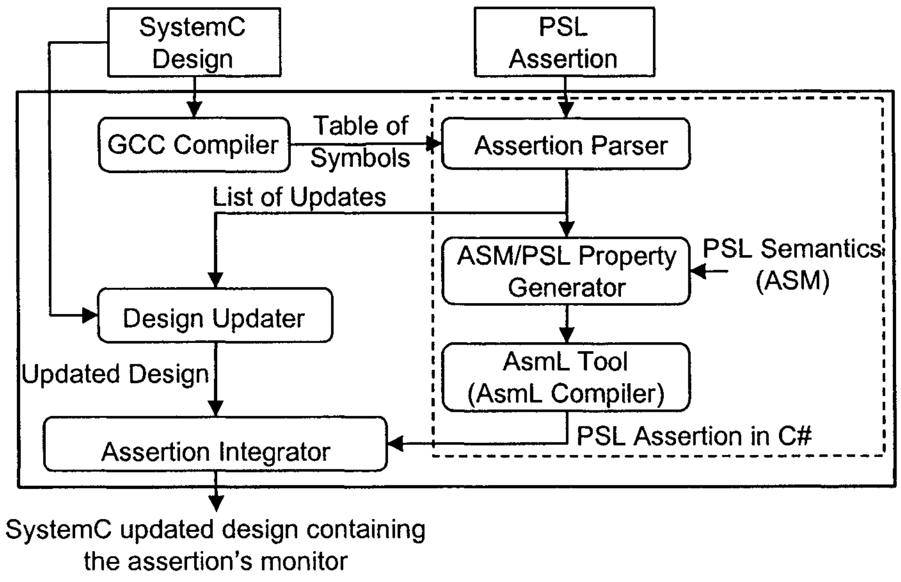


Figure 6.3: Extending SystemC by PSL Assertions

signals (variables) that are used in the assertion must be seen as external signals so that they can be input to the assertion monitor. So, we provide the design updater with a list of variables as defined by their unique identifier in the table of symbols. Then, the design updater modifies the SystemC design to make the required variables visible to the monitor. This transformation does not affect the behavior of the code as it will only be accessed in a read-only mode.

Once the code is updated and the assertion is generated, the design integrator will add the required instantiation of the assertion to bind it to the existing SystemC design modules. The assertion monitor, acting as part of the design, can do the following: (1) stop the simulation when the assertion is fired; (2) write a report about the assertion status and all its variables; and (3) send a warning signal to other modules (if required). Note that the internal code of the assertion is C# so the designer can update it or do any other functionalities that can be coded in C#.

6.3.1 Related Work

In [45], Gordon used the semi-formal semantics in the PSL/Sugar documentation [1] to create a deep embedding of the whole language in the HOL theorem prover [46]. The author developed the formal definition of the full PSL language in HOL. The combination of PSL/Sugar and higher-order logic is quite expressive and provides temporal logic constructs as higher level syntactic sugar for higher order-logic, thereby enabling properties to be formulated elegantly. Gordon *et al.* [45] described how to 'execute' the formal semantics of PSL using HOL and investigated the feasibility of implementing useful tools to conduct automatic verification of PSL from the formal semantics. They implemented two experimental tools: an interpreter that evaluates whether a finite trace, satisfies a PSL formula, and a compiler that converts PSL formulas to checkers in an intermediate format suitable for translation to HDL to be included in simulation test-benches. However, they did not provide any framework for the verification of PSL for any implementation language.

In a similar work, Claessen and Martensson [16] defined an operational semantics for a weak fragment of PSL, mainly the safety property subset of PSL, and then proved the correctness of these semantics with respect to a denotational semantics. They do not provide definitions for all PSL operators like clock operators and sequential composition, and yet, there is no execution for these semantics that provides verification solution.

There has been a potential work on ASM verification as discussed by Börger and Stärk [9]. Applying model checking algorithms on ASM was introduced in [108], where transformation algorithms are provided to transform ASM models into different verification tools. Two approaches were adopted: the first provides a transformation to the language of a symbolic model checker, SMV [108], and the second to the MDG verification tool [39]. Spielmann [97] investigated the problem of verifying a class of restricted abstract state machine programs automatically. The work we present here, is different since it provides a solution for the verification problem of system level design languages based on semantics definitions and executions of PSL and SystemC.

Stärk *et al.* [99] used an ASM-based modularization technique to define a structured sequence of mathematical models for the statics and dynamics of the Java programming language and for the Java Virtual Machine (JVM), covering the compilation of Java programs to JVM code and the JVM bytecode verifier. They present proofs of correctness, completeness, and type safety for the language and the Java machine. Börger *et al.* [8] used the method developed in [99] to define the semantics of C# programs in ASM, which provided a simple way to reflect those run-time-related features encountered upon executing a given C# program and allowed specifying the static and dynamic parts of the semantics separately. The dynamic semantics of the language is captured operationally by ASM rules which describe the run-time effect of program execution on the abstract state of the program, the static semantics comes as a declarative description of the relevant syntactical and compile-time checked language features. In a complementary work, Stärk and Börger [98] extended the modular definition of the semantics of C# in [8] by a new module for multi-threaded C# focusing on purely managed, fully portable threading features of C# and the .NET common language runtime. Jula and Fruja [62] provided an executable AsmL semantics for these C# semantics. In a later work, Jula [61] extended the work in [8] to handle C# 2.0 specific features like generics, anonymous methods and iterator blocks.

ASM has been used thoroughly to define the operational semantics of programming languages like C++, Prolog, SDL, and Standard ML [55]. However, these semantics definitions provide no execution of the language semantics itself in order to give a solution to design problems like verification.

6.3.2 Embedding PSL in AsmL

There are two ways to embed PSL properties into the design, either into the design code itself or by adding them as external monitors. We adopted the first approach, where all the parameters of PSL properties are defined as objects. The objective of the embedding is to reuse PSL properties, as embedded in AsmL, at lower design levels since the AsmL tool can automatically compile them into a C# or .NET code. This latter code can be compiled and executed with the concrete SystemC level [1].

6.3.3 Illustrative Example I: Packet Switch

Assertions

For illustration purposes, we consider the following three PSL assertions for the packet switch used in the previous Chapter.

The first assertion, $A1$, is intended to verify that if there is only one recipient for the packet, and the output queue is not full, then the register that holds the packet should be free in the next internal clock, and the packet should be received at the output queue.

```
$AssertionA1:$
    $forall\ send\ in\ \{0,\ 1,\ 2,\ 3\}$
      $if\ Reg[send].free\ ==\ true\ and$
        $Packet.dest0\ and\ not\ OutQueue[send].full\ and$
        $not\ Packet.dest1\ or\ Packet.dest2\ or\ Packet.dest3$
         $then\ at\ next\ SWCLK:$
          $Reg[send].free\ =\ true$
          $OutQueue[0]\ =\ Reg[send]$
```

The second assertion, $A2$, is intended to check the shortest path when sending from sender i to receiver i, where the input queue is not full. This operation should be performed in four internal clocks ($SWCLK$) or equivalently one external clock (CLK).

$AssertionA2:$

[1]The details of the PSL embedding in AsmL are provided in the Appendix B.

```
$forall\ send\ in\ \{0,\ 1,\ 2,\ 3\},\ forall\ rec\ in\ \{0,\ 1,\ 2,\ 3\}$
  $if\ send\ ==\ rec\ and\ not\ InQueue[send].full\ and$
    $Reg[send].free\ ==\ true\ and$
    $OutQueue[rec].empty\ ==\ true\ then$
     $OutQueue[rec] = Reg[send]\ in\ 4\ SWCLK$
     $and\ OutQueue[rec] = Reg[send]\ in\ 1\ CLK$
```

The third assertion, $A3$, is intended to check the (worst) longest path when sender 0 transmits to receiver 3, input queue 3 has only one free slot, all other input queues are full, and the output queue 0 is full. This assertion is specified as follows:

```
$ifsend\ ==\ 3\ and\ rec\ ==\ 0$
  $and\ InQueue[3].size\ ==\ 3$
  $and\ InQueue[0].full\ and\ InQueue[1].full$
  $and\ InQueue[2].full\ and\ OutQueue[0].full\ then$
    $OutQueue[0]\ ==\ Reg[3]\ in\ 8\ to\ 19\ SWCLK$
```

Results

The AsmL tool is used in order to generate automatically the corresponding C# code for the above PSL assertions. Figure 6.4 shows, as the example of the integration of the generated C# model for assertion $A1$ with the SystemC model. The connection to the existing objects in SystemC model is done using read-only signals extracted from the packet main module and the switch clock generator.

The simulation of the new model that combines the original design and the integrated PSL properties resulted in successfully verifying the correctness of assertions $A2$ and $A3$. Assertion $A1$, however, was violated, indicating a bug in the SystemC packet switch model. This bug showed, after further inspection of the code, that the switch will free any packet coming from senders 0, 2 and 3 and having at least two

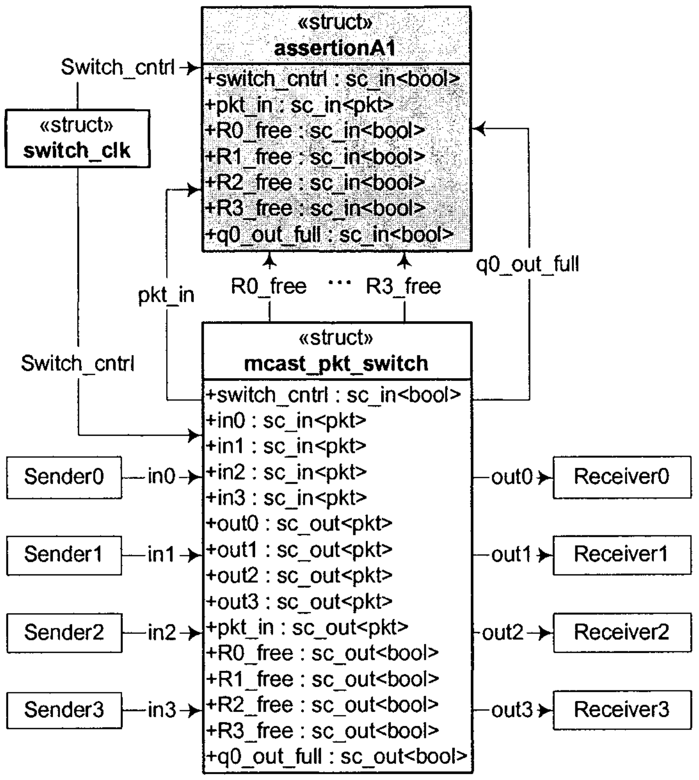


Figure 6.4: Integrating Assertion $A1$ with SystemC Model of the Packet Switch

destinations including port 1 before routing it to output port (different from port 1). The erroneous code is the following:

$if\,(R1.val.dest1||R1.val.dest1||R1.val.dest2||R1.val.dest3)$

$R1.free = true;$

where the condition to free the register does not check if the packet is having as destination the port 0 and uses a double copy of the check about the port 1 ($R1.val.dest1$). The correct condition should be:

$if\,(R1.val.dest0||R1.val.dest1||R1.val.dest2||R1.val.dest3)$.

6.3.4 Illustrative Example I: Simple Bus

We consider a bus structure with N masters and M slaves[2]. Each master is identified by a unique priority, that is represented by an unsigned integer. There are two possible modes for the bus: (1) *Blocking Mode*, where data is moved through the bus in a burst–mode. Here, the transaction cannot be interrupted by a request with a higher priority, (2) *Non-Blocking Mode*, where the master reads or writes a single data word. Figure 6.5 shows the protocol used in both modes of operation.

Bus Properties in PSL

For illustration purposes, we considered two properties for the bus architecture: one for the non–blocking master mode, and the other for the blocking master mode.

Property P1:

If ((*MasterBlock.Request* = *true*) & (*BusStatus* = *OK*) &
  (*MasterBlock.Priority is the highest*)) ***then***
  (*MasterBlock*[3] = *OKSend*) &
  (*BusStatus*[3] = *Used*) &
  (*MasterBlock.DestSlave*[4] = *Recev*) &
  (*MasterBlock.DestSlave*[5] = *Ack*) &
  (*MasterBlock*[7] = *Done*) &
  (*Bus*[8] = *Ready*)

meaning that when a blocking master generates a request while it has the highest priority to use the bus and the bus is available, then at the third clock cycle, the status of the master should be $OKSend$, and the bus should be in the $Used$ status. Then the status of the destination slave should be *Recev* at clock cycle 4, and *Ack* at clock cycle 5. The status of the master should be *Done* at clock cycle 7, and finally

[2]A more detailed description of the bus structure is provided in Appendix C.2.

the bus becomes ready to handle new requests (i.e., bus status set to *Ready*) at clock cycle 8. This property is illustrated as a sequence diagram in Figure 6.5(a).

Property P2:

If (($MasterNBlock.Request$ = $true$) & ($BusStatus$ = OK)
& ($MasterNBlock.Priority$ *is the highest*)) ***then***
($MasterNBlock$[3] = $OKSend$) &
($BusStatus$[3] = $Used$) &
($MasterNBlock.DestSlave$[4] = $Recev$) &
($MasterBlock$[5] = $Done$) &
(Bus[5] = $Ready$)

This property can be interpreted in a similar way as property $P1$ and is illustrated in Figure 6.5(b).

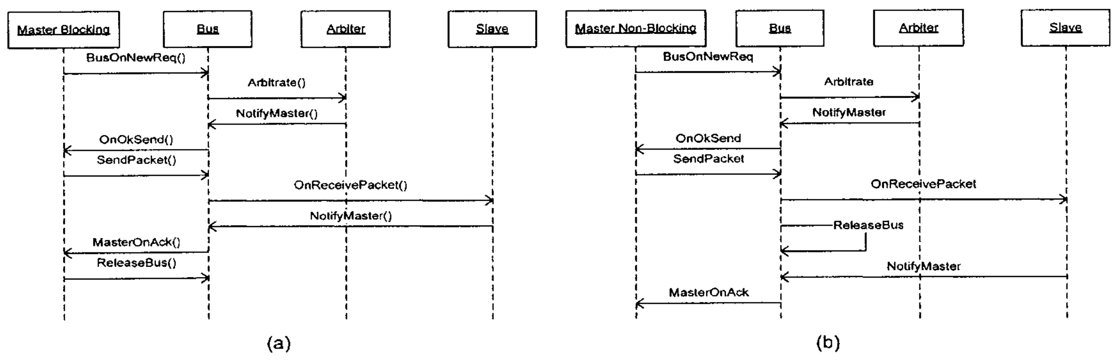


Figure 6.5: Simple Bus Modes: (a) Blocking Mode (b) Non-Blocking Mode.

Properties in PSL-ASM

Both properties were defined in AsmL based on the embedding of PSL layers. Figure 6.6 shows the definition of $P1$ as an example. The property is included in a PSL unit as an implication of two sequences $seq1$ and $seq2$, which are formed from basic Boolean items ($Bi1()$ and $Bi2()$ for $seq1$ and $Bi3()$ through $Bi7()$ for $seq2$). The construction of the above unit includes four steps:

- Creating the basic Boolean items: $Bi1()$ to $Bi7()$.
- Creating the sequences: $seq1$ and $seq2$.
- Constructing the implication property ($P1$) from $seq1$ and $seq2$ using the implication operator.
- Putting $P1$ into an embedded PSL unit.

```
//Blocking Masters Instances
var masterB1 as C_MasterBlocking = new C_MasterBlocking var
masterB2 as C_MasterBlocking = new C_MasterBlocking MASTERSB =
{masterB1, masterB2}

Bi1() as Boolean Bi: Boolean Item
  return exists master in MASTERSB where
                        master.m_status = MasterReq
... Bi7() as Boolean
  return bus.m_status = BusOK

var seq1 as PSL_SERE = PSL_SERE(2)
         seq1.AddElement(param1, param2)
var seq2 as PSL_SERE = PSL_SERE(5)
         seq2.AddElement(Bi3, Bi3, Bi4, Bi5, Bi6, Bi7)
var property as PSL_FL_Property = PSL_FL_Property()
property.AddImplication(seq1,seq2) var Assertion1 as
PSL_VerificationLayerUnit = new
PSL_VerificationLayerUnit(''Assertion1'')
Assertion1.AddProperty(property)
```

Figure 6.6: Definition of the PSL Property P1 in AsmL.

6.4 Application: PCI Bus

In order to evaluate our approach with complex yet real SystemC designs, we consider as application a PCI (Peripheral Component Interconnect) [50] Local Bus. The

PCI bus is a high performance bus for interconnecting chips, expansion boards, and processor/memory subsystems. It was adopted as an industry standard administered by the PCI Special Interest Group (PCI SIG) [50] [3].

6.4.1 Assertions Specification

In addition to the properties defined in [95], we considered several other more complex properties, which define a complete sequence of transactions over the bus. In what follows are presented three sample properties:

Property P_1 :

```
$forall\ Master\ in\ \{Master0,\ \ldots,\ Master4\}$
  $if\ (!Master.REQ\ ==\ true)\ then$
    $eventually\ (!Master.GNT\ ==\ true)$
```

meaning that if a master requests the bus $(!Master.REQ\ ==\ true)$ it will get access to it in the future $(!Master.GNT\ ==\ true)$, which guarantees that no master will use the bus indefinitely.

Property P_2 :

```
$forall\ Master\ in\ \{Master0,\ \ldots,\ Master4\}$
$forall\ Slave\ in\ \{Slave0,\ \ldots\ Slave4\}$
  $if\ (!Master.GNT\ ==\ true)\ and$
      $(!Master.DEST\ ==\ Slave.ID)$
  $then\ eventually\ (!Bus.FRAME\ ==\ true)\ and$
      $(!Master.TRDY\ ==\ true)\ and$
      $(!Slave[ID].TRDY\ ==\ true)\ and$
      $(!Master.GNT\ ==\ false)$
```

[3]A detailed bus description is provided in Appendix C.3.

meaning that if a master is selected by the arbiter, then it will be able to get access to the bus by setting $!Bus.FRAME$. Thereafter, its destination slave will be activated by setting its $!Slave.TRDY$. Finally, the master will release the bus once $!Master.GNT$ is set to false.

Property P_3 :

```
forall Master in {Master0, ..., Master4}
  if (!Master.STOP == true) and
      (!Master.GNT == true) then
      eventually{(!Bus.FRAME == false) and
      forall Slave in {Slave0, ... Slave4}
      (!Slave.TRDY == false) and
      (Slave.IDSEL == false)}
```

meaning that if a master stops a request, then all slaves will be released.

6.4.2 Experimental Results and Discussion

Table 6.2 shows a simulation evaluation of the PCI bus when implemented in SystemC. We display the average execution time per clock cycle as a function of the number of masters and slaves connected to the bus[4].

6.5 Summary

In this Chapter we described two possible approaches to integrate assertions to SystemC. In both cases, the assertion is considered as an external monitor connected in read-only mode to the original SystemC design. In the first approach, we proposed to directly generate monitors from SystemVerilog assertions. While, in the second approach, we used a PSL embedding in AsmL in order to automatically generate

[4]All experiments were performed on a 2.4 GHz Pentium IV and 512 MB of RAM (PC 2700).

Table 6.2: Simulation Results.

Number of		Average Execution
Masters	Slaves	Time per Clock Cycle (10^{-9}s)
1	1	24.31
1	2	29.32
3	1	29.766
2	2	30.891
2	3	32.744
3	2	34.032
3	3	36.828

C# assertion from properties encoded in PSL. Experimental results, show a very fast simulation models integrating assertions, which illustrates the suitability of using our ABV methodology with SystemC. The question of enhancing the assertion coverage by simulation is discussed in the next Chapter.

Chapter 7

Enhanced Simulation Coverage

7.1 Introduction

In the last Chapter we present two possible approaches how to integrate assertions to SystemC. However, the objective of the verification process is not only to write assertions but to verify them. This latter task is usually performed using test vectors generation tools mostly based on random processes. This kind of blind simulation does not guarantee that the assertion will be covered during the test execution. Therefore, it is very important to consider a smarter and more efficient test vector generation approach. For this reason, we propose to optimize tests at TLM and, then, to reuse them at RTL. To bring into play such an idea, two main questions must be answered at the transaction level: (1) how to measure the coverage? and (2) how to improve it?

As a solution to the coverage measurement question, we propose a layered design for verification approach involving both TLM and RTL. At the former level, the design is modeled in AsmL where communication between system's components relies on direct functional calls. This simplified model is more suitable for the generation of the system's FSM. Raising the level of abstraction tackles the problem of state explosion, usually faced with RTL designs. Once the FSM generated, we define a functional coverage in terms of state space coverage. For example, if we want to check a read operation, we define it as the set of states that must be visited in every operation.

In order to improve the coverage, we propose to use a genetic algorithm (GA) where the target is to optimize the random test generation. The basic concept is to find a good random distribution of the inputs' ranges in order to ensure a higher level of coverage. The final output of this operation is a test vector generator with a high coverage rate (at least in comparison to blind random simulation) w.r.t. a predefined objective.

The test generator produced using the GA optimization technique at transaction level is reused to validate the RTL design. A bi-simulation relation between both models guarantees that the coverage remains the same w.r.t. the same assertions. In general, such a relation is not guaranteed. For instance, designers may modify certain parts of the code for optimization purposes, for example, when writing the RTL model. Hence, we propose to compare the coverage results for the RTL level using our GA and using the random simulation provided by the commercial tool Specman of Verisity [106].

7.2 Related Work

Genetic algorithms have already been used for a broad range of applications. In contrast to other approaches, Godefroid *et al.* [41] addressed the exploration of large state spaces of concurrent reactive systems as defined for model checking. However, this work was restricted to simple Boolean assertions and was based on BDDs which is not suitable for high level languages like SystemC. In contrast to [41], we propose to add a static analysis phase of the code before applying the genetic algorithm. We also considered a chromosome-encoding based on weighted probability over the space of the possible values of the program variables. In this work, we propose to: (1) use

AsmL to implement transaction level models; (2) define a coverage as function of the system's FSM (generated from the AsmL model at TLM); (3) initialize the GA using the information gathered from the assertion and the system's FSM; and (4) employ TLM models as intermediate step in order to identify efficient test generator for RTL designs.

There exist a variety of efficient EDA tools for test and assertion coverage, e.g., Specman Elite [106] of Verisity, TestBuilder [12] from Cadence and TestBencher Pro [100] from SynaptiCAD. They use a user-defined constrained random simulation in order to perform higher functional coverage. However, these tools do not take advantage from the design specific properties. Besides, they relate the coverage to the number of times the assertion was executed while a correct evaluation has to consider what portion of the assertion's state space was covered. Therefore, current tools were developed for low HDL level designs (using Verilog and VHDL) and do not define coverage metrics for TLM languages such as SystemC assertions. In this work, we compare coverage results obtained using our approach to the output of Specman Elite tool for the same RTL model.

7.3 Proposed Methodology

Performing a full coverage of a system's state space using simulation is not feasible. Consequently, more interest has to be focused in order to develop smart verification approaches. In our proposed methodology we aim to make use of two features: transaction models and genetic algorithms.

Transaction models run faster than timed models. Avoiding clocks and raising up the level of abstraction by using channels and direct functional calls accelerates the simulation. Furthermore, these models are conceptually closer to the system specification. This latter is generally a collection of properties that could be verified, as assertion monitors, by simulation. Running faster simulation may result in better coverage. Confirming such a result requires defining a precise measurement for the coverage. Classical ways to measure the coverage at RTL (code, condition, assertion coverage, etc.) do not give a realistic evaluation about which functionality was verified. For this reason, we propose to take advantage of transaction models to define assertions that could be used to verify the final RTL product.

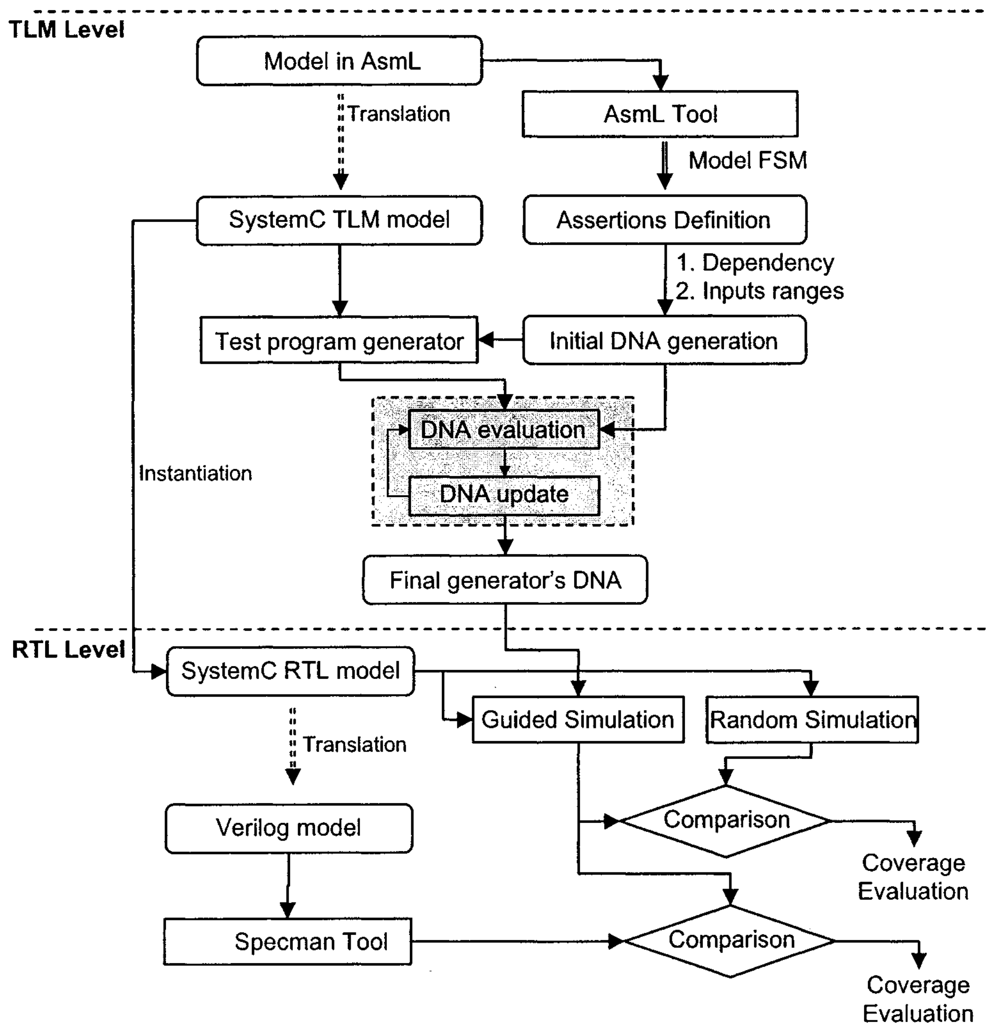


Figure 7.1: Proposed Methodology.

Generating finite state machines (FSM) from transaction models provides a way to define assertion as sequence of states. In our methodology of Figure 7.1, we proposed to use AsmL [51] as TLM modeling language. Two main reasons influenced our choice: the language features and the FSM generation algorithm available for models written in this language. For instance, AsmL is an object-oriented language rich with several mathematical constructs and data abstraction. A number of algorithms have been developed around this language, collected under the AsmL tool (Asmlt) [74], developed at Microsoft Research. In this work, we make use of the algorithm generating FSMs from AsmL models in order to extract the system's FSM. This latter is adequate to define functional coverage as function of the system's states.

Enhancing random simulation should always consider the assertion(s) under verification as reference. Testing a memory read access, for example, should only focus on the read port operation. For the general case, we propose to use the FSM generated from system's TLM model (in AsmL) as a guidance for relevant variables and input values for testing a specific operation. This is possible by looking into the state variables involved in the set of states representing the assertion. Therefore, optimizing the coverage becomes an issue of guiding the simulation to feed the system with particular values. Unfortunately, at TLM, it is not always possible to find a direct relation between the variables involved in the assertion and the system's inputs. Hence, what we can extract from the FSM is an over-approximation of this relation. It follows that we cannot define precisely the complete set of tests to validate a specific functionality. A solution to enhance the test generation process is to use genetic algorithms.

We use the initial knowledge gathered from the generated FSM as an initialization for a genetic algorithm aiming to enhance the assertions coverage. The input of the algorithm is a set (population) of test vector generators. After applying a number of tests, the generators are updated to form a new community. The fitness is a function of the assertion coverage. Theoretically, by iterating this process, we will obtain a generation of efficient test vectors generators w.r.t. to the assertion function criteria.

In this work, we propose to generate the RTL design from the TLM model by instantiation. This approach may not work for all kind of designs. But, generally, for pure hardware designs, it is always possible to translate TLM to RTL through instantiation. In fact, transaction modeling is an intermediate step to enhance the verification of the design at RTL. For this reason, once the coverage achieved at transaction level is satisfactory, we perform two evaluation processes at the RTL. We compare the coverage achieved using the best generation of test generators output of the previous phase to the coverage obtained using: (1) random simulation; and (2) the commercial tool Specman [106]. The first comparison aims to show that improvement of the coverage is preserved when lowering the abstraction level from TLM to RTL. The second comparison targets to illustrate the efficiency of our proposed approach when compared to commercial well-established tactics involved in the tool Specman tool.

7.3.1 AsmL TLM Models

The system's TLM model in AsmL is a behavioral specification. It includes a light description of the system's functionalities. All the components are communicating using transactions involving direct functional calls. The simulation environment includes the notion of *updates*, i.e., a variable value is not changed until an update is requested. For this reason, we embedded a light simulation environment in order to manage events and processes.

The system components are objects instantiations of classes (also called $Module^{A}_{TLM}$) defined according to Definition 7.3.1.

Definition 7.3.1. (AsmL TLM Module: $Module^{A}_{TLM}$)
An AsmL TLM module is a set $\langle AS_{DMem}, AS_{Mth}, AS_{Ctr} \rangle$, where AS_{DMem} is a set of the module data members, AS_{Mth} is a set of methods (functions) definition and AS_{Ctr} is the module constructor.

For every method in AS_{DMem} corresponds a Boolean pre-condition enabling its execution. This is a critical issue in constructing the actual AsmL TLM design because a wrong definition of the pre-condition may totally change the behavior of the system and consequently modify the verification results. The pre-condition rules define the way of communication between the design's components. The design is defined as a collection of modules and an initialization method according to Definition 7.3.2.

Definition 7.3.2. (AsmL TLM Design: $Design^{A}_{TLM}$)
An AsmL TLM design is a set $\langle LModule^{A}_{TLM}, INIT \rangle$, where $LModule^{A}_{TLM}$ is a set of AsmL TLM modules and $INIT$ is the initialization function of the model.

In order to perform adequate partial and total updates, we make use of a light simulation manager partially described in Figure 7.2. This simulator includes an initialization function that is executed after all the design's modules have been initialized (condition: SystemFlag = STARTED). The same method includes a second pre-condition setting that the method is only executed at the initialization phase (condition: SimStatus = INIT). This illustrates how the pre-condition constructor is used to manage the exploration algorithm performing the reachability analysis.

7.3.2 FSM Generation

We use an FSM generation algorithm defined by Gurevich et *al.* in [51]. It requires the following inputs: domains, methods, actions and variables (optional inputs are filters, action groups and properties). The transitions in the FSM are the method calls (including argument values) in the test sequences. The methods in the model program that appear in the transitions are called actions. The states in the FSM are determined by the values of selected variables in the model program, called state variables. The algorithm generates the FSM by executing the model program in a special execution environment. It keeps track of the actions while recording the states it visits. This process is called *exploration*.

```
class SimManager
 public var m_K as ClockEvent = CLK_UP       // Main system clock
 public var m_Ks as ClockEvent = CLK_DOWN // Negation of the main clock
 public var m_E as BANK_ID = BANK_0

 public SimManager()

public SimManager_Init()
  require SystemFlag  = STARTED and SimStatus = INIT
  me.m_K  := CLK_UP   //set first clock to high
  me.m_Ks := CLK_DOWN //set second clock to down
  forall  w in WPORTS
   if(w.m_E = m_E) then
    w.LA1_WP_OnReceiveData_Depth  := any rec | rec in {true,false}
   else
    w.LA1_WP_OnReceiveData_Depth  := false
  forall r in RPORTS where r.m_E = m_E
   if(r.m_E = m_E) then
    r.LA1_RP_OnReadData_Depth     := any rea | rea in {true,false}
   else
    r.LA1_RP_OnReadData_Depth     := false
  forall  s in SRAMS
   s.LA1_SRAM_OnWriteData_Depth  := false
  SimStatus := CHECKING_PROP

 public SimManager_Restart()
  require SystemFlag  = STARTED and SimStatus = STOPPED
  SimStatus := INIT
```

Figure 7.2: Light Simulation Manager in AsmL.

The generated FSM will, according to the algorithm's configuration options, represent only a portion – an under-approximation – of the huge FSM that would result if the model program could be explored completely. The FSM generation process requires a set of Boolean guards in order to reflect the state distinction that the model designer cared enough about to make explicit. The algorithm takes a distinguishing sequence as an additional input to produce corresponding equivalence classes, called *Hyperstates*.

The FSM generation algorithm in [51] requires the definition of the following:

- data types and static functions
- declarations of state variables $v_1, v_2, \dots v_s$ (v refers to variable and s is the total number of states) that characterize the state space of the considered system
- rules that describe the transition relation of the system: $c_1, c_2, \dots c_v$ (c refers to

a Boolean condition (rule), v is the total number of rules).

The classical problem challenging FSM based approaches is state explosion. For this reason, the notion of states indistinguishability represents the main key feature of the algorithm in [51] in comparison to other techniques. Furthermore, this notion fits well to the conceptual structure of TLM models where states can be combined according to their affiliation to a specific transaction.

Definition 7.3.3. (States Indistinguishability)
Let $\mathcal{C} = \{c_i(v_1, v_2, \ldots v_s), i \in \{1 \ldots n\}\}$ be a set of Boolean conditions on state variables. Two states $s_a = \{a_1, a_2, \ldots a_s\}$ and $s_b = \{b_1, b_2, \ldots b_s\}$ are indistinguishable if: $\forall c \in \mathcal{C}$, $c(s_a) = c(s_b)$.

Defining the conditions in $\mathcal{C}$ is an additional effort required in building the TLM models in AsmL. A good tactic to surmount this problem is to define a specific condition c in $\mathcal{C}$ for each transaction. Hence, a natural link will be defined between transactions and hyperstates.

7.3.3 Assertion Scope

An assertion is as a monitor tracking the system's state to verify a set of conditions. Considering a generated FSM, the assertion could be represented as a collection of hyperstates and transitions (sometimes portion of the global FSM). Classical FSM coverage, used at RTL, always deals with the full system's FSM (state or transition coverage) [60]. Generally, the problem of state explosion raises when considering data paths. That is why, most of the verification effort is focused on the control path. In contrast to the classical approaches, our target is not to go for all possible combinations raising from the system's FSM (which could be infinite). We define the assertion coverage as a coverage of a set of hyperstates (for the sake of simplicity, we will use the word state to refer to hyperstates in the rest of this Chapter) and transitions in the generated FSM. In Definition 7.3.4, we define an assertion as a collection of state variables and Boolean conditions involving at least one of the assertion's state variables.

Definition 7.3.4. (Assertion Definition: A)
Let $Design^{A}_{TLM}$ be an AsmL model, $\mathcal{V} = \{v_1, v_2, \dots v_s\}$ its state variables and $\mathcal{C} = \{c_i(v_1, v_2, \dots v_s), i \in \{1 \dots n\}\}$ be a set of Boolean conditions on state variables. An assertion A is the set $\langle A_v, A_c \rangle$ where A_v is a subset of $\mathcal{V}$ and A_c is a subset of $\mathcal{C}$ involving at least one variable $v_i \in \mathcal{V}$.

Checking the assertion does not involve all the states in the FSM. The subset that we are interested in is restricted to the states where one of the assertion's guards (conditions) is evaluated to true. This subset is called *Assertion Scope* A_{sco}.

Definition 7.3.5. (Assertion Scope: A_{sco})
Let $Design^{A}_{TLM}$ be an AsmL model, $\mathcal{F}$ its generated FSM, $\mathcal{V} = \{v_1, v_2, \dots v_s\}$ its state variables, $\mathcal{C} = \{c_i(v_1, v_2, \dots v_s), i \in \{1 \dots n\}\}$ be a set of Boolean conditions on state variables and $A = \langle A_v, A_c \rangle$ be an assertion. Then, the assertion scope is $A_{sco} = \{s \in \mathcal{F}, \text{ where } \exists c \in A_c \,|\, c(s) = true\}$.

The assertion scope, A_{sco}, collects all the states that are of interest for the assertion A. Definition 7.3.5 does not guarantee that the assertion's scope will be an automata [104] (the commonly used mathematical model to represent system properties). Nevertheless, since we are interested in verifying the assertion using simulation, defining the assertion scope A_{sco} as a set of states is sufficient.

7.4 Coverage Evaluation

Considering the assertion's definition and scope, we propose two principle coverage metrics: state coverage and transition coverage. In contrast to the RTL classical coverage, we are dealing with hyperstates. Furthermore, we are not looking into verifying the coverage of the whole system's FSM (which can be infinite or very large).

A trivial way to define the FSM state coverage is to count the number of states visited by the test vectors over the total number of states. This kind of coverage cannot be used when the FSM is formed from hyperstates. Visiting a hyperstate does not only depend on the state itself but also on the guards elements of the set of Boolean conditions on state variables, $\mathcal{C}$. It is possible for a guard to be true for different combinations of the state variables. Therefore, counting all possible combinations getting to a hyperstate is mandatory for getting a true evaluation of the state's coverage.

7.4.1 State Coverage

Before giving the state coverage's definition, we first introduce the notion of assertion state space, A_{sp}. This latter refers to all the possible state space combinations involved in the assertion according to Definition 7.4.1.

Definition 7.4.1. (Assertion State Space: A_{sp})
Let $A = \langle A_v, A_c \rangle$ be an assertion. The assertion state space is:

$$\begin{aligned} A_{sp} = \quad \{ \quad & (v_{c1}, v_{c2}, \ldots v_{cs}) \text{ instance of} \\ & (v_1, v_2, \ldots v_s) \in A_v \mid \\ & \exists c \in A_c \mid c(v_1, v_2, \ldots v_s) = \mathit{true} \} \end{aligned}$$

where $(v_{c1}, v_{c2}, \ldots v_{cs})$ is a concrete instance of $(v_1, v_2, \ldots v_s)$.

Considering concrete instances of variables in Definition 7.4.1 is important because it is possible to use abstract variables in AsmL. The number and the nature of the concrete elements depend on the variable's domain.

Next, we use the assertion's state space definition, in order to evaluate the assertion coverage.

Definition 7.4.2. (State Coverage: S_{cov})
Let A be an assertion, A_{sco} its scope and A_{sp} its state space. Let $T_{sp} = \{(v_{tc1}, v_{tc2}, \dots v_{tcs})$ concrete instance of $(v_{t1}, v_{t2}, \dots v_{ts}) \mid v_{ti} \in A_{sp}\}$ be a set of test vectors. The state coverage obtained by executing T_{sp} is:

$$S_{cov} = \frac{Card(T_{sp})}{Card(A_{sp})}$$

where *Card* is the set's cardinality.

The state coverage, S_{cov}, computes the fraction of states (element of the assertion's state space) visited during the execution of the test vectors' set. An optimal testing case will provide a state coverage equal to one. For a general graph structure, we cannot guarantee the existence of an optimal test sequence. However, when the FSM is a *connected* graph, Theorem 7.4.1 guarantees that such an optimal case exists. The theorem's proof provides a construction of such a sequence.

Theorem 7.4.1 *(Optimal Test Sequence for S_{cov})*
Let $Design^{A}_{TLM}$ be an AsmL model and $\mathcal{F}$ its generated FSM. If $\mathcal{F}$ is a connected graph, then, there exists a test sequence T_{sp} such that $S_{cov} = 1$

Proof 7.4.1. If $\mathcal{F}$ is a connected graph, then, for any two states a and b in $\mathcal{F}$, there exists a path from a to b. In this case, the proof of the theorem can be done by constructing a test sequence satisfying $Card(T_{sp}) = Card(A_{sp})$. Such a test sequence is formed from a set of test vectors each starting from the initial state and getting to a state in the assertion's state space A_{sp}. By defining at least a path for every element in A_{sp}, we ensure that $Card(T_{sp}) = Card(A_{sp})$.

7.4.2 Transition Coverage

We define transition coverage, T_{cov}, by identifying all the states involved in a transition from or to a state element of the assertion's space. In following, we first introduce in Definition 7.4.3 the assertion transition space, A_{tp}. Then, we will define the transition coverage.

Definition 7.4.3. (Assertion Transition Space: A_{tp})

Let $Design^{A}_{TLM}$ be an AsmL model, $\mathcal{F}$ its generated FSM, $\mathcal{V} = \{v_1, v_2, \ldots v_s\}$ its state variables, $\mathcal{C} = \{c_i(v_1, v_2, \ldots v_s), i \in \{1 \ldots n\}\}$ a set of Boolean conditions on state variables and $A = \langle A_v, A_c \rangle$ an assertion. The assertion transition space is:

$$\begin{aligned} A_{tp} = \quad \{ \quad & (v_{c1}, v_{c2}, \ldots v_{cs}) \text{ instance of } v \in \mathcal{V} \mid \\ & \exists v_a \in A_v \text{ and } tr \in \mathcal{T} \mid \\ & (tr(v, v_a) = true) \vee (tr(v_a, v) = true)\} \end{aligned}$$

where $\mathcal{T} = \{tr_1, \ldots tr_m\}$ is the set of the transition in $\mathcal{F}$.

Similarly to the assertion state coverage (see Definition 7.4.2), we can define the assertion's transition coverage, T_{cov}, using the assertion transition space, A_{tp}.

Definition 7.4.4. (Transition Coverage: T_{cov})

Let A be an assertion, A_{sco} its scope and A_{sp} its state space. Let $T_{tp} = \{(v_{tc1}, v_{tc2}, \ldots v_{tcs})$ concrete instance of $(v_{t1}, v_{t2}, \ldots v_{ts}) \mid v_{ti} \in A_{tp}\}$ be a set of the test set. The state coverage obtained after executing T_{tp} is:

$$T_{cov} = \frac{Card(T_{tp})}{Card(A_{tp})}.$$

The transition coverage, T_{cov}, computes the fraction of states (element of the assertion's transition space) visited by a test vector. The optimal test case will provide an assertion transition coverage equal to one. When the FSM is a *clique* graph, Theorem 7.4.2 guarantees that such an optimal case exists. The theorem's proof provides a construction of such a sequence.

Theorem 7.4.2 *(Optimal Test Sequence for T_{cov})*

Let $Design^{A}_{TLM}$ be an AsmL model and $\mathcal{F}$ its generated FSM. If $\mathcal{F}$ is a clique graph (complete graph), then, there exists a test sequence T_{sp} such that $T_{cov} = 1$

Proof 7.4.2. If $\mathcal{F}$ is a clique graph, then, for any two state a and b in $\mathcal{F}$, there exist a transition from a to b. In this case, the proof of the theorem can be done by constructing a test sequence satisfying $Card(T_{tp}) = Card(A_{tp})$. Such a test sequence is formed from a set of all test vectors each starting from the initial state and getting to a state in the assertion's state space (A_{sp}). By covering all the elements in A_{tp}, we ensure that $Card(T_{tp}) = Card(A_{tp})$.

7.5 Enhancing the Coverage

Theorems 7.4.1 and 7.4.2, respectively, guarantee the existence of optimal test sequence (with coverage equal to one) for the particular cases of connected and clique graphs, respectively. However, in the general case, finding or even proving the existence of an optimal test sequence is not trivial. Furthermore, even when an optimal sequence exists, its size could be very large. Hence, in this section, we propose a genetic algorithm based technique aiming to optimize the coverage using a randomly generated test sequences.

7.5.1 Genetic Algorithm

Genetic algorithms belong to a family of computational models inspired by evolution [52]. They encode a potential solution to a specific problem on a simple chromosomes like data structure and apply recombination operators to these structures to preserve critical information. They evolve candidate solutions to problems that have large solution spaces and are not amenable to exhaustive search or traditional optimization techniques. This is the reason why they are often viewed as function optimizers. Since their introduction by Holland [52], genetic algorithms have been applied to a broad range of learning and optimization problems [89].

Genetic algorithm is any population based model that uses selection and recombination operators to generate new sample points in a search space. Typically, a genetic algorithm starts with a random population of encoded candidate solutions (test generators for our case), called chromosomes. Through a recombination process and mutation operators, it evolves the population towards an optimal solution. The challenge is to design a genetic process that maximizes the likelihood of generating an optimal solution. This objective can be guaranteed by applying a number of steps. First, we evaluate the *fitness* of each candidate solution in the current population, and select the fittest candidate solutions to act as parents of the next generation of candidate solutions. Then, the selected parents are recombined (using a crossover operator) and mutated (using a mutation operator) to generate offsprings. The fittest parents and the new offsprings form a new population, from which the process is repeated to create new populations.

The state variables of the system are classified into three groups: inputs, outputs and internal variables. This step is required in order to define the connection between the system and test vectors generator.

Definition 7.5.1. (System Variables Classification)
Let $Design^{A}_{TLM}$ be an AsmL model and $\mathcal{V} = \{v_1, v_2, \ldots v_s\}$ a set of its state variables. We classify $\mathcal{V}$ into three subsets:

$$\begin{aligned}
\mathcal{V}_{in} &= \{v \in \mathcal{V} \mid v \text{ is an input variable}\}\\
\mathcal{V}_{out} &= \{v \in \mathcal{V} \mid v \text{ is an output variable}\}\\
\mathcal{V}_{int} &= \{v \in \mathcal{V} \mid v \text{ is an internal variable}\}
\end{aligned}$$

In our context, the search space to be explored is the assertion's state space A_{sp} (see Definition 7.4.1). Candidate solutions are finite sequences of input ranges and probability weights. Each candidate solution is identified by a unique chromosome (a finite string of bits). The information encoded in the chromosomes is composed of: (1) a list of input variables; (2) their domains (see Definition 7.5.2), and (3) a probability distribution of the domain (see Definition 7.5.3).

Definition 7.5.2. (Variable Domain)
Let $Design^{A}_{TLM}$ be an AsmL model and $\mathcal{V}_{in}$ its input variables set. To each variable $v \in \mathcal{V}_{in}$, there is a corresponding domain d.

Definition 7.5.3. (Variable Domain Distribution: p)
Let $Design^A_{TLM}$ be an AsmL model and $\mathcal{V}_{in}$ its input variables set. Then, for every variable $v \in \mathcal{V}_{in}$ corresponds a function p providing the variable's values distribution over its domain.

Definition 7.5.4. (Chromosome Encoding)
Let $Design^A_{TLM}$ be an AsmL model and $\mathcal{V}_{in}$ its input variables set. Then, the test generator's chromosome is the set $Chrom = \langle \mathcal{V}_{in}, D_i, P_i \rangle$.
where:

- $D_i = \{d_1, d_2, \ldots d_s\}$ is the set collecting all the variables domains
- $P_i = \{p_1, p_2, \ldots p_s\}$ is the set collecting all the variables distribution.

The chromosome encoding is the most important aspect of our algorithm. The variable values generation is controlled by their domains distributions. For example, for a variable of type *Integer*, we can use the following chromosome encoding:

- Variable Domain $d = [-2^{16}, 2^{16} - 1]$
- Variable Domain Distribution:
  - $\mathrm{p}([-2^{16}, 0[) = 0.3$
  - $\mathrm{p}([0, 2^{16} - 1]) = 0.7$

7.5.2 Fitness Criteria

The proposed fitness criteria serves to guide the genetic search towards covering the whole assertion's state space. The intuitive idea is to modify the shape of the variable's domain distribution, p, in order to accomplish a better coverage. For the sake of improving the efficiency of the algorithm, we keep track of the best and worst chromosome fitness in each generation; if both fitness values become equal, we increase the mutation rate, in order to help the genetic evolution get out of local maxima. Once there is an improvement in the overall fitness, we restore the original mutation rate to continue the evolution normally.

Definition 7.5.5. (Test Vector Generator: T_{Gen})
Let $Design^{A}_{TLM}$ be an AsmL model and $\mathcal{V}_{in}$ its input variables set. Then, a test vector generator is defined by a unique chromosome encoding *Chrom*.

For every coverage, there is a corresponding fitness function. For state coverage, S_{cov} (see Definition 7.4.2), the fitness function is given by Definition 7.5.6 where the fitness identifies the best test generator by checking for the one having a maximum state coverage.

Definition 7.5.6. (Fitness Criteria for State Coverage: F_{Scov})
Let $Design^{A}_{TLM}$ be an AsmL model, $\mathcal{V}_{in}$ its input variables set, A be an assertion, A_{sp} its space state and $\mathcal{T} = \{T^1_{Gen}, T^2_{Gen}, \ldots T^n_{Gen}\}$ a set of n-test generators. Then, the fitness criteria corresponding to state space coverage is:

$$F_{Scov} = \max_{T^i_{Gen} \in \mathcal{T}} (S_{cov}) = \max_{T^i_{Gen} \in \mathcal{T}} (\frac{Card(T^i_{sp})}{Card(A_{sp})})$$

where T^i_{sp} is a sequence of test vectors generated by T^i_{Gen}.

The fitness criteria corresponding to the transition state coverage is provided in Definition 7.5.7.

Definition 7.5.7. (Fitness Criteria for Transition Coverage: F_{Tcov})
Let $Design^{A}_{TLM}$ be an AsmL model, $\mathcal{V}_{in}$ its input variables set, A be an assertion, A_{sp} its space state and $\mathcal{T} = \{T^1_{Gen}, T^2_{Gen}, \ldots T^n_{Gen}\}$ a set of n-test generators. Then, the fitness criteria corresponding to transition state space coverage is:

$$F_{Tcov} = \max_{T^i_{Gen} \in \mathcal{T}} (T_{cov}) = \max_{T^i_{Gen} \in \mathcal{T}} (\frac{Card(T^i_{tp})}{Card(A_{tp})})$$

where T^i_{sp} is a sequence of test vectors generated by T^i_{Gen}.

The genetic mutation operation updates the set of test generators, $\mathcal{T} = \{T^1_{Gen}, T^2_{Gen}, \ldots T^n_{Gen}\}$, according to the coverage results. We propose to deduce the new population of generators using the following operations: inheritance, mutation and recombination. We keep track of all the populations using a unique sequence Seq_{TGen}.

Definition 7.5.8. (Generations of Test Generators: Seq_{TGen})
Let $\mathcal{T}_i$ be a set of test generators. Then, the sequence of generations of test generators is:

$$Seq_{TGen} = \{\mathcal{T}_1, \mathcal{T}_2, \ldots, \mathcal{T}_i, \ldots, \mathcal{T}_m\}$$

where:

- $\mathcal{T}_1$ is the initial generation.
- $\forall i \mid 1 < i \leq m$, $\mathcal{T}_i$ is the updated generation obtained from $\mathcal{T}_{i-1}$ by applying inheritance, mutation and recombination operations.

The convergence of the algorithm to a better solution w.r.t. the fitness criteria is granted if the sequence Seq_{TGen} is increasing w.r.t. an order based on the coverage. Such a result cannot be derived for a general case. It requires defining precisely: (1) the variables domains; (2) the variables domains distributions; and (3) the inheritance, mutation and recombination operations. In general though, using a simple uniform distribution and preserving the best generator from the previous generation grants that the sequence will not be decreasing.

7.5.3 Coverage Preserving From TLM to RTL

The most critical step in our methodology of Figure 7.1 concerns the generation of the RTL implementation from the TLM models. We propose to derive manually a synthesizable RTL implementation from a SystemC TLM design. Generally, it is better to start with the computation units. The RTL design includes flip-flop descriptions, register size and clock cycle accurate controls. Then, communication protocols can be refined. This process employs conventional ports, using signals like standard logic or Boolean. Communication protocols are implemented internally to each channel, and the synchronization is carried out using clock signals. The abstraction of data types cannot be used anymore. From a practical point of view, it is more efficient to focus on the problem using a hierarchical approach. It is useful, for example, to concentrate on refining modules and then refining communication, as done here.

Lowering the level of abstraction to RTL may modify the coverage results gained at transaction level. Theorem 7.5.1 guarantees the preserving of the coverage if both FSM's resulting from the RTL and TLM models are bisimulate.

Theorem 7.5.1 *(Coverage Preserving)*
Let $Design_{TLM}$ *be a TLM model and* $\mathcal{F}_{TLM}$ *its generated FSM. Let* $Design_{RTL}$ *be the RTL implementation obtained from* $Design_{TLM}$ *and* $\mathcal{F}_{RTL}$ *its generated FSM.*

If there exists a bi-simulation relation, α*, between* $\mathcal{F}_{TLM}$ *and* $\mathcal{F}_{TLM}$*, then, for any assertion A over the design the following two equalities hold:*

1. $S_{cov}^{TLM} = S_{cov}^{RTL}$

2. $T_{cov}^{TLM} = T_{cov}^{RTL}$

where:

- S_{cov}^{TLM} *and* T_{cov}^{TLM} *refer to TLM state coverage and TLM transition coverage, respectively.*

- S_{cov}^{RTL} *and* T_{cov}^{RTL} *refer to RTL state coverage and RTL transition coverage, respectively.*

Proof 7.5.1. (sketch)
The existence of a bi-simulation induces similar assertion's state space for both TLM and RTL. Therefore, the coverage metrics will be preserved in the transformation.

Theorem 7.5.1 offers a sufficient condition to guarantee coverage preserving. However, in the general case, proving a bi-simulation relation is not an easy problem. Hence providing a lighter condition to guarantee coverage preserving is quite important. As a future work, we consider defining a preserving rule based on traces, for example. Furthermore, we aim to provide translation rules enabling a by-construction coverage preserving RTL implementation.

7.6 Application: LA-1 Interface

New IPv6 systems and carriers increasingly demanding detailed lookups on packets and flows, the interface between network processors and other components, such as external co-processors and memory devices, is taking the spot light in the networking sector. Currently, the Look-Aside 1 (LA-1) interface [79] is the de-facto standard for linking these components [7][1]. It is being the key to several networking-specific applications, including packet forwarding, packet classification, admission control, and security.

7.6.1 Coverage Results

In order to evaluate our proposed methodology, we considered a set of assertions. Table 7.1 (Table 7.2) compares the assertion state coverage (assertion transition coverage) results obtained with:

- Blind random test generation of the TLM SystemC code.
- Guided simulation using a test generator obtained after 30 iterations of the GA.
- Guided random test generation of the RTL SystemC code.
- Blind random test generation of the RTL SystemC code.

[1]The technical description of the LA-1 is provided in Appendix C.5.

Table 7.1: State Space Assertions' Coverage Analysis

Assertion	A1	A2	A3	A4	A5
Rand. TLM (%)	10	8	4	12	14
GA TLM (%)	64	72	66	82	55
Guided SystemC RTL(%)	61	71	75	81	56
Rand. SystemC RTL (%)	5	3	6	7	4
Rand. RTL (Specman) (%)	12	11	14	17	9

Table 7.2: Transitions Space Assertions' Coverage Analysis

Assertion	A1	A2	A3	A4	A5
Rand. TLM (%)	15	17	13	12	15
GA TLM (%)	51	55	52	48	47
Guided SystemC RTL(%)	45	44	42	31	33
Rand. SystemC RTL (%)	4	3	5	6	4
Rand. RTL (Specman) (%)	12	11	8	7	13

- Random test generation of the RTL Verilog code using Speman Elite.

We used 10^9 functional calls and 10^9 simulation cycles for the TLM and RTL models, respectively. We iterated the genetic algorithm for 30 generations (each with 10^9 tests). We used a uniform variables distributions over the variables domains.

7.6.2 Discussion

At the transaction level, our proposed genetic algorithm provided an enhanced coverage in comparison to the blind random simulation by a factor of five to seven. The value of the coverage vary according to the assertion. When applying the GA, we noticed that it takes relatively quick progress in the beginning stages of evolution. We also noted that there exist some phases, where the algorithm hits local maxima before mutating further, which improves its performance. We even noticed that the coverage sometimes decreases slowly from generation to generation due to the fact that the evaluation of the assertion is based on weighted random generation. In other terms, since the number of tests is finite, a generator may have two different coverage results for two different test trials.

We used the final test generator obtained from the GA algorithm procedure at TLM to verify the same assertion at RTL. Coverage results were comparable. Random simulation at RTL provided very low coverage results due to a larger system state space at this level. By defining a suitable environment using the e-language [106], we succeeded to improve the coverage results in comparison to the blind random simulation. Nevertheless, the coverage remained low in comparison to our GA algorithm by a factor of four to five for all the assertions. We also noticed that the execution time using TLM SystemC was very fast (a factor of 50 to 100) in comparison to the simulation of the Verilog implementation with Specman Elite.

7.7 Summary

In this Chapter, we presented a methodology to enhance assertion coverage using transaction level models as intermediate step in the design process. We used AsmL as a TLM language for the sake of automatically generating a finite state machine of the system. We defined assertions as a set of states (part of the system's FSM). We introduced two assertion coverage metrics: state and transition coverage. We provided a construction technique: (1) for a *connected* FSM an optimal test sequence that covers the whole assertion's state space (Theorem 7.4.1); and (2) for *clique* FSM an optimal test sequence that covers the whole assertion's transition space (Theorem 7.4.2).

In the second part of the Chapter, we proposed a genetic algorithm to enhance the coverage. Our genetic algorithm, when applied to the Look-Aside Interface standard (LA-1) as an application, showed an improvement of the assertions coverage by a factor of seven in comparison to blind random simulation and a factor of four in comparison to a guided simulation using Specman Elite of Verisity.

Chapter 8

SystemC Verification Methodologies

8.1 Introduction

In this chapter we introduce two applications of our proposed SoC verification framework to SystemC. In the first proposed methodology, *top-down*, the verification is integrated as part of the design process starting from the behavioral specification of the system. The final product is a by-construction correct SystemC model w.r.t. verified properties. In the second proposed methodology, *bottom-up*, we consider the verification of existing designs modelled in SystemC.

8.2 Top-Down Design for Verification

Our proposed top-down design methodology, as displayed in Figure 8.1, includes two parallel paths concerning the design and its properties. We model the design in the classical way a C++ design is modeled using UML (i.e., using use cases, class diagrams, etc.) Then, we translate the UML model to ASM in order to perform model checking of certain properties. These latter are extracted from the UML sequence diagram and encoded in the PSL syntax. The verification process leads to: (1) a completion either with a success or failure of the property; or (2) a state explosion. The UML update and UML to ASM translation tasks are repeated until all the properties pass (either succeeds or do not complete). Then, we compile the PSL properties into a set of C# classes, using the AsmL tool to be used as assertion monitors. The design in ASM is, from the other side, translated to C++ (SystemC model) and co-integrated with the assertions for verification by simulation.

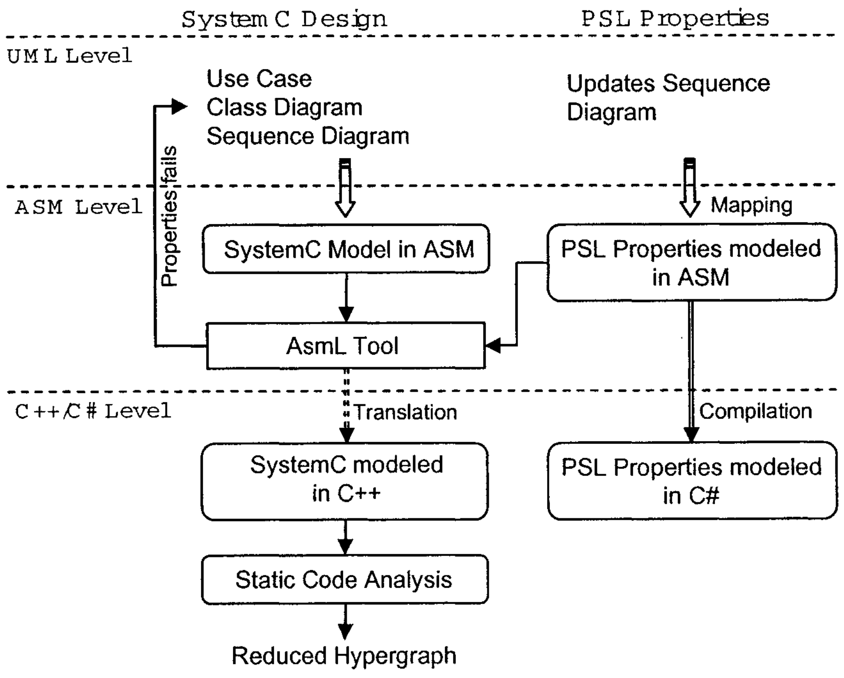


Figure 8.1: Top-Down Design and Verification Methodology.

8.3 Bottom-Up Verification Methodology

In our bottom-up verification methodology, as displayed in Figure 8.2, we perform the model checking of SystemC by translating the original design to an intermediate representation that omits all the details of the SystemC simulator. The target (or transformed) program is modeled in AsmL to be cross-produced with the system properties that will be verified over the whole system's state space. To model the properties, we use the property specification language. Properties are embedded in the design as external monitors; hence, they can be used as stand-alone IP block(s) to validate other devices, either at the AsmL level by model checking or at the SystemC level by assertion based verification.

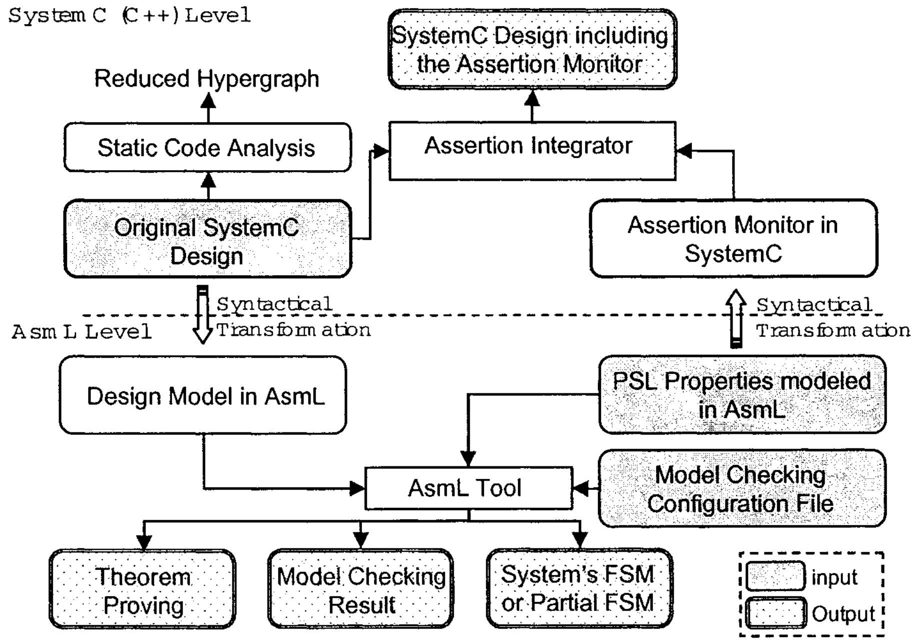

SystemC (C++) Level
Reduced Hypergraph
SystemC Design including the Assertion Monitor
Static Code Analysis
Assertion Integrator
Original SystemC Design
Assertion Monitor in SystemC
Syntactical Transformation
AsmL Level
Syntactical Transformation
Design Model in AsmL
PSL Properties modeled in AsmL
AsmL Tool
Model Checking Configuration File
Theorem Proving
Model Checking Result
System's FSM or Partial FSM
input
Output

Figure 8.2: Bottom-up Verification Methodology.

8.4 Static Code Analysis

As a solution to the SystemC (as an SLL language) verification problem we propose to define an abstract environment that can be used for: (1) the analysis and verification of SystemC programs, (2) abstract debugging and (3) possible interfacing with model checking and simulation. The analysis of the design is, as defined in [26], based on approximate semantics of programs to provide sound answers to questions about their run-time behaviors. The abstract debugging will be possible thanks to the abstraction of the memory (allocation blocks and the stack), the language simulation manager, component responsible for running the simulation, the events' stack and to the code of the program itself. Both the program execution environment as well as the simulation environment will be represented in order to allow abstract execution of the program.

In order to interface to abstract interpretation with model checking (i.e. feed the abstracted code into a model checker) objects' and events' aspects of SoC designs need to be translated into a procedural like code. Eventually this may seem to be not always feasible since we are starting from an object-oriented program structure. However, the approach can still be valid when restricted to some parts of the program to verify local properties.

8.5 Modeling PSL Properties

PSL is an implementation independent language to define properties. PSL is a hierarchical language, where every layer is built on top of the layer below. This approach allows the expressing of complex properties from simple primitives.

8.5.1 UML Model

Using UML as a high level of abstraction for design showed a lot of success when applied to software. Main proposals consider either to use UML as new system level design [30] or as top layer in combination with existing languages (such as SystemC) [94]. Nevertheless, these proposals neglected totally to consider the properties of the system (PSL like properties in particular) while sequence diagrams for example includes very useful information to set transaction properties for TLM in particular.

Unfortunately, sequence diagrams do not allow a direct mapping to PSL due to two reasons: (1) the complexity of the PSL property which may include temporal operators; and (2) the need for instantiation in the PSL. In fact, PSL was defined for real instances from the design formed from objects while the sequence diagram considers only classes. For these facts, UML will not present completely and precisely all PSL property. However, it can be used to provide a general skeleton of the property that could be refined and instantiated at the ASM level.

In order to make the UML sequence diagram more adequate for PSL representation, we introduced the following operators:

Clocks: we use the operator to specify the clock that activates the current action (if there exist one).

Number of cycles: every action can be include the information about after how many cycles the method is start executing (e.g., *Mtd[5]()* says that the method *Mtd* is executed for exactly 5 consecutive cycles).

Temporal operators: these includes operators specifying if the method will be Always executed (A), Eventually executed (E), executed Until a condition is fulfilled (U), etc. These, in fact, represent a mapping to the PSL temporal operators (second layer of PSL).

Sequence operations: includes information about the order of executing certain sequences (e.g., *next*, *prev* etc.)

Text output: refers to a message that is displayed in case the method fails. This is included in PSL to track the progress of the assertion based verification.

Method duration: certain methods are supposed to execute for a certain number of cycles (for e.g., reading from memory may take 4 cycles). So, we added an operator $ to specify such an information.

Figure 8.3 gives an example of a sequence diagram describing a PSL property saying that if a bus sends a new request, then in the next cycle the arbiter will be notified and will make the arbitration. In the third cycle, the master starts sending. The bus is released in the fourth cycle and a notification will be sent, eventually, by the slave to the bus who will forward it in the next cycle to the master.

When mapping to ASM the UML sequence diagram needs to be instantiated according to the design objects. For instance we need to specify, for example, that the notification must be to the original master and not to all the masters.

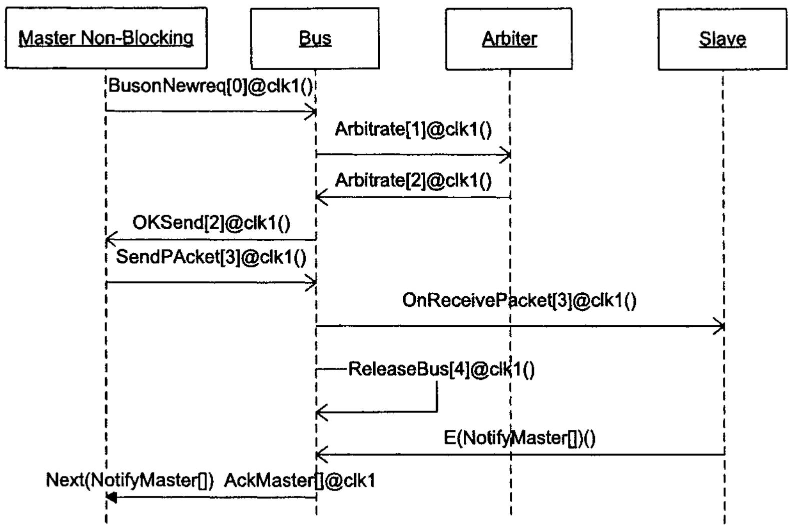


Figure 8.3: Example of a Modified UML Sequence Diagram.

8.5.2 ASM Model

We embed the PSL properties into the design as external monitors. Such an approach enables reusing PSL properties at lower design levels since the AsmL tool can automatically compile them into a C# or .NET code, which can be compiled and executed with the concrete SystemC level or as a stand-alone module. A detailed description of the PSL embedding in AsmL is provided in Appendix B.

8.6 Modeling SystemC

8.6.1 ASM Model

The design model at the ASM level is purely object-oriented where every class includes a set of parameters and methods. The particularity of this model resides in the fact that it will be used to generate an FSM using the reachability algorithm [47] embedded AsmL tool [74]. So, a specific style of programming is required in addition to a precise configuration of the algorithm. Before going further into the details of the

ASM model, we will discuss the FSM generation algorithm which will give a better idea about the requirements of the ASM model.

The FSM generation algorithm requires as input: domains, methods, actions and variables (optional inputs are filters, action groups and properties). The transitions in the FSM are the method calls (including argument values) in the test sequences. The methods in the model program that appear in the transitions are called actions. The states in the FSM are determined by the values of selected variables in the model program, called state variables. The algorithm generates the FSM by executing the model program in a special execution environment, keeping track of the actions it performs and recording the states it visits. This process is called *exploration*. Usually a model program implies so many states and transitions that it is not feasible to include them all in the FSM, so it is necessary to limit the number of states and transitions that the tool explores.

The final FSM will, according to the algorithm's configuarion options, represent only a portion – an under-approximation – of the huge FSM that would result if the model program could be explored completely. Critical information to ensure the correctness of certain properties concerning identifying the actions and state variables in the FSM. Domains, finite collections of values from which method arguments are taken, are defined in order enable better coverage on particular issues. Filters express stopping conditions that limit exploration (used to stop the FSM generation if a property fails, for e.g.).

For a model M including a set of classes, $C = \{c_1, \ldots, c_n\}$, where n is the total number of classes in M. For every class c_i in C, we denote its set of methods by c_i^{mth} and the set of members by c_i^{mem}. We defined a set of rules (called R_{FSM}) to guarantee the generation of an FSM representing a portion of the complete system's FSM; these include:

Rule R^1_{FSM}: For every class c_i in C, we have to define a list of instantiations of the class. This ensures that the algorithm will not through an exception.

Rule R^2_{FSM}**:** The firstly executed method in the design must verify that all the objects from the class domains were correctly instantiated. This ensures that the algorithm will not misbehave.

Rule R^3_{FSM}**:** For every class c_i in C, every method in c_i^{mem} must include a list of *pre-conditions* to specify when the algorithm considers this method in the exploration process. This ensues that in every state we only explore the involved methods.

Rule R^4_{FSM}**:** For every class c_i in C, domains for all members in c_i^{mem} must be inherited from AsmL types and restricted to the possible values the system can accept (in particular for inputs). This will allow exploring known types and limits the risks of state explosion.

The optimal scenario is to explore all the methods and domains in the model; nevertheless, this is not possible all the time due to the state space explosion. For this reason, working carefully the domains and the set of actions is the very critical path in the FSM generation process. For illustration purpose, Figure 8.4 shows a generic ASM model with a method including a precondition (denoted by the *require* keyword) setting that the method needs the system to be initialized (*SystemInit* = *true*) and that it has both variables *m_gnt* and *m_req* set to *false* before it can be executed. Such conditions define strictly at which state the system can execute a particular set of actions.

```
class PCI_Arbiter
 private var m_ActiveMaster as Integer = -1
 private var m_req as Boolean = false
 private var m_gnt as Boolean = false
 public PCI_Arbiter()
 public PCI_ArbiterUpdate_m_req()
  require (SystemInit = true) and me.m_gnt = false and me.m_req = false
  me.m_ActiveMaster := min id | id in Masters_Range where
                        (MASTERS(id).m_req =  true)
  me.m_req := true
```

Figure 8.4: An Example of an ASM Model.

8.6.2 Translation to C++

Once the ASM model verified using the properties describing its behavior, we translate it to SystemC according to a set of rules to ensure that the final SystemC model preserves the original ASM code properties. The transformation is purely syntactical, it is performed to certain rules (that we call R_{C++}) that could be summarized in the following:

Rule R^1_{C++}: "Basic types": ASM basic types are all mapped to their equivalent SystemC types (e.g. *Integer* to *int*, *Byte* to *unsigned char*, etc.). AsmL includes the same types as C++ which are used for SystemC also.

Rule R^2_{C++}: "Class Translation": this includes two separate rules for variables and methods:

Rule $R^{2.1}_{C++}$*:* "Class Members": are translated into SystemC signals having the same basic type. For example, *var m_val as Integer* is translated to *sc_signal<int> m_val.*

Rule $R^{2.2}_{C++}$*:* "Class Methods": in ASM contain two parts which are the post-/pre-conditions and the method core. The first part is integrated in the SystemC module's *constructor*. For instance, a method *Send* defined in ASM with the following pre-condition *require clk=true* is inserted in the SystemC module constructor area as "*SC_THREAD(Send); sensitive << clk;*". The method core is integrated as it is in the SystemC module (we just modify the basic types according to the Rule 1).

Rule R^3_{C++}: "Global Modules": are integrated in the SystemC's main procedure *sc_main*. The naming mapping is used to link different modules together.

8.7 Verification Techniques

The verification process is decomposed into two parts: (1) by model checking at the ASM level; and (2) by assertion based verification at the C++/C# level.

8.7.1 Model Checking

PSL properties are embedded in ASM as assertions, the assertion here means the validity of the property. It provides a unique view of the property in every system's state. It also simulates the design with the property as a monitor. We build the assertion starting from basic Boolean components, sequences, and then verification units. We encapsulate sequences in the verification unit as an assertion which is embedded in the design. Given a set of Boolean items $x_1, x_2, \ldots, x_n$, and $y_1, y_2, \ldots, y_m$ belonging to the Boolean layer, and the sequences, S_1 and S_2 belonging to the temporal layer, we can define: $S_1 = \{x_1, x_2, \ldots, x_n\}$, and $S_2 = \{y_1, y_2, \ldots, y_m\}$ and then use assertions to check any PSL operation between S_1 and S_2 such as $S_1\ OP\ S_2$, where OP is a PSL operator (e.g., implication (:), or equivalence ($\Leftrightarrow$)). The assertion is built as follows:

1. Add all the Boolean items to the sequences:

   $\forall\ i\ in\ 1\ to\ n:\ S_1.AddElement(x_i)$

   $\forall\ j\ in\ 1\ to\ m:\ S_2.AddElement(y_i)$

2. Create the property: $P\ := S_1\ OP\ S_2$

3. Define the *verification unit* as an assertion, say A, that includes the above property: $A.Add(P)$

This property is embedded in every state in the FSM generated by the AsmL tool and is represented by two Boolean state variables *P_eval* and *P_value* (saying, respectively, if the property can be evaluated and the value of the property in the current state). A violated property is detected once *P_eval = true and P_value = false.* We set the previous condition as filter for the FSM generation algorithm. This way the generation stops when an error is detected. The generated portion of the state machine, at this point, can be used to identify the problem through a scenario of a counter-example. For multiple properties, the filter is set as conjunction of all the conditions for the separate properties. This technique minimizes radically the number of the state variables (the FSM size and its generation time). A correct verification process results on the generation of the system's FSM (according the configuration file constraints).

8.7.2 Assertion Based Verification

We target to add the assertion as an external monitor to the SystemC design. We consider three steps:

(1) Updating the SystemC design to interface to the assertion.

(2) Generating the assertion (in C#) from its ASM description.

(3) Integrating the assertion in the design.

The translation to C# of the PSL assertions embedded in ASM is a matter of compilation using the AsmL tool. Most of the effort is spent in updating the SystemC design to get it connected to the assertion monitor. For instance, we validate the assertion syntactically by generating the list of its involved variables. Then, we perform a type check to make sure the variables are well instantiated in the SystemC design. For instance, the signals (variables) that are used in the assertion must be seen as external signals so that they can be input to the assertion monitor. So, we modify the SystemC design to make the required variables visible to the monitor. This transformation does not affect the behavior of the code as it will only be accessed in a read-only mode.

Once the design is updated, we add the required instantiation of the assertion to bind it to the existing SystemC design modules. The assertion monitor, acting as part of the design, can do the following: (1) stop the simulation when the assertion is fired; (2) write a report about the assertion status and all its variables; and (3) send a warning signal to other modules (if required). We note that the internal code of the assertion is C# so the designer can update it or do any other functionalities that can be coded in C#.

8.8 Summary

In this chapter, we first presented a top-down methodology to design and verify SystemC transactional models starting from a UML system specification and integrating an intermediate ASM layer. We proposed to upgrade the UML sequence diagram in order to capture transaction related system properties. Then both of the design and its properties are modeled in ASM to enable performing model checking. On the other hand, to cover for the state explosion problem that may result due to the system's complexity, we completed our approach by offering a methodology to apply assertion based verification re-using the already defined PSL properties. To do so, we defined a set of translation rules to transform the design's model in ASM to its implementation in SystemC.

We also presented a bottom-up approach where starting from an existing SystemC design we generate internally a model in AsmL, an Object-Oriented language used to model systems, and verify the system property at the ASM level.

Chapter 9

Conclusions and Future Work

9.1 Conclusions

In this thesis, we presented a framework for the verification of system level languages, where we considered SystemC (the future IEEE standard for system level languages) as an application. We proposed two complementary methodologies: (1) top-level: where we start from a behavioral specification of the design aiming to produce a verified SLL design; and (2) bottom-up: where we verify existing SystemC designs. Both methodologies share an intermediate layer for the verification where the design is modeled in variant of Abstract State Machines (ASM) languages (AsmL).

The proposed verification framework includes four techniques:

1. Static code analysis using abstract interpretation: We extended the abstract interpretation framework to support SystemC semantics. We also provide a graphical environment for the abstract analysis and debugging of SystemC.

2. Model checking: We adapted an existing state exploration algorithm to enable SystemC model checking.

3. Assertion based verification: We provided two techniques to integrate SystemVerilog and PSL assertions to SystemC.

4. Guided simulation: We provided a formalization of the functional coverage. We also proposed a genetic algorithm based technique to enhance the coverage using guided simulation.

In order to prove the soundness and the correctness of our proposed verification framework: first, we defined a generic trace semantics for both SystemC and AsmL; then, we proved the soundness and the completeness of this semantics; and finally, we established the correctness of the SystemC to AsmL transformation.

9.2 Future Work

The methodology presented in this thesis opens new avenues in using formal methods for the verification of system level languages. Furthermore, it presents a first step towards combining both formal techniques to simulation for better coverage of real yet complex designs.

- Design and implement a compiler to generate and manipulate automatically the hypergraph structures.
- Develop reduction techniques to extract invariants from hypergraphs.
- Improve the state exploration algorithm in order to tackle the state space exploration problem.
- Provide a formal link between the transaction level and register-transfer levels functional coverage.
- Optimize the genetic algorithm by finding optimal variables domains and distributions over theses domains.
- Investigate the verification of complex SoCs using our proposed methodology.

Appendix A

System Level Languages

This Appendix overviews state-of-the-art System Level Languages (SLL) proposals. We consider a classification composed from four main classes: (1) reusing existing software SLL; (2) extending classical hardware languages; (3) readapting software languages; and (4) creating new languages specified for system level design.

A.1 Reuse of Existing Hardware Languages: SystemVerilog

SystemVerilog [2] is a radically revised Verilog [57] language reaching toward much higher levels of abstraction. SystemVerilog blends Verilog, C/C++ and Co-Design Automation's SuperLog [96] language to bring unprecedented capabilities to chip designers.

SystemVerilog is a set of extensions to the IEEE 1364-2001 Verilog HDL to aid in the creation and verification of abstract architectural level models. The key features include interfaces that allow module connections at a high level of abstraction; C-language constructs such as global; and an assertion construct that allows property checking. SystemVerilog includes the synthesizable subset of SuperLog, and its assertion capability will likely be derived from the Design Assertion Subset that

Co-Design and Real Intent Corp. recently donated to the Accellera standards body.

With all these enhancements, SystemVerilog may remove some of the impetus from C-language design, at least for register-transfer-level chip designers. The basic intent is to give Verilog a new level of modeling abstraction, and to extend its capability to verify large designs.

SystemVerilog borrows the C-language "char" and "int" data types, allowing C/C++ code to be directly used in Verilog models and verification routines. Both are two-state signed variables. Other new constructs include "bit" a two-state unsigned data type, and "logic" a four-state unsigned data type that claims more versatility than the existing "reg" and "net" data types.

The slogan of the people supporting Verilog is: "The right solution is to extend what works". However, this may face a harsh critic from the people supporting C++ based solutions and who advocate that Verilog or SystemVerilog or whatever is the Verilog based name will not offer a shorter simulation time. The debate is quite hard and the last word will not be said soon!

A.2 C/C++ Based Approach

Some optimistic software designers are supporting the establishment of C/C++ [38] or even Java [56] based languages for future SoC designs. They think that, over time, improvements in automatic methods and increases in hardware functionality will extend the pieces handled automatically to where whole designs can be implemented by "cc -silicon". In the near term, however, tools are not going to be good enough and manual refinement will be required. This means that the version of the language that will be used must allow for the expression of hardware at lower levels, not only at the algorithmic level.

Over the past decade, several different projects have undertaken the task of extending C to support hardware [31], including SpecC [38] at the University of

California, Irvine, HardwareC [68] at Stanford University, Handel-C [83] at Oxford University (now moved to Embedded Solutions Ltd.), SystemC++ [69] at C Level Design Inc., SystemC [81] at Synopsys Inc., and Cynlib [29] at CynApps.

This variety of projects falls roughly into two complementary categories. The first, exemplified by SpecC, has focused on adding keywords to the basic C language, supporting hardware description at a high level as a basis for synthesis. The second, exemplified by SystemC, exploits the extensibility of C++ to provide a basic set of hardware primitives that can be easily extended into higher level support [88]. These two complementary approaches span all levels of hardware description (algorithmic, modular, cycle-accurate, and RTL levels).

C-Based Solutions

As a C-based solution, we will consider the case of SpecC and HardwareC.

SpecC

The SpecC language [38] is defined as extension of the ANSI-C programming language. This language is a formal notation intended for the specification and design of digital embedded systems including hardware and software. Built on top of ANSI-C, the SpecC supports concepts essential for embedded systems design, including behavioral and structural hierarchy, concurrency, communication, synchronization, state transitions, exception handling and timing.

To defend SpecC against C++ proposals and mainly SystemC, when this latter was first introduced in 1999, SpecC supporters advanced that SystemC is primarily aimed at simulation, however, SpecC was developed with synthesis and verification in mind. They also considered that SystemC targets RTL design, but SpecC is a system-level design language intended for specification and architectural modeling. Nevertheless, after the release of the version 2.0 of SystemC, all these argument were broken. In fact, SystemC is nowadays supporting most system level design requirements.

HardwareC

Under the same class of C-based languages we find also a language called HardwareC [68]. This is a language that uniformly incorporates both functionality and design constraints. A HardwareC description is synthesized and optimized by the Hercules and Hebe system [67], where tradeoffs are made in producing an implementation satisfying the timing and resource constraints that the user has imposed on the design. The resulting implementation is in terms of an interconnection of logic and registers described in a format called the Structural Logic Intermediate Format.

HardwareC attempts to satisfy the requirements stated above. As its name suggests, it is based on the syntax of the C programming language. The language has its own hardware semantics, and differs in many respects from C. In particular, numerous enhancements are made to increase the expressive power of the language, as well as to facilitate hardware description [68].

C++-Based Solutions

As any real system a SoC is composed by a number of entities and objects interacting together. This is the reason for the limitation of C-based proposals. By reference to software design, languages that can be used at the system level have to be preferably object-oriented. That is why nowadays more mature proposals are based on C++. We will discuss mainly two C++ based proposals: Cynlib and SystemC.

Cynlib

Cynlib [29] provides the vocabulary for hardware modeling in C++. It is a set of C++ classes which implement many of the features found in the Verilog and VHDL hardware description languages. It is considered as a "Verilog-dialect" of C++, but it is more correct to say that it is a class library that implements many of the Verilog semantic's features. The purpose of this library is to create a C++ environment in which both hardware and testing environment can be modeled and simulated.

Cynlib supports the development of hardware in a C/C++ environment. To do this, Cynlib extends the capabilities of C/C++ by supplying many features such as: concurrent execution model, cycle-based, modules, ports, threads, etc.

SystemC

SystemC [80] comes to fill a gap between traditional HDLs and software development methods based on C/C++. Developed and managed by leading EDA and electronics companies, SystemC comprises C++ class libraries and a simulation kernel used for creating behavioral- and register-transfer-level designs. Combined with commercial synthesis tools, SystemC can provide the common development environment needed to support software engineers working with C/C++ and hardware engineers working in HDLs such as Verilog or VHDL.

New releases of Cadence Design Systems Inc.'s Signal Processing Worksystem (SPW) [28] software and Axys Design Automation Inc.'s MaxSim Developer Suite increase support for SystemC. Mentor Graphics Corp. has also added a C language interface to its Seamless hardware/software co-verification environment [43] that lets designers use mixed C/C++ and HDL descriptions for hardware. The interface, called C-Bridge, will be included with Seamless version 4.3, which is to be released in late March 2003 [43].

A.3 Java-Based Proposals

While there has been some discussion about the potential of Java as a system-level language or high-level hardware description language, LavaLogic may be the first commercial EDA provider to bring that option into contemporary design systems with a language called JHDL [6]. LavaLogic's Java-to-RTL compiler is an "architectural synthesis" tool that turns Java into synthesizable HDL code. LavaLogic is offering a tool that will take high-level Java descriptions down to gate-level netlists, starting with FPGAs [56].

According to Java advocates, Java appears to be the purest language to solve the productivity problems currently at hand. They also claim the language can express high-level concepts with far less code than today's HDLs, yet offer more support for concurrency than C or C++.

The main point all Java advocates stressed in comparing their approach to the C/C++ based ones is the concurrency. In fact, a classic problem with C and C++ is their inherent inability to express concurrency. In Java, in contrast, concurrency can be explicitly invoked with threads. Nevertheless, this unique criterion for comparison is not enough to balance the choice C/C++ to Java. Java can be classified as the "next best" choice after C++, since it does not support templates or operator overloading, resulting in a need for numerous procedure calls.

A.4 Developing a New Language: SuperLog

SuperLog [37] is a Verilog superset that includes constructs from the C programming language. Because of its Verilog compatibility, it has earned good reviews from chip designers who express considerable skepticism about C language hardware design. SuperLog holds great promise for the next three to four years. It is the case mainly because SuperLog does not require a radical shift in methodologies and allows groups to retain legacy code.

SuperLog (sometimes said to be "Verilog done well!" [92]) combines the simplicity of Verilog and the power of C, and augments the mix with a wealth of verification and system features [37]. For now, we have Verilog 2001 at the lower end of the sophistication spectrum ("Jolly nice but lacking a lot of features" [73]) and SuperLog at the other end ("incredibly powerful, but as yet falling well short of industry-wide support" [73]). In a certain way, as defined, SuperLog is a smart idea to deal with system level design. In fact, SuperLog utilizes the power of C with the simplicity of Verilog to provide the right balance for productive design

A.5 Discussion

System-on-a-Chip design, as an open project, initiated a debate between hardware and software communities. The real issues for embedded chip design are increasingly embedded software issues. Three main trends are proposed to design SoC. The software based methodologies consider that basing everything from the HDL approach is starting from the wrong point. On the other side, hardware designers are looking into upgrading Verilog to support system level design requirements. The third group is supporting the idea of defining new languages designed from scratch for system level design. Nevertheless, this last approach is facing very hard critics because most designers think that the idea of a grand unified language is fundamentally flawed. The best choice will be to have multiple languages targeting specific areas. These languages will be easier to learn than a grand language and they will also be easier for tools to parse.

Whatever the syntax looks like C, Verilog, VHDL or UML, most options share a common goal: the transition of functional design from the minutiae of Boolean logic, wiring and assembly code up to the level of the designer's issues. Thus, functionality becomes modeled as an architectural interaction of behaviors, protocols and channels. This is great, except that most languages handle only digital functionality and cannot comprehend trade-offs of power, timing, packaging or cost.

There are also shortcomings with today's methodology in the face of the growing use of SoCs. Individual designs that were once an entire system are now blocks in an SoC. The associated verification strategy with many of these blocks was not designed to scale up to a higher integration level. If the methodology does not permit easy integration of block-level verification code into a system-level SoC environment, then the verification task will become a major bottleneck to the entire system design flow.

Appendix B

Embedding PSL in AsmL

This Appendix provides a description of the embedding of the Property Specification Language (PSL) [1] in the Abstract state machines Language (AsmL) [51].

PSL properties are defined in a hierarchical way inspired from the hardware design modular concept. For this reason we defined the embedding in a similar structure, where all the components are defined as objects and every PSL layer *extends* its lower layer using the inheritance feature of AsmL as described in Figure B.1.

B.1 Boolean Layer

This layer is the basic layer of PSL. Even though it is called *Boolean layer*, it includes types other than Boolean such as integers and bit vectors. We embedded this layer in AsmL by defining classes for all types and expressions including their methods. Our embedding is based on the semi-formal semantics presented in the reference manual [1], and the formal semantics definition in HOL [45].

The embedding of the PSL Boolean layer mainly includes:

1. *Expression type class* includes the basic five types: *Boolean*, *PSLBit*, *PSLBitVector*, *Numeric* and *String*. Both *Boolean* and *String* types are directly inherited from the AsmL's *AsmL.Boolean* and *AsmL.String*, respectively. The *PSLBit*

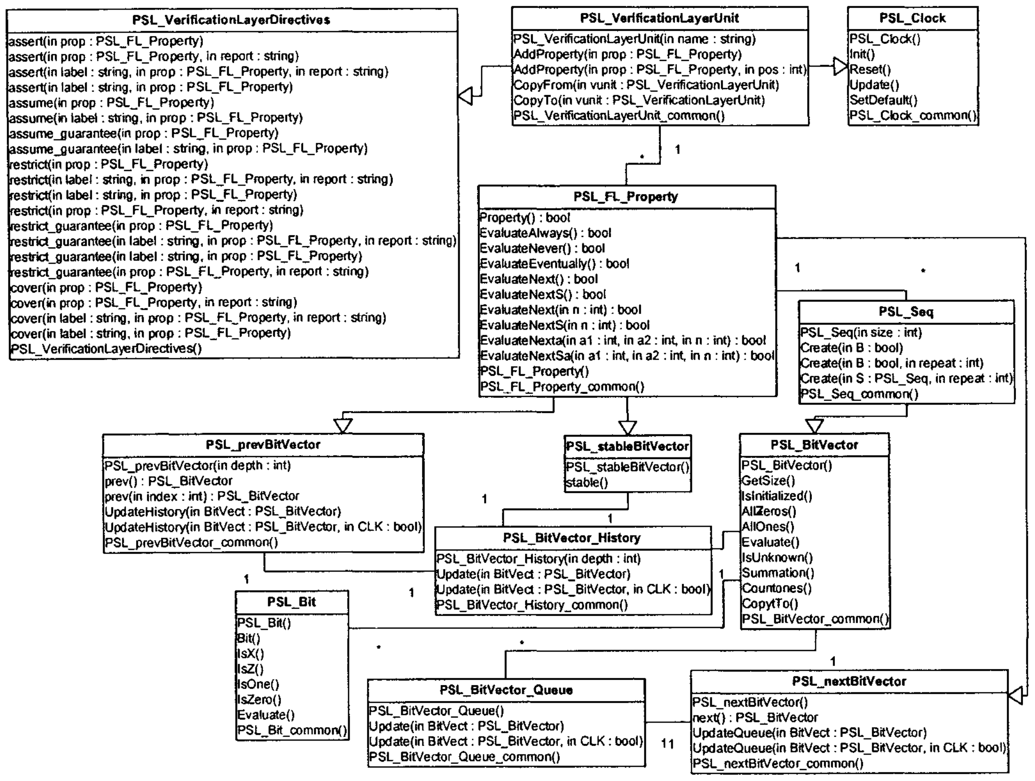

PSL_VerificationLayerDirectives
assert(in prop : PSL_FL_Property)
assert(in prop : PSL_FL_Property, in report : string)
assert(in label : string, in prop : PSL_FL_Property, in report : string)
assert(in label : string, in prop : PSL_FL_Property)
assume(in prop : PSL_FL_Property)
assume(in label : string, in prop : PSL_FL_Property)
assume_guarantee(in prop : PSL_FL_Property)
assume_guarantee(in label : string, in prop : PSL_FL_Property)
restrict(in prop : PSL_FL_Property)
restrict(in label : string, in prop : PSL_FL_Property, in report : string)
restrict(in label : string, in prop : PSL_FL_Property)
restrict(in prop : PSL_FL_Property, in report : string)
restrict_guarantee(in prop : PSL_FL_Property)
restrict_guarantee(in label : string, in prop : PSL_FL_Property, in report : string)
restrict_guarantee(in label : string, in prop : PSL_FL_Property)
restrict_guarantee(in prop : PSL_FL_Property, in report : string)
cover(in prop : PSL_FL_Property)
cover(in prop : PSL_FL_Property, in report : string)
cover(in label : string, in prop : PSL_FL_Property, in report : string)
cover(in label : string, in prop : PSL_FL_Property)
PSL_VerificationLayerDirectives()
PSL_VerificationLayerUnit
PSL_VerificationLayerUnit(in name : string)
AddProperty(in prop : PSL_FL_Property)
AddProperty(in prop : PSL_FL_Property, in pos : int)
CopyFrom(in vunit : PSL_VerificationLayerUnit)
CopyTo(in vunit : PSL_VerificationLayerUnit)
PSL_VerificationLayerUnit_common()
PSL_Clock
PSL_Clock()
Init()
Reset()
Update()
SetDefault()
PSL_Clock_common()
*
1
PSL_FL_Property
Property() : bool
EvaluateAlways() : bool
EvaluateNever() : bool
EvaluateEventually() : bool
EvaluateNext() : bool
EvaluateNextS() : bool
EvaluateNext(in n : int) : bool
EvaluateNextS(in n : int) : bool
EvaluateNexta(in a1 : int, in a2 : int, in n : int) : bool
EvaluateNextSa(in a1 : int, in a2 : int, in n : int) : bool
PSL_FL_Property()
PSL_FL_Property_common()
1
*
PSL_Seq
PSL_Seq(in size : int)
Create(in B : bool)
Create(in B : bool, in repeat : int)
Create(in S : PSL_Seq, in repeat : int)
PSL_Seq_common()
PSL_prevBitVector
PSL_prevBitVector(in depth : int)
prev() : PSL_BitVector
prev(in index : int) : PSL_BitVector
UpdateHistory(in BitVect : PSL_BitVector)
UpdateHistory(in BitVect : PSL_BitVector, in CLK : bool)
PSL_prevBitVector_common()
PSL_stableBitVector
PSL_stableBitVector()
stable()
PSL_BitVector
PSL_BitVector()
GetSize()
IsInitialized()
AllZeros()
AllOnes()
Evaluate()
IsUnknown()
Summation()
Countones()
CopytTo()
PSL_BitVector_common()
1
1
PSL_BitVector_History
PSL_BitVector_History(in depth : int)
Update(in BitVect : PSL_BitVector)
Update(in BitVect : PSL_BitVector, in CLK : bool)
PSL_BitVector_History_common()
1
1
1
PSL_Bit
PSL_Bit()
Bit()
IsX()
IsZ()
IsOne()
IsZero()
Evaluate()
PSL_Bit_common()
*
*
1
PSL_BitVector_Queue
PSL_BitVector_Queue()
Update(in BitVect : PSL_BitVector)
Update(in BitVect : PSL_BitVector, in CLK : bool)
PSL_BitVector_Queue_common()
11
PSL_nextBitVector
PSL_nextBitVector()
next() : PSL_BitVector
UpdateQueue(in BitVect : PSL_BitVector)
UpdateQueue(in BitVect : PSL_BitVector, in CLK : bool)
PSL_nextBitVector_common()

Figure B.1: Partial Class Diagram for Embedding PSL in ASM.

type is constructed using the enumerated structure One, Zero, X, and Z. The *PSLBitVect* type extends the *PSLBit* type and offers additional operations such as access to the bit vector contents. Finally, the *PSLNumeric* type extends the AsmL *Integer* type (AsmL.Integer) by adding some conversion methods from *PSLBitVector* to integers and vice–versa.

2. *PSL expressions* construct properties using the implication and equivalence operators. Both operators are built using AsmL's *implies* operator.

3. *PSL built functions* include all the functions defined by PSL to operate at the Boolean layer. We distinguish here two methods: a method that provides the previous values of a variable (e.g., *prev()*) and a method that provides the future

values of a variable (e.g., *next()*). For both methods, we define a queue structure that extends the *PrimitiveArray* class of AsmL, to store the values of the signals (*PSL_Bit_Vector_Queue* for the *PSLBitVector* type). Note that all the methods over the Boolean layer are overridable according to the type of the input. This approach simplifies writing the properties in AsmL syntax as they will look very close to the PSL structure.

Figure B.2 shows the AsmL code for *PSL_Bit_Vector* class with the method *IsInitialized*() that checks if a BitVector is initialized.

```
class PSL_BitVector
  var m_size as Integer = 1
  var m_sum  as Integer = 0
  var m_array as PrimitiveArray of PSL_Bit = null
  public IsInitialized() as Boolean
    non_initailized = (exists x in {1..m_size} where
      (m_array(x).m_value = X or m_array(x).m_value=Z))
    return not non_initailized
```

Figure B.2: AsmL Embedding of PSL BitVector.

B.2 Temporal Layer

The most important part of this layer is the Sequential Extended Regular Expressions (SERE) feature, which embedding mainly includes:

1. *Sequential Expressions*, where a SERE is defined as an AsmL sequence of Boolean. It offers several operations to construct, manipulate and evaluate the SERE expression. *PSL_Sequence* extends the *PSL_SERE* class. It adds operations needed to create and update the SERE.

2. *Properties* in the form of operations necessary to create properties from sequential expressions. It also controls when and how the sequence is to be verified

(i.e., the property "verify the sequence is true after *n* states" is defined as *PSL_Property.EvaluateNext(n)*).

Figure B.3 shows the example of the *PSL_SERE.Evaluate()*, which checks if a sequence is true in a certain path. This method is activated according to an *INIT* signal that must be set by the property.

```
class PSL_SERE
  var m_size as Integer = 0
  var m_seq as Seq of Boolean
  var m_actualState as Integer = 0
  var m_evaluation as SERE_Evaluation = NOT_STARTED
  var m_evaluationState as SERE_Evaluation = NOT_STARTED
  public Evaluate() as SERE_Evaluation
     require m_evaluationState = INIT
     if(me.m_seq(m_actualState) = false)
       m_evaluation := FAILED
       return FAILED
      else
        if m_actualState = m_size
          m_actualState := m_actualState + 1
          return IN_PROGRESS
        else
          m_actualState := 0
          return SUCCEEDED
```

Figure B.3: AsmL Embedding of PSL SERE.

B.3 Verification Layer

This layer is intended to tell the verification tool how to perform the verification process. It allows the construction of assertions from properties and to specify relations between them. The embedding mainly includes:

1. *Verification directives* to specify how the property will be interpreted (assertion,

requirement, restriction or assumption). This class extends the temporal layer class *PSL_Property* defined above.

2. *Verification unit* is a compact way to include several properties together. The embedded class includes several operations to add/remove and update the unit's list of properties.

Figure B.4 shows the example of the *PSL_VerificationLayerUnit.CopyFrom()* and *PSL_VerificationLayerUnit.CopyTo()* methods. These latter are usually used to construct the unit by copying properties from or into other existent units, respectively.

```
class PSL_VerificationLayerUnit
  var m_name as String = ""
  var m_size as Integer = 0
  var S as Seq of PSL_FL_Property = null
  CopyFrom(vunit as PSL_VerificationLayerUnit)
    forall i in {1..m_size}
      me.AddProperty(vunit.S(i))
  CopyTo(vunit as PSL_VerificationLayerUnit)
    forall i in {1..m_size}
      vunit.AddProperty(me.S(i))
```

Figure B.4: AsmL Embedding of PSL Verification Layer.

B.4 Modeling layer

This layer is not used in our verification approach since it is intended for VHDL and Verilog flavors of PSL. So we did not consider it in our current embedding.

Appendix C

Description of Case Study Models

This Appendix provides a description of the case studies that have been displayed through out the thesis.

C.1 Packet Switch

Figure C.1 provides a general structure of a 4x4 multi-cast packet switch from the SystemC library [80]. The switch uses a self routing ring of shift registers to transfer cells from one port to another in a pipelined fashion, resolving output contention and efficiently handling multi-cast cells. Input and output ports have FIFO buffers of depth four each. Input and output signals are 16-bit packets. Each input port is connected to a sender process. Each output port is connected to a receiver process. The sender and receiver processes are given distinguished *id* numbers during instantiations. A sender process, writes a random value to data, and sends it to one or more of the four receivers. Sender processes send packets at random intervals, varying from 1 to 4 units of its clock. A receiver process is activated whenever a packet arrives. Then, it displays the content of the packet and the receiver *id*. The switch operates on an external clock, CLK, and an internal clock, $SWCLK$, which is four times faster. Input and output signals are 16-bit packets with the structure given in Figure C.2.

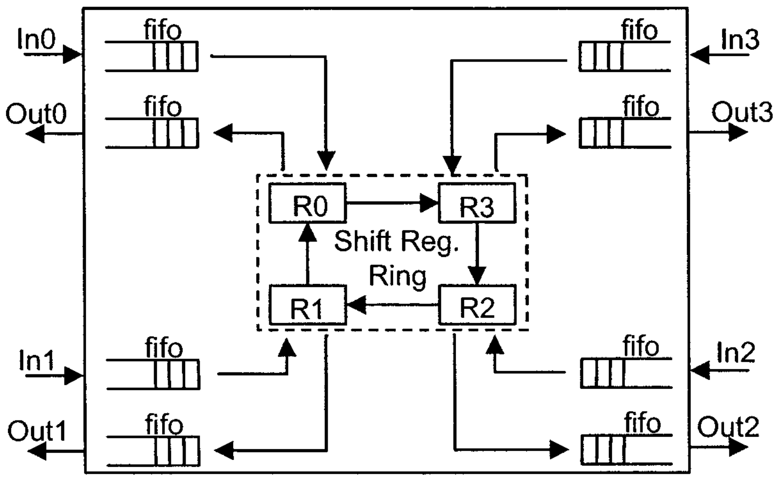


Figure C.1: Switch Structure

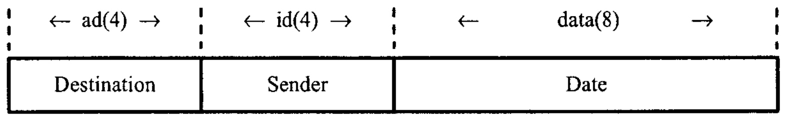


Figure C.2: Packet Structure

C.2 Simple Bus

This bus structure as described in Figure C.3 is part of the SystemC library [80]. It uses an overall form of synchronization where modules attached to the bus execute on the rising clock edge, and the bus itself executes on a falling clock edge. Multiple masters can be connected to the bus. Each master is identified by a unique priority, that is represented by an unsigned integer number. The lower this priority number is, the more important the master is. Each master communicates with the bus via an interface which describes the communication between masters and the bus; three modes are possible:

- Blocking Mode: Data is moved through the bus in burst-mode. The transaction cannot be interrupted by a request with a higher priority.

- Non-Blocking Mode: Read or write a single data word. After the transaction is completed, the caller must take care of checking the status of the last request. The status of the request is one of: SIMPLE_BUS_REQUEST (request issued and placed on the queue), SIMPLE_BUS_WAIT (request being served but is not completed), SIMPLE_BUS_OK (request completed without errors) or SIMPLE_BUS_ERROR (an error occurred during processing of the request).

- Direct Mode: The direct interface functions perform the data transfer through the bus, but without using the bus protocol. They are usually used to debug the state of the memory.

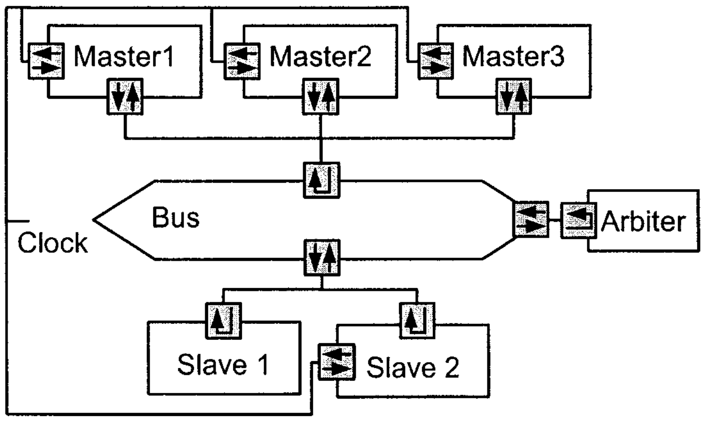


Figure C.3: Simple Bus Structure.

The slave interface describes the communication between the bus and the slaves. Multiple slaves can be connected to the bus. Each slave models some kind of memory that can be accessed through the slave interface. Two modes are possible:

- Direct interface: immediate read or writing of data without using the bus protocol.

- Indirect interface: read or write a single data element, pointed to by data in or from the slave's memory. The functions return instantaneously and the caller must check the status of the transfer.

To the bus more than one master can be connected. Each master is independent of the others, so each one can issue a bus request at any time. The arbiter selects the most appropriate request according the following rules:

- If the current request is a locked burst request, then it is always selected.
- If the last request had its lock flag set and is again 'requested', it is selected from the collection queue and returned, otherwise:
- The request with the highest priority is selected from the collection queue and returned.

C.3 PCI Bus

The PCI bus boasts a 32-bit data path, 33MHz clock speed and a maximum data transfer rate of 132MB/sec. A 64-bit specification exists for future PCI designs, which will double data transfer performance to 264MB/sec. In Figure C.4, we show a generic structure of the PCI bus with a single master and a slave. We added also an external monitor module that will be used to track the signals at the input and output ports of the bus in order to validate the good functioning of the bus.

Each PCI master has a pair of arbitration lines that connect it directly to the PCI bus arbiter. When a master requires the use of the PCI bus, it asserts its device specific REQ# line to the arbiter. When the arbiter has determined that the requesting master should be granted control of the PCI bus, it asserts the GNT# (grant) line specific to the requesting master. In the PCI environment, bus arbitration can take place while another master is still in control of the bus.

In PCI terminology, data is transferred between an initiator, which is the bus master, and a target, which is the bus slave. The initiator, drives the C/BE[3:0]# signals (Figure C.4) during the address phase to signal the type of transfer (memory read, memory write, I/O read, I/O write, etc.). During data phases, the C/BE[3:0]#

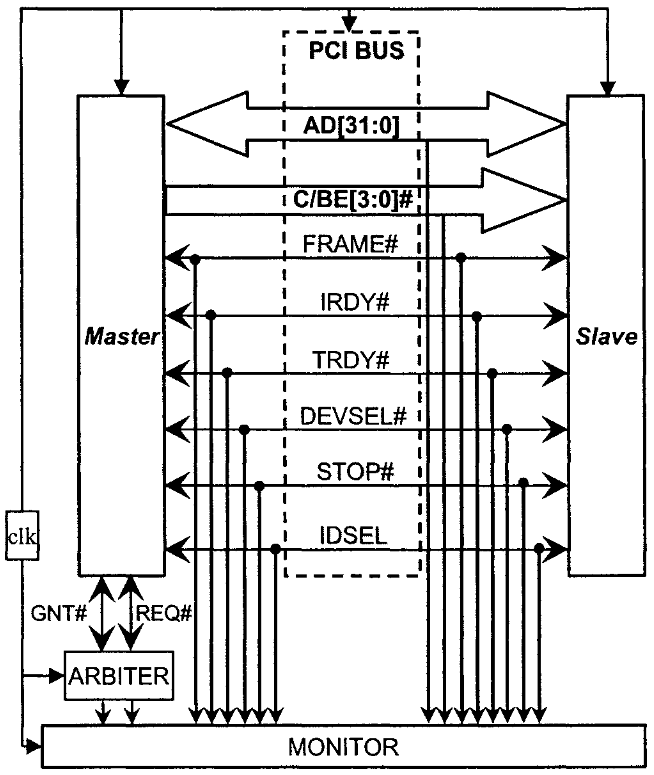


Figure C.4: PCI Bus Structure.

signals serve as byte enable to indicate which data bytes are valid. Both the initiator and target may insert wait states into the data transfer by de-asserting the IRDY# and TRDY# signals. Valid data transfers occur on each clock edge in which both IRDY# and TRDY# are asserted. A target may terminate a bus transfer by asserting STOP#. When the initiator detects an active STOP# signal, it must terminate the current bus transfer and re-arbitrate for the bus before continuing. If STOP# is asserted without any data phases completing, the target has issued a retry. If STOP# is asserted after one or more data phases have successfully completed, the target has issued a disconnect.

C.4 AGP Bus

AGP (Accelerated Graphics Port) [58] was introduced to meet consumer demand for high-resolution 3D graphics in home computers. New software programs (especially games) require more and more video bandwidth for fancy textures, high frame rate animations, etc. While the AGP bus employs 66 MHz clocked PCI specifications; it also has the advantage of allowing large amounts of graphics data to be transferred directly between the computer's main memory and the AGP video card. This feature allows the video card to share the system memory on demand. The AGP bus is designed strictly for video processing and does not have to share available bandwidth with other connected devices. Most high-performance video cards are now only available as an AGP version.

Both AGP bus transactions and PCI bus transactions may be run over the AGP interface. An AGP master (graphics) device may transfer data to the system memory using either AGP transactions or PCI transactions. The corelogic can access the AGP master device only with PCI transactions. Traffic on the AGP interface may consist of a mixture of interleaved AGP and PCI transactions. The access request and data queue structures are illustrated in Figure C.5.

In addition to the PCI features, AGP includes:

- Direct Memory Execute (DME) that gives AGP chips the capability to access the main memory directly for complex operations of texture mapping.
- Pipelining and sideband addressing of directly accessing texture maps in system memory.
- Multiple requests for data during a bus or memory access.
- A dedicated non-shared bandwidth with other devices.

There are two primary AGP usage models for 3D rendering that have to do with how data is partitioned and accessed, and the resultant interface data flow

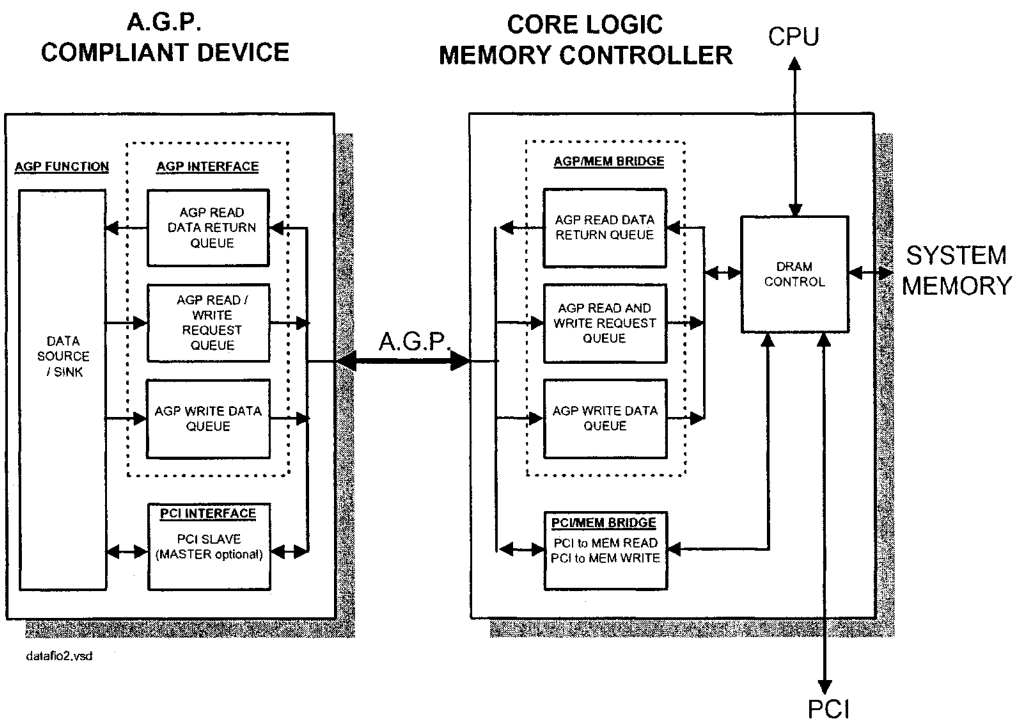


Figure C.5: AGP Access Queuing Model [58].

characteristics: the DMA model and the execute models. In the *DMA model*, the primary graphics memory is the local memory associated with the accelerator, referred to as local frame buffer. 3D structures are stored in system memory, but are not used (or executed) directly from this memory; rather they are copied to primary (local) memory (the DMA operation) to which the rendering engines address generator makes its references. In the *execute model*, the accelerator uses both the local memory and the system memory as primary graphics memory. From the accelerators perspective, the two memory systems are logically equivalent; any data structure may be allocated in either memory, with performance optimization as the only criteria for selection.

C.5 Look-Aside Interface (LA-1)

LA-1 [79]is a standard interface used to interconnect network-processing units (NPUs). It targets look-up-tables and memory-based coprocessors and emphasizes as much as possible on the use of existing technologies. It is based on QDR and Sigma RAM technologies. Although modeled on an SRAM interface, the LA-1 specification aims to accommodate other devices as well, such as classifiers and encryption co-processors.

The LA-1 interface major features include:

- Concurrent read and write operation.
- Unidirectional read and write interfaces.
- Single address bus.
- 18 pin DDR data output path.
- 18 pin DDR data input path.
- Byte write control for writes.

The LA-1 interface requires a master-clock pair. The master clocks (K and K#) are ideally 180 degrees out of phase with each other, and they are outputs for the host device and inputs for the slave device. A write cycle is initiated by asserting WRITE_SEL (W#) low at rising edge of K (K clock). The address of the write cycle is provided at the following falling edge of K (K# clock which 180 degrees out phase from clock K). A read cycle is initiated by asserting READ_SEL (R#) low at rising edge of K (K clock) and the read address is presented on the same rising edge. A block diagram of an LA-1 with four banks is given in Figure C.6.

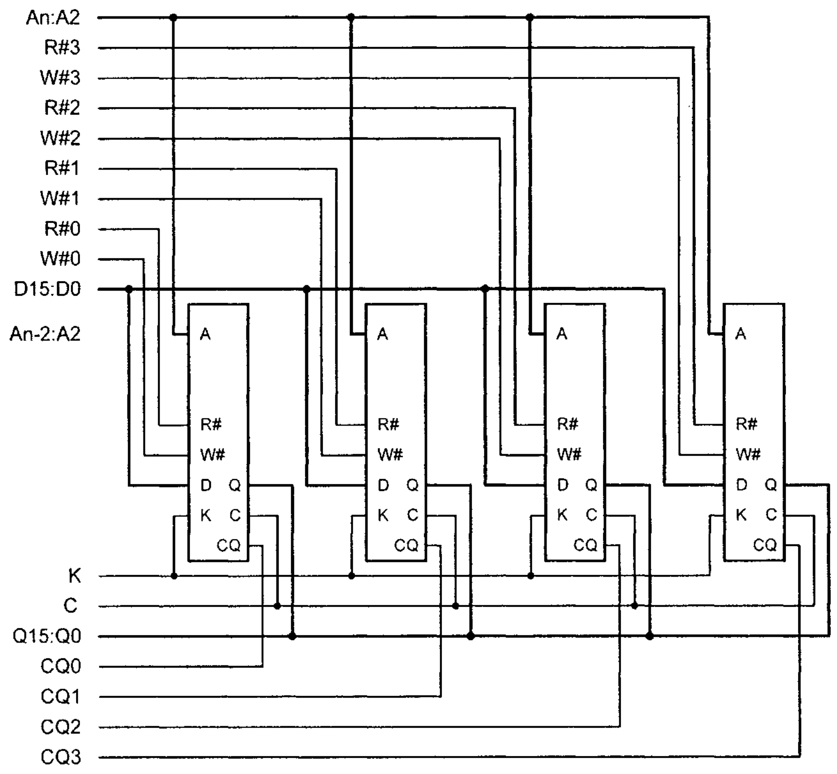


Figure C.6: Look-Aside Interface (4 Banks) [79].

Bibliography

[1] Accellera Organization. *Accellera Property Specification Language Reference Manual, Version 1.1. www.accellera.org.* www.accellera.org, June 2004.

[2] Accellera Organization. *SystemVerilog 3.1a Language Reference Manual: Accelleras Extensions to Verilog.* www.accellera.org, May 2004.

[3] P. Alexander, R. Kamath, and D. Barton. System Specification in Rosetta. In *Proc. IEEE Engineering of Computer Based Systems Symposium*, pages 299–307, UK, April 2000.

[4] T. Ball and S. K. Rajamani. The SLAM Project: Debugging System Software via Static Analysis. In *Proc. Conference on Principles of Programming Languages*, pages 1–3, Pittsburgh, PA, USA, January 2002.

[5] I. Beer, S. Ben-David, C. Eisner, D. Fisman, A. Gringauze, and Y. Rodeh. The Temporal Logic Sugar. *Lecture Notes in Computer Science*, 2102:363–367, July 2001.

[6] P. Bellows and B. L. Hutchings. JHDL - An HDL for Reconfigurable Systems. In *Proc. IEEE Symposium on FPGAs for Custom Computing Machines*, pages 175–184, Los Alamitos, California, USA, 1998.

[7] H. Bhugra. LA-1B: Moving the Look-Aside Interface Forward. *CommsDesign.com*, August 2002.

[8] E. Börger, G. Fruja, V. Gervasi, and R. Stark. A High-Level Modular Definition of the Semantics of C#. *Theoretical Computer Science*, (to appear).

[9] E. Börger and R. Stark. *Abstract State Machines: A Method for High-Level System Design and Analysis*. Springer–Verlag, 2003.

[10] F. Bourdoncle. *Semantiques des Langages Imperatifs d'Ordre Superieur et Interpretation Abstraite*. PhD thesis, Ecole Polytechnique, Paris, France, 1992.

[11] Cadence Design Systems. *Formal Verification Using Affirma FormalCheck, version 2.4*. August 1999.

[12] Cadence Design Systems. *Cadence Verification Extensions, V. 5.0*. 2003.

[13] L. Cai and D. Gajski. Transaction Level Modeling: an Overview. In *Proc. of the International Conference on Hardware/Software Codesign and System Synthesis*, pages 19–24, Newport Beach, California, USA, October 2003.

[14] C. Chambers and D. Ungar. Iterative Type Analysis and Extended Message Splitting: Optimizing Dynamically-typed Object-oriented Programs. *Lisp and Symbolic Computation*, 4(3):283–310, 1991.

[15] A. Church. *Introduction to Mathematical Logic*. Princeton University Press, 1996.

[16] K. Claessen and J. Mårtensson. An Operational Semantics for Weak PSL. In *Formal Methods in Computer-Aided Design*, Lecture Notes in Computer Science, pages 337–351. Springer Verlag, 2004.

[17] E. Clarke, O. Grumberg, and D. Long. A Decade of Concurrency – Reflections and Perspectives. *Lecture Notes in Computer Science*, 803:124–175, June 1993.

[18] E. M. Clarke and J. M. Wing. Formal Methods: State of the Art and Future Directions. *ACM Computing Surveys*, 28(4):626–643, 1996.

[19] C. Consel and F. Noel. A General Approach for Run-time Specialization and its Application to C. In *Proc. of the Symposium on Principles of Programming Languages*, pages 145–156, St. Petersburg Beach, Florida, USA, January 1996. ACM Press.

[20] P. Cousot. Constructive Design of a Hierarchy of Semantics of a Transition System by Abstract Interpretation. *Theoretical Computer Science*, 277(1-2):47–103, 2002.

[21] P. Cousot and R. Cousot. Abstract Interpretation: a Unified Lattice Model for Static Analysis of Programs by Construction of Approximation of Fixpoints. In *Proc. Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, California, USA, January 1977.

[22] P. Cousot and R. Cousot. Static Determination of Dynamic Properties of Recursive Procedures. In *Proc. of the Formal Description of Programming Concepts Conference*, pages 237–277, St-Andrews, USA, March 1977.

[23] P. Cousot and R. Cousot. Constructive Versions of Tarskis Fixed Point Theorems. *Pacific Journal of Mathematics*, 82(1):43–57, 1979.

[24] P. Cousot and R. Cousot. (s)ystematic Design of Program Analysis Frameworks. In *Conference Record of the Sixth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 269–282, San Antonio, Texas, USA, 1979.

[25] P. Cousot and R. Cousot. Systematic Design of Program Analysis Frameworks. In *Proc. Symposium on Principles of Programming Languages*, pages 269–282, San Antonio, Texas, USA, 1979.

[26] P. Cousot and R. Cousot. Abstract Interpretation Frameworks. *Journal of Logic and Computation*, 2(4):511–547, August 1992.

[27] P. Cousot and R. Cousot. Systematic Design of Program Transformation Frameworks by Abstract Interpretation. In *Proc. Symposium on Principles of Programming Languages*, pages 178–190, New York City, New York, USA, January 2002.

[28] CoWare Inc. *SPW Users Guide, Product Version 4.85.* www.coware.com, August 2004.

[29] CynApps Inc. Cynlib: A C++ Library for Hardware Description Reference Manual, 1999.

[30] R. Damasevicius and V. Stuikys. Application of UML for Hardware Design based on Design Process Model. In *Proc. Asia South Pacific Design Automation Conference*, pages 244–249, Yokohama, Japan, 2000.

[31] G. De Micheli. Hardware Synthesis from C/C++ Models. In *Proc. of the Conference on Design, Automation and Test in Europe*, pages 80–81, Munich, Germany, March 1999. ACM Press.

[32] K. Driesen and U. Holzle. The Direct Cost of Virtual Function Calls in C++. In *Proc. of the Object-oriented programming, systems, languages, and applications Conference*, pages 306–323, San Jose, California, United States, October 1996.

[33] M. Dwyer, J. Hatcliff, R. Joehanes, S. Laubach, C. Pasareanu, and R. W. Visser amd H. Zheng. Tool-supported Program Abstraction for Finite-state Verification. In *Proc. International Conference on Software Engineering*, pages 177–187, Toronto, Ontario, Canada, May 2001.

[34] H . Egli. A Mathematical Model for Nondeterministic Computations. Technical report, ETH, Zurich, Switzerland, 1975.

[35] C. Eisner and D. Fisman. Sugar 2.0 Tutorial Explains Language Basics. *EE Design*, May 2002.

[36] F. Fallah, P. Ashar, and S. Devadas. Simulation Vector Generation from HDL Descriptions for Observability-Enhanced Statement Coverage. In *Proc. of Design Automation Conference*, pages 666–671, Washington, DC, USA, June 1999. IEEE Computer Society.

[37] P. L. Flake and S. J. Davidmann. Superlog, a Unified Design Language for System-on-Chip. In *Proc. of the Asia South Pacific Design Automation Conference*, pages 583–586, Yokohama, Japan, 2000. ACM Press.

[38] D. D. Gajski, J. Zhu, R. Domer, A. Gerstlauer, and S. Zhao. *SpecC: Specification Language and Methodology.* Kluwer Academic Publishers, 2000.

[39] A. Gawanmeh, S. Tahar, and K. Winter. Formal Verification of ASM Designs using the MDG Tool. In *Proc. of the Software Engineering and Formal Methods Conference*, pages 210–219, Brisbane, Australia, September 2003. IEEE Computer Society.

[40] R. Giacobazzi, F. Ranzato, and F. Scozzari. Making Abstract Interpretations Complete. *Journal of the ACM*, 47(2):361–416, 2000.

[41] P. Godefroid and S. Khurshid. Exploring Very Large State Spaces using Genetic Algorithms. *Lecture Notes in Computer Science*, 2280:266–280, April 2002.

[42] R. Goering. A Big Leap Claimed in Functional Verification. *EE Times*, May 2001.

[43] R. Goering. Mentor Adds C Interface to Verification Environment. *EE Times*, February 2002.

[44] R. Goering. Next-generation Verilog Rises to Higher Abstraction Levels. *EE Times*, March 2002.

[45] M. Gordon. Validating the PSL/Sugar Semantics using Automated Reasoning. *Formal Aspects of Computing*, 15(4):406–421, December 2003.

[46] M. Gordon and T. Melham. *Introduction to HOL: A Theorem Proving Environment for Higher-Order Logic.* Cambridge University Press, 1993.

[47] W. Grieskamp, Y. Gurevich, W. Schulte, and M. Veanes. Generating Finite State Machines from Abstract State Machines. *Software Engineering Notes*, 27(4):112–122, 2002.

[48] D. Groβe and R. Drechsler. Checkers for SystemC Designs. In *Proc. Formal Methods and Models for Codesign*, pages 171–178, San Diego, California, USA, June 2004.

[49] T. Grotker. *System Design with SystemC.* Kluwer Academic Publishers, Norwell, MA, USA, 2002.

[50] PCI Special Interest Group. www.pcisig.com, 2004.

[51] Y. Gurevich, B. Rossman, and W. Schulte. Semantic Essence of AsmL. Technical report, Microsoft Research Tech. Report MSR-TR-2004-27, March 2004.

[52] J. Holland. *Adaptation in Natural and Artificial Systems.* The University of Michigan Press, 1975.

[53] G. J. Holzmann. *The Spin Model Checker: Primer and Reference Manual.* Addison-Wesley, 2003.

[54] J. Horgan. Assertion Based Verification. *EDA Weekly*, October 2004.

[55] J. Huggins. Abstract state machines website, www.eecs.umich.edu/gasm, 2005.

[56] B. L. Hutchings and B. B. Nelson. Using General-Purpose Programming Languages for FPGA Design. In *Proc. of the Design Automation Conference*, pages 561–566, Los Angeles, California, USA, June 2000. ACM Press.

[57] IEEE Inc. IEEE Std. 1364–2001, IEEE Standard for Verilog Hardware Description Language 2001. IEEE Product No. SH94921-TBR, 2001.

[58] Intel Corp. AGP v3.0 interface specification, 2002.

[59] A. A. Jerraya, M. Romdhani, P. Le Marrec, F. Hessel, P. Coste, C. Valderrama, G. F. Marchioro, J. M. Daveau, and N. E. Zergainoh. Multi-language Specification for System Design and Codesign. In *Proc. NATO-ASI 1998 on System Level Synthesis*, Lucca, Italy, August 1999. Kluwer Academic Publishers.

[60] J. Y. Jou and C. N. Liu. Coverage Analysis Techniques for HDL Design Validation. In *Proc. Asia Pacific CHip Design Languages*, Fukuoka, Japan, October 1999.

[61] H. Jula. ASM Semantics for C# 2.0. In *Proc. of the International Conference on Abstract State Machines*, Paris, France, March 2005.

[62] H. Jula and N. Fruja. An Executable Specification of C#. In *Proc. of the International Conference on Abstract State Machines*, Paris, France, March 2005.

[63] S. N. Kamin and U. S. Reddy. Two Semantic Models of Object-Oriented Languages. *Theoretical Aspects of Object-oriented Programming: Types, Semantics, and Language Design*, pages 463–495, 1994.

[64] M. Kantrowitz and L. Noack. I'm done simulating; now what? Verification Coverage Analysis and Correctness Checking of the DECchip 21164 Alpha Microprocessor. In *Proc. of Design Automation Conference*, pages 325–330, Las Vegas, Nevada, USA, June 1996. IEEE Computer Society.

[65] M. Karr. Affine Relationships among Variables of a Program. *Acta Informatica*, 6:133–151, 1976.

[66] C. Kern and M. Greenstreet. Formal Verification in Hardware Design: a Survey. *ACM Transactions on Design Automation of Electronic Systems*, 4(2):123–193, 1999.

[67] D. Ku and G. De Micheli. HERCULES–a System for High-level Synthesis. In *Proc. of the Design Automation Conference*, pages 483–488, Los Alamitos, California, USA, June 1988. IEEE Computer Society Press.

[68] D. Ku and G. De Micheli. HardwareC – A Language for Hardware Design (Version 2.0). Technical report, Stanford, California, USA, 1990.

[69] T. Kuhn, T. M. Oppold, M. Edwards, and Y. Kashai. A Framework for Object Oriented Hardware Specification, Verification, and Synthesis. In *Proc. of the Design Automation Conference*, pages 413–418, Las Vegas, Nevada, USA, June 2001. ACM Press.

[70] R. P. Kurshan. *FormalCheck User's Manual.* Cadence Design Inc., 1998.

[71] F. Logozzo. *Anhalyse Statique Modulaire de Langages a Objets.* PhD thesis, Ecole Polytechnique, Paris, France, June 2004.

[72] F. Masdupuy. *Analyse Semantique Relationnelle des Indices de Tableaux par Congruence et Trapezoides Rationnels.* PhD thesis, Ecole Polytechnique, Paris, France, 1993.

[73] C. Maxfield. A Plethora of Languages. *EE Design*, August 2001.

[74] Microsoft Corp. AsmL for Microsoft .NET Framework. research.microsoft.com, 2005.

[75] W. Müller, J. Ruf, and W. Rosenstiel. *SystemC Methodologies and Applications.* Kluwer Academic Pub., 2003.

[76] B. Monsuez. *Typage par Interpretation Abstraite.* PhD thesis, Ecole Polytechnique, Paris, France, 1994.

[77] P. D. Mosses. *Dotational Semantics*, volume B of *Handbook of Theoretical Computer Science*, chapter 11, pages 575–631. Elsevier Science B.V., 1990.

[78] D. Moundanos, J. A. Abraham, and Y. V. Hoskote. Abstraction Techniques for Validation Coverage Analysis and Test Generation. *IEEE Transactions on Computers*, 47(1):2–14, 1998.

[79] Network Processing Forum. *Look-Aside (LA-1) Interface, Implementation Agreement, Revision 1.1.* Kluwer Academic Publishers, April 2004.

[80] Open SystemC Initiative. www.systemc.org, 2005.

[81] OSI. *SystemC 2.0.1 Language Reference Manual.* 2005.

[82] R. Otten and P. Stravers. Challenges in Physical Chip Design. In *Proc. of International Conference on Computer-Aided Design*, pages 84–92, Piscataway, New Jersy, USA, 2000. IEEE Press.

[83] I. Page. Constructing Hardware-Software Systems from a Single Description. *Journal of VLSI Signal Processing*, 12(1):87–107, 1996.

[84] P. R. Panda. SystemC: a Modeling Platform Supporting Multiple Design Abstractions. In *Proc. of the International Symposium on Systems Synthesis*, pages 75–80, Montral, Quebec, Canada, August 2001. ACM Press.

[85] P. G. Paulin, C. Pilkington, and E. Bensoudane. StepNP: A System-Level Exploration Platform for Network Processors. *IEEE Design & Test of Computers*, 19(6):17–26, 2002.

[86] P. C. Pixley. Integrating Model Checking into the Semiconductor Design Flow. *Computer Design's Electronic Systems Journal*, pages 67–74, March 1999.

[87] H. G. Rice. Classes of Recursively Enumerable Sets and their Decision Problems. *Transactions of the American Mathematical Society*, 74:358–366, October 1953.

[88] R. Roth and D. Ramanathan. A High-Level Hardware Design Methodology using C++. In *Proc. of High Level Design Validation and Test Workshop*, pages 73–80, San Diego, California, USA, November 1999. IEEE Computer Society.

[89] H. Rudin. Protocol Development Success Stories: Part I. In *Proc. International Symposium on Protocol Specification, Testing, and Verification*, Lake Buena Vista, Florida, USA, June 1992.

[90] A. Salem. Formal Semantics of Synchronous SystemC. In *Proc. Design, Automation and Test in Europe Conference*, pages 376–381, Munich, Germany, March 2003.

[91] M. Santarini. Mentor Offers Formal Verification for SoC Designs. *EE Times*, April 2000.

[92] M. Santarini. System-Level Languages Won't Appear Overnight, Experts Say. *EE Times*, March 2001.

[93] S. E. Schultz. The New System-Level Design Language. In *Integrated System Design Magazine*, July 1998.

[94] C. Schulz-Key, M. Winterholer, T. Schweizer, T. Kuhn, and W. Rosenstiel. Object-oriented Modeling and Synthesis of SystemC Specifications. In *Proc. Conference on Asia South Pacific design automation*, pages 238–243. IEEE Press, 2004.

[95] K. Shimizu, D. L. Dill, and A. J. Hu. Monitor-Based Formal Specification of PCI. In *Proc. Formal Methods in Computer-Aided Design*, pages 335–353, Austin, Texas, November 2000. LNCS 1954, Springer-Verlag.

[96] D. Slogsnat, P. R. Schulz, and U. Brning. Lessons Learned from using Superlog, SystemVerilog's Predecessor. In *Proc. of Forum on Specification and Design Languages*, Frankfurt, Germany, September 2003.

[97] M. Spielmann. Automatic Verification of Abstract State Machines. In *Proc. of the International Conference on Computer Aided Verification*, pages 431–442, London, UK, 1999. Springer-Verlag.

[98] F. Stark and E. Börger. An ASM Specification of C# Threads and the .NET Memory Model. In Wolf Zimmermann and Bernhard Thalheim, editors, *Abstract State Machines*, volume 3052 of *Lecture Notes in Computer Science*, pages 38–60. Springer, 2004.

[99] R. Stark, J. Schmid, and E. Börger. *Java and the Java Virtual Machine : Definition, Verification, Validation.* Springer–Verlag, 2001.

[100] SynaptiCAD Inc. Website: http://www.syncad.com/, 2005.

[101] Open SystemC Initiative. SystemC 2.0.1 Language Reference Manual., 2003.

[102] A. Tarski. A Lattice-theoretical Fixpoint Theorem and its Applications. *Pacific Journal of Mathematics*, 5:285–309, 1955.

[103] A. M. Turing. Computing Machinery and Intelligence. In A. Collins and E. E. Smith, editors, *Readings in Cognitive Science: A Perspective from Psychology and Artificial Intelligence*, pages 6–19. Kaufmann, San Mateo, CA, 1988.

[104] C. D. Turner and D. J. Robson. The State-based Testing of Object-oriented Programs. In *Proc. of the Conference on Software Maintenance*, pages 302–310, Washington, DC, USA, 1993. IEEE Computer Society.

[105] F. Vederine. *Analyses Totales de Programmes par Interpretation Abstraite.* PhD thesis, Ecole Polytechnique, Paris, France, 2000.

[106] Verisity Ltd. Website: http://www.verisity.com/, 2005.

[107] B. A. Wichmann, A. A. Canning, D. L. Clutterbuck, L. A. Winsbarrow, N. J. Ward N J, and D. W. R. Marsh. Industrial Perspective on Static Analysis. *Software Engineering Journal*, 10(2):69–75, March 1995.

[108] K. Winter. *Model Checking Abstract State Machines.* PhD thesis, Technical University of Berlin, Berlin, Germany, 2001.

Biography

Education

- **Concordia University:** Montreal, Quebec, Canada
  Ph.D candidate, in Electrical Engineering, 9/01-09/05

- **Ecole Nationale des Ingenieurs de Tunis (ENIT)):** Tunis, Tunisia
  M.Sc., in Communication Systems, 9/99 - 5/01

- **Ecole Superieure de Communication (Sup'Com):** Tunis, Tunisia
  B.Sc., in Telecommunication Engineering, 9/94 - 7/99

Work Experience

- **Research Assistant:** 9/01-9/05
  Hardware Verification Group (HVG), Concordia University

- **Software Engineer:** 8/99-8/01
  Biodata Carthago, Tunis, Tunisia

Publications

Jornal Papers [1]

[Bio:Jr–1] A. Habibi and S. Tahar. On the Transformation of SystemC to AsmL using Abstract Interpretation. *Electronic Notes in Theoretical Computer Science*, 131:39–49, May 2005.

[Bio:Jr–2] A. Habibi and S. Tahar. Design and Verification of SystemC Transaction Level Models. *IEEE Transactions on Very Large Scale Integration Systems*, (to appear).

[Bio:Jr–3] A. Habibi and S. Tahar. A Survey: System-on-a-Chip Design and Verification. *ACM Transactions on Programming Languages and Systems*, (submitted).

[Bio:Jr–4] A. Habibi and S. Tahar. Formal Verificaction of the ADSP-2100 Processor using the HOL Theorem Prover. *IEE Proceedings on Computers and Digital Techniques*, (submitted).

[Bio:Jr–5] A. Habibi and S. Tahar. SystemC Denotational Semantics. *Logical Methods in Computer Science*, (submitted).

Conference Papers [2]

[Bio:Cf–1] A. Habibi and S. Tahar. An Approach for the Verification of SystemC Designs using AsmL. In *Proc. of the Symposium on Automated Technology for Verification and Analysis*, LNCS 3707, pages 69–83, Taipei, Taiwan, 2005.

[1]**Bio:Jr refers to a jornal paper in the biography section.**
[2]**Bio:Cf refers to a conference paper in the biography section.**

[Bio:Cf–2] M. Zaki, A. Habibi, S. Tahar, and G. Bois. On the Formal Analysis of Analog Systems using Interval Abstraction. In *Proc. of NETCA Workshop on Verification and Theorem Proving for Continuous Systems*, Oxford, UK, August 2005.

[Bio:Cf–3] H. Moinudeen, A. Habibi, and S. Tahar. An Executable Specification of the PCI-X Bus Standard in AsmL. In *Proc. of the Canadian Conference on Electrical & Computer Engineering*, Saskatoon, Saskatchewan, Canada, May 2005.

[Bio:Cf–4] A. Habibi and S. Tahar. Design for Verification of SystemC Transaction Level Models. In *Proc. of the Design Automation and Test in Europe*, pages 560–565, Munich, Germany, March 2005.

[Bio:Cf–5] A. Habibi, A. I. Ahmed, O. Ait-Mohamed, and S. Tahar. On the Design and Verification of the Look-Aside Interface. In *Proc. of the Design Automation and Test in Europe*, pages 290–296, Munich, Germany, March 2005.

[Bio:Cf–6] A. Habibi and S. Tahar. AsmL Semantics in Fixpoint. In *Proc. of the International Conference on Abstract State Machines*, pages 233–245, Paris, France, March 2005.

[Bio:Cf–7] A. Gawanmeh, A. Habibi, and S. Tahar. Embedding and Verification of PSL using ASM. In *Proc. of the International Conference on Abstract State Machines*, pages 201–215, Paris, France, March 2005.

[Bio:Cf–8] K. Oumalou, A. Habibi, and S. Tahar. Design for Verification of a PCI Bus in SystemC. In *Proc. of the Symposium on System-on-Chip*, Tampere, Finland, November 2004.

[Bio:Cf–9] A. Habibi, A. Gawanmeh, and S. Tahar. Assertion Based Verification of

PSL for SystemC Designs. In *Proc. of the Symposium on System-on-Chip*, Tampere, Finland, November 2004.

[Bio:Cf–10] A. Habibi and S. Tahar. Towards an Efficient Assertion Based Verification of SystemC Designs. In *Proc. of the High Level Design Validation and Test Workshop*, pages 19–22, Sonoma Valley, California, USA, November 2004.

[Bio:Cf–11] A. Gawanmeh, A. Habibi, and S. Tahar. Enabling SystemC Verification using Abstract State Machines. In *Proc. of the Forum on Specification & Design Languages*, pages 19–22, Lille, France, September 2004.

[Bio:Cf–12] A. Habibi, S. Tahar, and L. Halleb. Formal Verification of a Bus Structure Modeled in SystemC. In *Proc. of the Northeast Workshop on Circuits and Systems*, pages 61–64, Montreal, Quebec, Canada, June 2004.

[Bio:Cf–13] A. Habibi and S. Tahar. On the Extension of SystemC by SystemVerilog Assertions. In *Proc. of the Canadian Conference on Electrical & Computer Engineering*, volume 4, pages 1869–1872, Niagara Falls, Ontario, Canada, May 2004.

[Bio:Cf–14] A. Habibi and S. Tahar. A Survey on System-On-a-Chip Design Languages. In *Proc. of the International Workshop on System-on-Chip*, volume 4, pages 212–215, Calgary, Alberta, Canada, June–July 2003.

[Bio:Cf–15] F. Wang, A. Habibi, and S. Tahar. Translating LTL Specification to MDG-HDL. In *Proc. of the Canadian Conference on Electrical & Computer Engineering*, Montreal, Quebec, Canada, May 2003.

[Bio:Cf–16] A. Habibi, S. Tahar, and A. Ghazel. Formal Verification of a DSP Chip using an Iterative Approach. In *Proc. of the Digital System Design*, pages 12–19, Dortmund, Germany, September 2002.

[Bio:Cf–17] A. Habibi, S. Tahar, and A. Ghazel. Behavioral Modeling and Verification of the ADSP-2100 Processor using HOL. In *Proc. of the Canadian Conference on Electrical & Computer Engineering*, Winnipeg, Manitoba, Canada, May 2002.

[Bio:Cf–18] A. Habibi, S. Tahar, and A. Ghazel. A Progressive Methodology for the Verification of a DSP Chip. In *Proc. of the Great Lakes Symposium on VLSI*, New York City, New York, USA, April 2002.

Technical Reports [3]

[Bio:Tr–1] A. Habibi and S. Tahar. Constructing Sound SystemC/AsmL and AsmL/SystemC Transformations. Technical report, Department of Electrical and Computer Engineering, Concordia University, Montrea, QC, Canada, January 2005.

[Bio:Tr–2] A. Habibi and S. Tahar. On the Transformation of SystemC to Asml using Abstract Interpretation. Technical report, Department of Electrical and Computer Engineering, Concordia University, Montrea, QC, Canada, December 2004.

[Bio:Tr–3] A. Habibi and S. Tahar. Asml Fixpoint Semantics. Technical report, Department of Electrical and Computer Engineering, Concordia University, Montrea, QC, Canada, November 2004.

[Bio:Tr–4] A. Habibi and S. Tahar. SystemC Fixpoint Semantics. Technical report, Department of Electrical and Computer Engineering, Concordia University, Montrea, QC, Canada, September 2004.

[Bio:Tr–5] A. Habibi, A. Gawanmeh, and S. Tahar. Enabling SystemC Verification using Abstract State Machines. Technical report, Department of Electrical

[3]**Bio:Tr refers to a technical report paper in the biography section.**

and Computer Engineering, Concordia University, Montrea, QC, Canada, May 2004.

[Bio:Tr–6] A. Habibi, A. Gawanmeh, and S. Tahar. A Survey: System-on-a-Chip Design and Verification. Technical report, Department of Electrical and Computer Engineering, Concordia University, Montrea, QC, Canada, January 2003.

[Bio:Tr–7] A. Habibi, A. Gawanmeh, and S. Tahar. Formal Verification of the ADSP-2100 Processor using the HOL Theorem Prover. Technical report, Department of Electrical and Computer Engineering, Concordia University, Montrea, QC, Canada, March 2002.

Index

Abstract Interpretation, 9, 11, 69
  Approximate Semantics, 170
  Concrete Domain, 70
  Galois Connection, 71
  Meaning Function, 71
  Semantical Analysis, 69
AGP, 114
ASM, 26, 110
AsmL, 28, 146
  AsmL tool (Asmlt), 113
  Denotational Semantics, 54
  Observation Function, 66
  TLM Model, 148
Assertion Based Verification, 6, 177
Assertion Scope, 151

BANDERA, 6
Bottom-up Verification Methodology, 168

Chromosome, 158
Coverage, 16
  Coverage Preserving, 160
  Enhancing, 156
  Evaluation, 152
  State Coverage, 153
  Transition Coverage, 154
Cynlib, 5, 184

Equivalence Checking, 15

Fitness Criteria, 158
Fixpoints, 25
  Greatest Fixpoint (gfp), 25
  Least Fixpoint (lfp), 25
  Tarski's Theorem, 25
Formal Verification, 8
FormalCheck, 11, 107
FSM, 10, 173
  Hyperstate, 110, 150
  State Exploration, 173
Function Definition, 24

Genetic Algorithms, 144, 156

HardwareC, 183, 184
Hypergraph, 14, 79

Java, 2, 185

Lattice, 24

Model Checking, 4, 9, 105, 176
    AGP Bus, 114
    AsmL Based Approach, 110
    Direct Approach, 105
    Liveness, 107
    Reachability, 111
    Safety, 107
    SPIN, 109

Packet Switch, 97
Partial Order, 23
PCI, 6
Program Transformation, 39
PSL, 15, 170
    Boolean Layer, 189
    Modeling Layer, 193
    Temporal Layer, 191
    Verification Layer, 192

Rosetta, 2, 5
RTL, 1

Semantics
    Abstract Semantics, 78
    Collecting Semantics, 72
        Basic Domains, 76
        Collecting Pairs, 76
Semi-Formal Verification, 9
Simulation, 7, 162
SLAM, 6
SLV, 6
SoC, 2, 5
Soundness of the Transformation, 67
SpecC, 182, 183
Specman Elite, 9, 144
Static Analysis, 8
Sugar, 4
SuperLog, 5, 181, 186
SVA, 122
    Concurrent Assertions, 123
    Immidiate Assertions, 122
SystemC, 2, 183, 185
    Abstraction, 83
        Analysis Tactics, 92
        Classes, 86
        Code, 89
        Declaration, 85
        Expression, 85
        Instructions, 85
        Memory, 84, 88
        Simulator, 86, 91
    ASM model, 173
    Clocks, 33
    Data Types, 31
    Denotational Semantics, 41
    Design Rules, 35
    Methods, 32

Modules, 30

MUTEX, 34

Observation Function, 65

Ports, 30

Signals, 30

Simulator, 36

Threads, 33

SystemC Bus Structure, 137

SystemVerilog, 2, 181

Theorem Prover, 4

Theorem Proving, 8

TLM, 143

Top-down Design Methodology, 167

Traces, 26

UML, 2, 167, 170, 187

Verilog, 1, 182

VHDL, 1