RICE UNIVERSITY

Dynamic Assertion-Based Verification for SystemC

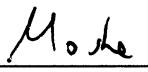
by

Deian Tabakov

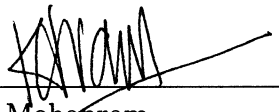
A Thesis Submitted
in Partial Fulfillment of the
Requirements for the Degree

Doctor of Philosophy

Approved, Thesis Committee:



Moshe Y. Vardi, Chair
Karen Ostrum George Professor
in Computational Engineering



Kartik Mohanram
Assistant Professor of
Electrical and Computer Engineering
and Computer Science

Luay Nakhleh
Associate Professor of Computer Science
and Biochemistry and Cell Biology

Houston, Texas

December, 2010

ABSTRACT

Dynamic Assertion-Based Verification for SystemC

by

Deian Tabakov

SystemC has emerged as a de facto standard modeling language for hardware and embedded systems. However, the current standard does not provide support for temporal specifications. Specifically, SystemC lacks a mechanism for sampling the state of the model at different types of temporal resolutions, for observing the internal state of modules, and for integrating monitors efficiently into the model's execution. This work presents a novel framework for specifying and efficiently monitoring temporal assertions of SystemC models that removes these restrictions.

This work introduces new specification language primitives that 1) expose the inner state of the SystemC kernel in a principled way, 2) allow for very fine control over the temporal resolution, and 3) allow sampling at arbitrary locations in the user code. An efficient modular monitoring framework presented here allows the integration of monitors into the execution of the model, while at the same time incurring low overhead and allowing for easy adoption. Instrumentation of the user code is automated using Aspect-Oriented Programming techniques, thereby allowing the integration of user-code-level sample points into the monitoring framework.

While most related approaches optimize the size of the monitors, this work focuses on minimizing the runtime overhead of the monitors. Different encoding configurations are identified and evaluated empirically using monitors synthesized from a large

benchmark of random and pattern temporal specifications.

The framework and approaches described in this dissertation allow the adoption of assertion-based verification for SystemC models written using various levels of abstraction, from system level to register-transfer level. An advantage of this work is that many existing specification languages can be adopted to use the specification primitives described here, and the framework can easily be integrated into existing implementations of SystemC.

Acknowledgments

I would like to extend my sincere gratitude to my adviser Professor Moshe Y. Vardi. His guidance and support have been invaluable and his belief in me unfaltering. I cannot ever hope to repay the time and efforts that he devoted to my growth both academically and as a person; I can only hope that I will have the opportunity to pay this debt forward to a colleague or a student.

I would like to thank Dr. Kartik Mohanram and Dr. Luay K. Nakhleh for serving as members of my dissertation committee. Their questions, comments and suggestions have helped me distill the arguments and clarify the exposition. I appreciate the time and efforts that they put in reading and evaluating my dissertation and presentations.

I also thank Eli Singerman and Gila Kamhi for hosting me during my internship with Intel's Design Technology and Solutions Division. Their insightful comments and many discussions about this work have helped me understand the specification needs of the actual practitioners of dynamic verification. I would also like to acknowledge that this work was supported in part by a grant from the Intel Corporation.

I owe thanks to all past and present students in the Verification and Algorithms research groups who helped me find my bearings in the department. I have had many stimulating discussions with Seth Fogarty, Sumit Nain, Guoqiang Pan, Ben McMahan and Kristin Y. Rozier.

I owe special thanks to Ioan Şucan for his help untangling thorny C++ issues and for the countless hours we have spent in discussion and exchange of ideas.

Last but not least, I would like to thank my parents, Evelina and Todor, and my wife Linh, for their support and encouragement.

Contents

Chapter 1

Introduction

1.1 Design crisis

One logarithmic graph with 4 data points on a straight line led to the following observation in a 1965 issue of the *Electronics* magazine:

> The complexity for minimum component costs has increased at a rate of roughly a factor of two per year [...] Certainly over the short term this rate can be expected to continue, if not to increase. [Moo65]

The author of these now famous words was Gordon Moore, a chemist and a physicist, and co-founder of Intel Corp. 10 years later, Moore updated his prediction, stating that "the new slope might approximate a doubling every two years, rather than every year." [Moo75]

In the decades since, Moore's Law has been widely misunderstood and often misquoted, and has been invoked when discussing capacities of dynamic RAM modules and hard disk drives, the computational power of microchips, and the number of megapixels in consumer digital cameras. In a 1995 speech, Gordon Moore looked back over the 30 years since he first stated his famous "law", and joked:

> The definition of "Moore's Law" has come to refer to almost anything related to the semiconductor industry that, when plotted on semi-log paper, approximates a straight line. [Moo95]

Part of the reason for the popularity of Moore's Law is its uncanny accuracy. Indeed, if we plot the number of transistors on commodity processors versus time,

starting with the early designs in 1960 and going through the latest microprocessor from the Itanium family[1], the curve is clearly exponential. However, as our ability to put more transistors on a die has increased, so has the complexity of the designs. Addressing the increasing complexity has led to an increasing level of abstraction in designs.

In the 1960s and 1970s it was possible to design circuits at the transistor level. As the circuits grew bigger, designers started thinking in terms of gates. In the 1990s, tools for automatic synthesis of gate-level designs allowed the practitioners to adopt an even higher level of abstraction: Register Transfer Level (RTL). This move was widely acclaimed as a revolution of the design process [Fos08].

However, in the 2000s a new type of devices started to become more commonplace, for example, cell phones, network routers, and smartcards, collectively called systems-on-chip (SoC). A SoC often consists of more than one processor, and contains on-chip and external memory, analog and digital signal processors, peripheral devices, and complex on-chip buses. Such systems are often capable of running a modern operating system such as GNU Linux or Windows Mobile. The software and the hardware of such systems are tightly connected, with the software both controlling the hardware, and requiring it to run. Designing and debugging software for such systems is a difficult task by itself, usually requiring many months and even years. Shrinking time-to-market made it all but impossible to wait for the hardware to be designed, taped out, and for the first prototypes to roll off the production floor, before starting to design the software. Instead, there is increasing pressure to co-design the hardware and the software together, a task not well suited for RTL.

Another problem with RTL is that a micro-architecture is usually specified by a natural language document, referred to as the *micro-architectural specification* (MAS) [Var07]. It is increasingly difficult to design RTL from such informal and

[1]At the time of writing of this thesis, the latest Intel microprocessor was Tukwila, with more than 2 billion transistors on a single die

often underspecified documents. Moreover, if there are errors in the RTL, it is almost impossible to determine whether the error was in the original design, or in the translation of the design from MAS to RTL, as the RTL serves as both the micro-architectural model and its implementation [Var07].

The divergence between what engineers could put on the chip and what the designers could simulate and verify, has been increasing, leading Gadi Singer, one of Intel's Vice Presidents, to issue a call to action for the electronic design automation (EDA) industry:

> We have a very good flow generally to go down from RTL to layout. However, RTL was established two decades ago. Since then, complexity has grown severalfold and the EDA industry has not provided a similar level of support for design and refinement at the next level. This is a level that is an absolute requirement to deal with the growing complexity [Goe05].

The need to move beyond the register transfer level motivated academic and industrial research into language for "system-level" design. A new language would ideally support specification and design at various levels of abstraction, incorporation of embedded software, and creation of executable specifications. One of the most popular languages that were intended to answer the needs of the design community is SystemC.

1.2 SystemC

SystemC (IEEE Standard 1666-2005) is both a modeling and a simulation language: designs can be created with various levels of details, and then they can be compiled into executables using standard C++ compilers[2]. The core language consists of macros for modeling fundamental components of hardware designs, such as modules and signals. SystemC also provides hardware-oriented data types like 4-valued logic

[2]A detailed introduction to SystemC follows in Chapter 2.

and arbitrary-precision integers. The implementation of SystemC distributed by the Open SystemC Initiative (OSCI) also provides a reference simulator that is linked to the user code and drives the simulation. Various vendors supply their own simulators, which provide additional support for debugging and co-simulation with other hardware description languages.

One of the reasons for SystemC's success is that it allows designers to model systems at several abstraction levels, from the most concrete (gate level) through the most abstract (system level) [GLMS02]. Individual components can be replaced freely without affecting the rest of the design, thus allowing the engineers to investigate alternative approaches and new ideas. This also allows the design process to be parallelized, with different teams working on different components simultaneously.

As a simulation framework, SystemC provides an *event-driven* environment, where important occurrences like writing to a signal, a clock tick, or a token being consumed from a channel are each represented by an *event*. Synchronization between events and processes is done behind the scenes by SystemC's *simulation kernel*. The kernel keeps track of events, schedules processes to run, and updates the values of signals and channels in a fashion that mimics concurrent execution, even though in reality the processes are run sequentially. The seamless integration between hardware-like components (modules, signals) and software (processes) makes SystemC a prime choice for prototyping and testing hardware and hybrid systems early in the design process [CA02].

The object-oriented encapsulation of classes in C++ is naturally extended in SystemC to protect each module's internal data members (representing local memory) from other modules and processes, except through explicitly defined interfaces. C++'s inheritance capabilities allow for the creation of modular designs in SystemC, which, in turn, facilitate reuse and make IP transfer possible [BGM04].

Various libraries provide further functionality. For example, a popular library called TLM (short for Transaction-Level Modeling) defines channels, interfaces, and

protocols that streamline and standardize the development of high-level models in which complex communication and protocols are reduced to a single "transaction". These factors have helped propel SystemC as a *de facto* industry-wide standard modeling language, less than a decade after its first release, and with this increase in use came the increasing need to verify the designs written in SystemC.

1.3 Design verification

Before we can check if the behavior of a design is correct, we need to have a "yardstick" against which the behavior is compared. Behavior of the design over time is often expressed using temporal formulas [Pnu77]; in his dissertation such temporal formulas are called *properties* of the design. A *specification* is a set of asserted properties, which describe intended behavior of the system. Informally, the goal of design verification is to ensure that the actual behavior of the design is consistent with its specification. If an inconsistency is discovered, the verification tool can usually produce a *witness* of the violation: a trace which corresponds to an execution of the model and which violates the specification. There are two major directions of research in this field: formal verification and dynamic verification.

1.3.1 Formal verification

One feature that all formal verification methods share is that they produce a mathematical proof that the design can never violate the specification. If the design is not written in a formalism with formal semantics, it needs to be translated before model checking can be applied. Practical application is limited to small blocks that contain mostly control logic such as state machines, as opposed to blocks that are used to transform data, such as multipliers [CH07].

One common application of formal verification is in *equivalence checking*: showing that two models have the same behavior. Usually it is done after a refinement step or after some optimizations (e.g., clock-tree synthesis) to ensure that the functionality

of the model or the circuit is still correct [Ber03]. In these cases the old model serves as a specification for the new one. Equivalence checking can also be used to verify the correctness of the output of the synthesis tool, for example, when synthesizing netlists from RTL.

1.3.2 Dynamic verification

While formal verification checks if *all executions* of the system conform to the specification, dynamic verification checks if *a particular execution* of the model conforms to the specification. This approach involves executing the model under verification (MUV) in some environment, while running checkers in parallel with the model. The checkers typically monitor the inputs to the MUV and ensure that the behavior or the output is consistent with the expected behavior or output and, if a violation is detected, return the trace of execution leading to the violation. Dynamic verification is weaker than formal verification because it provides no guarantees that the system can never violate the specification. It is sometimes called "functional verification" (see, e.g., [Ber03, Fos08, Piz07]) and "runtime verification" (see, e.g. [CR07, LS09]). This thesis uses the term "dynamic verification" to highlight the distinction from "formal verification".

There are two major approaches to dynamic verification: *black box* and *white box*. The difference is mainly in the amount of information exposed to the verification framework:

- *Black box*. When using this paradigm, dynamic verification is performed without any knowledge of the details of implementation of the design. All verification is done through explicitly declared interfaces. A disadvantage of using this approach is that an error may occur inside the design without manifesting itself at the interface. However, an advantage of using black box dynamic verification is that the verification effort does not depend on the specific implementation. Pure black-box dynamic verification is impractical for large designs because

they have too many internal signals and states to effectively verify all of the functionality from the periphery [Ber03]. Black box verification is mostly used to specify the behavior of third-party designs and libraries.

- *White box*. This method gives the verification framework full access to the internal structure and implementation of the MUV. Verification efforts can focus on specific blocks or individual functions. In case an error is detected, it is reported sooner than when using black box verification, and it is usually easier to identify the incorrect implementation. White box dynamic verification requires detailed knowledge of the implementation of the design, which is often available only to the engineers who are writing the model.

1.3.3 Trade-offs between dynamic and formal verification

- Capacity vs. Completeness. Formal verification has capacity limits and is best applied to small blocks with critical importance (for example, arbiters and bus controllers). Even small models can have a large state space, which makes formal verification intractable. Dynamic verification has fewer capacity limitations, however, it cannot provide a complete verification solution. In contrast to formal verification, dynamic verification does not constitute proof that the model conforms to the specification, and corner cases may remain unexercised, resulting in undiscovered bugs.

- Formal semantics. Formal verification requires that the model has well defined formal semantics. In particular, C++ is known to lack formal semantics [Vel05], which means that formal verification methods are restricted to a subset of C++. As a consequence, formal verification of SystemC requires that the model be translated to another language (e.g., abstract state machines [GHT04]) with formal semantics.

1.3.4 Limitations of design verification

Although design verification is crucial for uncovering design or implementation bugs, successful application requires good understanding of its limitations. One possible source of frustration comes from specification errors: a specification may misrepresent the design intent, thus triggering a false positive. Software engineers have faced this issue for decades: in 1990 Moser and Melliar-Smith wrote

> "Even carefully written formal specifications are prone to error, and experience has shown that unverified specifications are comparable in reliability to untested programs [MMS90]."

Lam goes even further and claims that even "missing specifications" can be considered a type of specification errors [Lam05].

Another issue is that in most applications, specifications do not provide complete functional coverage. This means that even if the specifications have been proven to hold in full, there may still be undetected errors. The correctness of the model is not an absolute measure; it is always with respect to specific specification of its behavior.

1.4 Assertion-based verification

Monitors (also called "functional checkers" or just "checkers") are used as aids for dynamic verification. In the late 1990s, manually written monitors were a traditional part of the simulation environment (see, e.g., [GBA+99]). Typically, a monitor observes the execution of the MUV and issues a warning or terminates execution if the observed behavior deviates from the expected behavior. In cases when deviation is observed, the problem and its source are easier to identify and debug. Furthermore, using monitors automates the analysis of the tests results and allows a large number of random test vectors to be executed without the need for immediate attention by a verification engineer. The disadvantage is that writing and maintaining monitors

manually is an expensive and laborious process, and for intricate specifications it is very easy to make mistakes when constructing the monitor by hand.

Automated monitor generation was first proposed by Abarbanel et al. [ABG+00], who used the FoCs tool to generate VHDL monitors from specifications written in a temporal language based on Computation Tree Logic (CTL) and regular expressions [BBL98]. Their overall experience with FoCs was very favorable, particularly because it allowed to "[leverage] the same formal rules for model checking of small design blocks as well as for simulation analysis across all higher simulation levels." [ABG+00].

The advantage of automatic generation of monitors from specifications was acknowledged almost immediately by the industry (see, e.g., US Patent 6591403, "System and method for specifying hardware description language assertions targeting a diverse set of verification tools" [BF03], which was submitted in October 2000). The term "assertion-based verification" (ABV) started to be used in white papers from the electronic design automation (EDA) vendors in 2002 (see, e.g., [Syn02]), in peer-reviewed papers in 2003 (see, e.g., [NdPF+03]), and in books in 2003 (e.g., [FKL03]). The industry also recognized the need for temporal languages that can express properties related to ongoing behavior, such as p must hold until q is true [AFF+02]. Assertions are most commonly used to facilitate verification, hence the name assertion-based *verification.* Some authors even prefer using the term assertion-based *design* to emphasize the idea that assertions should be introduced in the earliest stages of the design process [BZ08, FKL03].

Among the several advantages of using ABV is the modular nature of assertions: each one is a partial specification of the system, and those specifications can be added incrementally, as time permits. Designs with thousands of assertions are not uncommon [BZ08], and the only practical way to build such a set of specifications is to add them to the design and to debug them one at a time.

Another advantage of using ABV is that once the assertions have been added to a particular design, they can be reused across different refinement steps (with possible

minor modifications to the sampling rate; see Chapter 3 for a full discussion). Some designs are constructed with reuse in mind and the associated assertions serve both as a formal description of the design, as well as constraints on the inputs that detect and reject incorrect usage.

A further benefit of using ABV in the initial specification of the design is that the assertions allow the verification and the design teams to base their work on a common set of formal properties. This usage of assertions supplements the natural language description of the design, and is an important part of the documentation of the design.

A successful ABV solution requires three components:

1. A formal declarative language for expressing assertions and a formal definition of a trace of execution. The semantics of some temporal operators can only be defined with respect to an execution trace.

2. A monitoring framework capable of observation of the execution trace. This includes both the ability to decide the truth value of any atomic formula, and the ability to sample the values of atomic formulas at all states on the execution trace.

3. A mechanism for automatic generation of monitors from specifications. The monitor must detect all finite executions of the model that violate the property.

This thesis addresses all of these issues and provides a complete framework for applying assertion-based verification to SystemC. Components of the framework are discussed briefly in this chapter, and at length in the subsequent chapters.

1.5 Contributions of this thesis

1.5.1 Specification primitives for SystemC

There have been a few attempts to adapt temporal specification languages to SystemC (see discussion of related work in Chapter 3), but they suffer from several drawbacks. Previous works do not address the most fundamental issue for temporal specification languages: a precise definition of a trace of execution. Moreover, existing temporal specification languages cannot handle the different levels of abstraction that may coexist in a single SystemC design, and cannot be adapted easily as the model is refined. Another issue is that existing temporal specification languages for SystemC focus on the hardware-oriented nature of SystemC and ignore the fact that SystemC models both hardware and software.

This thesis describes a new approach to defining temporal languages for SystemC. The starting point is a precise definition of a SystemC trace. Intuitively, a trace is a sequence of states in the execution of the model. Defining this notion precisely for SystemC is nontrivial because it requires finding a good abstraction of the simulation semantics of SystemC. It is then argued that modern specification languages fail to identify important Boolean properties relevant to the execution of SystemC models. This thesis proposes enriching the Boolean layer with a new set of atomic propositions, exposing the operations of the SystemC kernel, as well as the control flow and the syntax of the user code, thereby making the temporal specification language more expressive. Finally, this work leverages the fact that the clock-sampling mechanism available in modern temporal specification languages offers a way to express temporal properties at different levels of abstraction. A fine sampling would correspond, for example, to subcycle-level abstraction, while coarser sampling would correspond, for example, to transaction-level abstraction. Since any Boolean expression can be used as a clock expression, the additions to the Boolean layer that is proposed here also provides a much finer control over the temporal resolution.

The new primitives proposed in this dissertation are at the Boolean layer and can be added easily to existing temporal specification languages such as PSL [PSL07]; all that is needed is to adapt the underlying syntax for state assertions. The main feature of the resulting framework is the ease with which properties can be expressed at different levels of abstraction, without having to use different languages. This contribution was published as [TVKS08].

1.5.2 Monitoring framework for SystemC

The traditional approaches to dynamic verification involve connecting a separate checker module in parallel with the MUV for each property to be checked. The difficulty of applying this approach to SystemC is that it only allows monitoring the state of the model when control is passed to the checker module. Thus, this approach cannot be applied to monitor properties that refer to finer temporal resolution, e.g., referring to a particular SystemC event. This dissertation argues that the specification primitives discussed earlier require that certain nominal information about the kernel, specifically, kernel phases and event notification, has to be exposed to the monitors.

Once it becomes clear that the SystemC kernel needs to be exposed, the two key questions are how to do it with small changes to an existing implementation, and how to avoid performance penalties. On one hand, optimizing for performance alone would require direct modification of the existing source code to hook the new functionality to the existing data structures. This would require the monitoring framework to be rewritten for each SystemC implementation, limiting portability. On the other hand, optimizing for portability would require adding a layer of indirection that abstracts away the concrete implementation of the SystemC kernel, which would slow down the execution. Since each optimization affects negatively the other, the challenge is to find a good balance between the two. This thesis describes an approach that accomplishes both small change and low performance overhead. The necessary

changes to the SystemC code are modularized to make it easy and fast to modify existing implementations. The framework can easily adapt to changes in the SystemC semantics that may be added in future releases.

This dissertation shows that monitoring SystemC properties using the framework presented here has reasonable overhead (0.05% – 1% per monitor) and that the marginal cost of additional monitors decreases. As proof of concept, the framework is used to specify and check properties of two SystemC models. Based on the empirical results it is argued that the additional expressive powers and flexibility of the framework does not incur a prohibitive performance hit. This contribution was published as [TV10a].

1.5.3 Automatic instrumentation of user code

The benefits of object-oriented encapsulation and data hiding inherent in SystemC are deterrents for effective monitoring. Access to internal variables is granted only to the objects' own processes and is denied to the monitors, which execute as external processes. A key requirement for monitoring the specification language primitives described earlier is allowing monitoring processes access to all internal variables, even those marked **`protected`** or **`private`**. The third contribution of this work is defining a simple yet powerful approach for exposing the execution flow, the syntax, and the state of the user code, without requiring extensive annotation or manual instrumentation from the model designers.

The approach presented here uses Aspect-Oriented Programming (AOP) [KIL+97] techniques to expose function calls and returns, the actual parameters passed to functions, the return values of functions, and the precise instances when the function execution starts and ends. In addition, the approach presented here allows identifying, via regular expressions, any line or a set of lines of code, and exposing them to the monitoring framework as atomic propositions. Empirical results show that the overhead due to automated instrumentation of the user code is very low ($\sim 0.5 \times$

10^{-4}% of baseline execution time per monitor call).

1.5.4 Automatic generation of efficient monitors

The last component of assertion-based verification for SystemC calls for a method for generating runtime monitors from formal properties. For simple properties it may be feasible to write the monitors manually; however, in most industrial workflows, writing and maintaining monitors manually would be an extremely high-cost, labor-intensive, and error-prone process.. This work uses existing approaches to construct a Deterministic Finite Word automaton (DFW) from a temporal property, such that the automaton accepts the finite traces that violate the property. Many works have elaborated on that approach; this thesis follows the algorithm presented in [dR05].

Most of prior work on this subject has focused on the underlying algorithmics or on heuristics to generate smaller monitors or on fast monitor generation. In this thesis the focus is shifted toward optimizing the runtime overhead that monitor execution adds to simulation time. This reflects more accurately the priorities of the industrial applications of monitors. A large SystemC model may be accompanied by dozens and even hundreds of monitors, so lower runtime overhead is a crucial optimization criterion, much more than monitor size or monitor-generation time. This work identifies several algorithmic choices that need to be made when generating temporal monitors for SystemC and presents extensive experimentation to identify the configurations that lead to superior performance. We investigate the effect of individual optimizations on the runtime overhead of monitors, leading to the identification of a combination of optimizations that exhibits the best overall performance. This contribution was published as [TV10b].

1.6 Outline of the thesis

Chapter 2 gives introduction to SystemC and discusses its simulation semantics. Chapter 3 presents new specification primitives that allow existing temporal lan-

guages to handle a rich set of SystemC properties. The mechanisms for exposing the operations of the SystemC kernel are described in Chapter 4. Chapter 5 presents techniques for automating the instrumentation of the user code, and Chapter 6 shows how to generate efficient monitors. Conclusions and future work are described in Chapter 7.

Chapter 2

Fundamentals of SystemC

SystemC is a system-level design framework that is capable of handling both hardware and software components. It allowed a designer to combine complex electronic systems and control units in a single model, to simulate and observer the behavior, and to check if it meets the performance objectives. In 2005 the SystemC language reference manual (LRM) was ratified as IEEE 1666-2005 standard.

In the past, the design process for embedded systems has been mostly serial, moving from the architecture designers to the hardware designers to the hardware verification team and finally to the software team, and each team has been using a different language to implement and refine their portion of the design. This approach is known as "waterfall schedule" among project managers [BD05]. Such workflows may work for designs consisting of a few thousands of lines of RTL code, but it quickly becomes inefficient when the designs get bigger and more complex.

Many contemporary systems consist of application-specific hardware and software, and tight production cycles make it impossible to wait for the hardware to be manufactured before starting to design the software. In a typical system-on-chip [CCH+99], for example, a cell phone, there are hardware components that are controlled by software. In addition, many hardware design decisions, for example, numeric precision or the width of communication buses, are determined based on the needs of the software running on them. This has led to a design methodology where hardware and software are co-designed in the same abstract model. The partitioning between what will be implemented in hardware and what will be written as software is intentionally left blurry at the beginning, allowing the designers the ability to consider different

configurations before committing a functional block to silicon or C.

One of the reasons for SystemC's popularity is its ability to parallelize the design process. SystemC allows blocks implemented at different abstraction levels to run together in the same model. Different components can be refined in parallel, using the abstract model as a blueprint showing how the different blocks interact. Communication between modules is specified using well-defined interfaces, which allows two blocks that conform to the same interface to be swapped seamlessly. This gives designers the ability to explore alternative approaches early in the design process, before committing to a particular architecture.

In the strict sense of the word, SystemC is not a new language. In fact, it is a library of C++ classes and macros that model hardware components, like modules and channels; provide hardware-specific data types, like 4-valued logic types; and define both abstract and specific communication interfaces, like Boolean input. SystemC is built entirely on standard C++, which means that every SystemC model can be compiled with a C++ compiler. The compiled model has to be linked with a SystemC simulator (for example, the OSCI-provided reference implementation) to produce an executable program.

Software typically executes sequentially, partly because most computer architectures have a single CPU core, and partly because a single thread of execution is easier to manage by the operating system. However, in a hardware system, many components execute simultaneously. For example, when using a cellphone to make a call, we activate simultaneously a radio subsystem that handles two-way communication with the cell tower, a signal processing unit that converts voice to signal and signal to voice, and a display controller that shows details about the conversation on the screen. Simulating such a system in software requires the ability to simulate a large number of tasks executing simultaneously, and is critical for the early stages of the design.

SystemC addresses this issue by providing mechanisms for simulating (in software)

parallel execution. This is achieved by a layered approach where high-level constructs share an efficient simulation engine [GLMS02]. The base layer of SystemC provides an event-driven simulation kernel that controls the model's processes in an abstract manner. The kernel leverages a concept borrowed from hardware design languages, called *delta cycle*, to give the executing processes the illusion of parallel execution. The details of the simulation semantics of the kernel, the mechanisms for simulating parallel execution, and the way the kernel interacts with the MUV are described in the second half of this chapter.

2.1 SystemC as a modeling language

2.1.1 Modules

SystemC modules are the most fundamental building blocks. Similar to C++ objects, modules allow related functionality and data to be incorporated into individual entities and to remain inaccessible by the other components of the system unless exposed explicitly. This allows modules to be developed independently and to be reused or sold in commercial libraries [BGM04]. As an example, the skeleton of a SystemC module is presented in Listing 2.1:

```
1 SC_MODULE(Nand) {
2   // Definitions of processes, internal data, etc
3
4   SC_CTOR(Nand) {
5     // Body of constructor, process registration,
6     // sensitivity lists, etc.
7   }
8 };
```

Listing 2.1: Skeleton code for defining a SystemC module.

In this code fragment, **SC_MODULE** is one of SystemC's macros, which declares a C++ class named "Nand". Like any other C++ class, a module can declare local variables and functions. **SC_CTOR** is another predefined macro that simplifies the definition of a constructor for the module. A constructor of a module serves the same

purpose as a constructor of a C++ class (i.e., initializing local variables, executing functions, etc.), but has some additional functionality that is specific to SystemC. For example, the *processes* of the module have to be declared inside the constructor. This is done using pre-defined SystemC macros that specify which class functions should be treated by the SystemC kernel as runnable processes. After declaring each process, the user can optionally specify its *sensitivity list*. The sensitivity list may include a subset of the channels and signals defined in the module, as well as externally defined clock objects or events. Whenever there is a change of value of any of the channels or signals listed in the sensitivity list, the corresponding process is triggered for execution. Listing 2.2 illustrates these concepts.

```
1  SC_MODULE(Nand) {
2    // Definitions of ports
3    sc_in <bool> A, B; // Input signal ports
4    sc_out<bool> F;    // Output signal port
5
6    // Definitions of processes
7    void some_function() {
8      F.write( !(A.read() && B.read()) );
9    }
10
11   SC_CTOR(Nand) {
12     // Process registration, sensitivity lists, etc.
13     SC_METHOD(some_function); // Indicate that this function
14                               // is a ''method process''
15     sensitive << A << B;
16   }
17 };
```

Listing 2.2: A SystemC module of a NAND gate

This code fragment declares two input and one output signals of type **bool** (lines 3–4). The function `some_function()` defined on lines 7–9 implements the expected functionality of the NAND gate. Nothing distinguishes this function from any other standard C++ function, until the macro on line 13 declares it to be a process. **SC_METHOD** indicates that this process is a *method process*. When triggered, a method process executes from start to finish. In particular, a method process cannot

suspend while waiting for some resource to become available. In contrast, a *thread process* may suspend its execution by calling `wait()`. The state of the thread process at the moment of suspension is preserved, and upon subsequent resumption (for example, when the waited-for resource becomes available) the execution continues from the point of suspension. Thread processes are declared using the macro **SC_THREAD**. Both thread and method processes can define a sensitivity list. Each sensitivity list declaration applies to the process immediately preceding the declaration. Line 15 indicates that the method process `some_function()` should be triggered as soon as one of the input signals changes its value.

Any function that is declared as a method process can also be declared as a thread process, at the cost of simulation performance.Nevertheless, certain functionality cannot be expressed easily using method processes, so the extra cost of thread processes is outweighed by their expressiveness. For example, when a process needs to write to a buffer that is currently full, it is very common to suspend execution until the buffer becomes available. An example illustrating such processes is presented later in this chapter.

2.1.2 Hierarchical modules

Building modules that include other modules is as easy as using C++ objects inside another C++ class. For example, we can use the model of a NAND in Listing 2.2 to design a module implementing an EXOR gate. Figure 2.1 shows how the gates need to be connected.

We need three "wires" (labeled $S1$, $S2$ and $S3$) to connect the inner gates to each other. The two inputs signal ports (labeled A and B) and the output signal port (F) can be connected directly to the gates, thus we do not require additional wires. Listing 2.3 shows the SystemC implementation.

```
1 SC_MODULE(Exor) {
2   // Definitions of ports
```

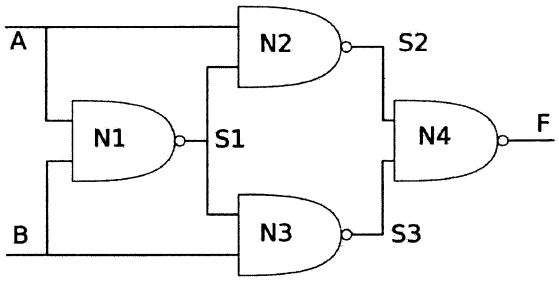


Figure 2.1 : An EXOR gate implemented using 4 NAND gates.

```
3   sc_in <bool> A, B; // Input signal ports
4   sc_out<bool> F;    // Output signal port
5
6   // Four instances of the Nand module
7   Nand n1, n2, n3, n4;
8
9   // Three Boolean ''wires''
10  sc_signal<bool> S1, S2, S3;
11
12  // No definitions of processes needed. All functionality
13  // is already implemented in the NAND gates
14
15  SC_CTOR(Exor) : n1(''N1''), n2(''N2''), n3(''N3''), n4(''N4'')
        {
16
17    // Connect the wires
18    N1.A(A);
19    N1.B(B);
20    N1.F(S1);
21
22    N2.A(A);
23    N2.B(S1);
24    N2.F(S2);
25
26    N3.A(S1);
27    N3.B(B);
28    N3.F(S3);
29
30    N4.A(S2);
31    N4.B(S3);
32    N4.F(F);
33  } // End of constructor
```

```
34 };
```

Listing 2.3: A SystemC module of an EXOR gate using 4 NAND gates.

Just like the Nand module, the Exor module has two Boolean input signal ports and one Boolean output port, defined on lines 3–4. The four instances of the NAND gate are declared on line 7, and the three inner "wires" are represented by Boolean signals declared on line 10. In the constructor the wires are connected to the proper ports of the modules. For example, on lines 18–20 we connect Exor input signal A to the input port of `N1` named A. Exor input signal B is connected to input port B of `N1`, and signal $S1$ is connected to the output port of `N1`, which is named F (see Listing 2.3). We connect similarly the remaining Nand modules.

Notice that there is no ambiguity between the input and output signal ports of the EXOR gate and the NAND gates, even though they all share the names A, B and F. The input ports of the NAND gates are defined inside the modules, and are accessed through the instance names of the modules, e.g., `N1.A` or `N3.F` using the `C++` operator "`()`", while the input ports of the Exor module are defined locally and can be accessed directly. Also notice that the Exor module does not define any processes. The processes already defined in the Nand module carry out the necessary functionality. This example illustrates that complicated models can be easily built from existing components with minimal effort.

2.1.3 Interfaces, ports, and channels

In traditional hardware models all communication and synchronization between processes is done using signals. SystemC allows the designers to raise the level of abstraction significantly, allowing the processes to exchange arbitrary data. Communication is done using *interfaces*, *ports*, and *channels*, which work together to hold and transmit data.

Interfaces

SystemC interfaces are sets of "operations", represented by functions. Only the function name, parameter, and return value are specified, without any restrictions how the operation is to be implemented.

A commonly used interface in SystemC is `sc_signal_in_if<T>`, which provides a virtual function `virtual const T& read() const`, allowing reading of arbitrary data (represented by the template argument `T`). Another commonly used interface, `sc_signal_inout_if<T>`, provides a virtual function `virtual void& write(const T&) = 0` that allows writing of arbitrary data of template type `T`. The particular way of carrying out the operations specified in the interface is determined by each channel implementing the interface.

Ports

Ports in SystemC allow modules to connect to and communicate with their environment. Ports are actual objects that are instantiated inside modules, and pass communication requests from the module to the channel. Having an intermediary object that handles the communication allows the module's processes to remain ignorant about the channel that carry out the communication, as long as the processes use the correct operations as defined in the interface.

Each port declaration specifies the operations that the port is able to perform, i.e., the port's interface. For example, the following statement: `sc_port< sc_signal_in_if<bool> > p;` declares a port p that can access a channel using the functions declared in the `sc_signal_in_if<bool>` interface. As discussed earlier, this interface provides the operation `const bool& read()`. Thus, we can use the port to read a Boolean value from the channel attached to the port by calling `p.read()`.

In the two examples presented earlier in Listing 2.2 and Listing 2.3, the code implementing the NAND gate and the EXOR gate contains port declarations that han-

dle the communication of the module with the environment. For example, lines 3–4 in Listing 2.3 declare **sc_in <bool>** A, B and **sc_out<bool>** F. Here **sc_in <bool>** is actually shorthand notation for **sc_port< sc_signal_in_if<bool> >**, and similarly, **sc_out<bool>** is shorthand for **sc_port< sc_signal_inout_if<bool> >**. Calling A.read() and F.write() (lines 7–9 in Listing 2.2) is allowed because the corresponding ports use the **sc_signal_in_if** and **sc_signal_inout_if** interfaces.

Channels

While interfaces describe what operations are available at the ports, channels define how those operations are implemented. Different channels can implement the same interface, and one channel can implement multiple interfaces. As an example, the implementation of an EXOR gate in Listing 2.3 uses three "wires" of type **sc_signal<bool>** to connect the Nand modules; **sc_signal<T>** is a pre-defined SystemC channel that implements both **sc_signal_in_if<T>** and **sc_signal_inout_if<T>** interfaces.

Producer-consumer model

The next example shows communication between modules through a shared channel. The model consists of two modules, Producer and Consumer, and they communicate using a bounded capacity FIFO. We first define the interfaces through which the modules communicate, then show the code implementing the Producer and the Consumer, and finally present the code for the FIFO.

```
1 template <class T>
2 class write_if : virtual public sc_interface {
3
4 public:
5   virtual void write(T& token) = 0;
6   virtual void reset() = 0;
7 };
```

```
8
9  template <class T>
10 class read_if : virtual public sc_interface {
11
12 public:
13   virtual T& read() = 0;
14   virtual int num_available() = 0;
15 };
```

Listing 2.4: Communication interfaces for the producer-consumer model

Listing 2.4 shows a typical SystemC interface definition. It is required that all user-defined interfaces extend **sc_interface** (line 2, 10). In SystemC the operations allowed on the interfaces must be encoded as pure virtual functions. The implementing channel is required to provide concrete definitions of those functions.

Every second the `Producer` (Listing 2.5) writes a character to the FIFO with 50% probability, and every second the `Consumer` (Listing 2.6) reads a character from the FIFO with 50% probability. It is important to point out that the 1-second intervals apply only to simulation time and not to wall-clock time. Wall-clock time is the actual time it takes to run the compiled executable model, and simulation time reflects the apparent time as perceived by the SystemC processes in the model. In the case of the producer-consumer model we can simulate thousands of seconds of simulation time in one wall-clock second.

```
1  SC_MODULE(Producer) {
2    // Definitions of ports
3    sc_port< write_if<char> > out;     // Output port
4
5    // Definitions of processes
6    void producer_process() {
7      const char* str = ''Hello world!'';
8      const char* p = str;
9
10     while (true) {
11       if (rand() % 2) {
12         out.write(*p);
13         p++;
14         if (!*p) {
15           p = str;
```

```
16          }
17        }
18
19        wait(1, SC_SEC);
20      }
21    }
22
23    SC_CTOR(Producer) {
24      // Process registration, sensitivity lists, etc.
25      SC_THREAD(producer_process); //  thread process
26
27      // No sensitivity list
28    }
29  };
```

Listing 2.5: A SystemC model of a Producer

The producer has only one port (line 3) that uses the interface `write_if<char>` declared earlier, and only one process (lines 6–21). The infinite **while**-loop writes a character through the output port and then suspends itself explicitly (line 19) using a call to the built-in function **wait()**. Time in SystemC is defined using a numeric value and a time unit; in this case, **SC_SEC** is the pre-defined unit corresponding to seconds. Since `producer_process()` contains a **wait()** statement, it has to be declared a thread process (line 25). Also notice that no sensitivity list is needed. Once triggered, the `producer_process()` never terminates, thus it will not need to be triggered again.

The `Consumer` module is presented next in Listing 2.6.

```
1  SC_MODULE(Consumer) {
2    // Definitions of ports
3    sc_port< read_if<char> > in;     // Input  port
4
5    // Definitions of processes
6    void consumer_process() {
7
8      while (true) {
9        if (rand() % 2) {
10         std::cout << in.read();
11       }
12
```

```
13        wait(1, SC_SEC);
14      }
15    }
16
17    SC_CTOR(Producer) {
18      // Process registration, sensitivity lists, etc.
19      SC_THREAD(consumer_process); //  thread process
20
21      // No sensitivity list
22    }
23  };
```

Listing 2.6: A SystemC model of a Consumer

The `Consumer` model is very similar to the `Producer` model. An input port is declared on line 3 and the functionality of the module is implemented on lines 6–15. The `consumer_process()` uses an infinite **`while`**-loop with explicit suspension of execution (line 13), and thus it must be declared a thread process (line 19).

One challenge presented by the `Producer` and the `Consumer` modules is that characters are produced and consumed nondeterministically. Since the FIFO has bounded capacity, it may eventually be full when the `Producer` attempts to store another character, or it may be empty when the `Consumer` attempts to read another character. The design of the FIFO, presented in Listing 2.7, handles this via *blocking* `read()` and `write()` operations.

```
1  template<class T>
2  class fifo : public sc_channel,
3               public write_if<T>,
4               public read_if<T> {
5
6  private:
7    int max = 10;  // Capacity of the FIFO
8    T* data[max];
9    int num_elements = 0;
10   int first = 0;
11   sc_event write_event, read_event;
12
13 public:
14
15   // Implementation of the read_if
```

```
16    virtual T& read() {
17      if (num_elements == 0) {
18        wait(write_event);
19      }
20
21      T* token = data[first];
22      -- num_elements;
23      first = (first + 1) % max;
24      read_event.notify();
25      return token;
26    }
27
28    virtual int num_available() {
29      return num_elements;
30    }
31
32
33    // Implementation of the write_if
34    virtual void write(T& token) {
35      if (num_elements == max) {
36        wait(read_event);
37      }
38
39      data[(first + num_elements) % max] = &token;
40      ++num_elements;
41      write_event.notify();
42    }
43
44    virtual void reset() {
45      num_elements = 0;
46      first = 0;
47    }
48
49  };
```

Listing 2.7: A SystemC model of a (blocking) FIFO

Like every user-defined channel, `fifo` must derive from the pre-defined class **`sc_channel`**. In order to handle the communication between the `Producer` and the `Consumer`, the `fifo` must also implement the interfaces `write_if` and `read_if`. This is declared explicitly on lines 2–4.

The `fifo` defines **`private`** local data to store tokens and to keep track of the total number of tokens, as well as the location in the array where the next token

will be stored (lines 7–11). The rest of the code implements the functions defined in `read_if` (lines 16–30) and `write_if` (lines 34–47).

The implementation of the `read()` function checks if there is something to read, and if there are no tokens currently in the FIFO, the function suspends execution with a call to **`wait`**`(write_event)` (line 18). Execution can resume only after `write_event` has been *notified*. The intended behavior of this piece of code is that execution will resume only after there is something to read from the FIFO. Ensuring that this is the case is the responsibility of the designer of the FIFO. In this case `write_event` is notified only on line 41 right after a token is placed in the FIFO, thus we can expect correct behavior from the code.

The implementation of `write()` similarly checks if there is space for one additional token in the FIFO. If this is not the case, execution is suspended until the FIFO is nonempty, by issuing a call to **`wait`**`(read_event)` on line 36. The expectation of the designer is that `read_event` is notified if and only if there is space in the FIFO. The only place where `read_event` is notified is on line 24, right after a token has been consumed from the FIFO, thus the designer can expect the code to execute correctly.

2.1.4 SystemC events

As seen in Listing 2.7 (line 11), SystemC events are represented by objects of the pre-defined class **`sc_event`**. An event determines whether and when a process is triggered or resumed, and usually represents a *condition* that may occur during simulation. The meaning of each event object is determined by the designer, and reporting that the condition represented by the event object has occurred is done by *notifying* the event object.

A very commonly occurring condition in SystemC is the change of value of a signal. Internally, SystemC defines for each **`sc_signal`** an associated event, accessible via function call `value_changed_event()`, that is notified whenever the value of the

signal is written or modified [IEE06]. For example, the implementation of a NAND gate in Listing 2.2 defines a process that is sensitive to input ports; implicitly, the process is sensitive to the `value_changed_event()` associated with the signals connected to each port.

The effect of the notification of an event `e` causes all processes that are sensitive to `e` (or have called **`wait(e)`**) to be triggered or resumed. There may be, however, some delay from the instance when `e.`**`notify`**`()` is issued and when its effect takes place, depending on the argument passed to **`notify`**`()`:

- Calling **`notify`**`()` without arguments constitutes *immediate notification.* Any and all processes that are sensitive to the event are triggered before the function call **`notify`**`()` returns. Triggering a process is not equivalent to starting it, however. In SystemC there is a pool of processes that are ready to execute in the current simulation cycle. Triggering a process simply adds the process to this pool. This notion will be made more precise in Section 2.2.

- Calling **`notify`**`()` with a zero-valued time unit (e.g., `(0,` **`SC_SEC`**`)` or, more commonly, `SC_ZERO_TIME`) delays the effect of the event notification until all currently triggered processes have finished executing. Between the instance when **`notify`**`()` is called and when its effect takes place there may be an arbitrarily long delay (in terms of wall-clock time) during which all other runnable processes take their turn to execute. During that delay, however, the simulation clock does not advance, so the notification of the event and its effect happen in the same instance of time as measured by the simulation clock. This type of event notification is called *delta-delayed* notification.

- Calling **`notify`**`()` with a non-zero argument delays the effect of the notification by the requested number of time units. The argument is added to the current value of the simulation clock, and the event is put in a queue, ordered by the scheduled notification time. This type of event notification is called *time-delayed*

notification.

If the event notification is pending (i.e. it is delta-delayed or time-delayed) it can be *canceled*, which removes any pending effect of the event. It is also allowed to wait for an event for some bounded amount of time, and then resume execution even if the event was not notified. For example, a process issuing a call to `wait(2, SC_SEC, some_event)` resumes execution after 2 seconds of simulation time, unless `some_event` is notified earlier.

2.1.5 Data types

A SystemC model can use all C++ data types. This is convenient for the designers of the model's software, but not sufficient for hardware designers. One example why this is the case is given by Grötker et al. [GLMS02]: in software, a loop ranging from 0 to 31 would be indexed by an **int** variable, even though only 6 bits would be sufficient. Implementing this in hardware would require a 6-bit register and a 5-bit adder, and using a 32-bit register and a 32-bit adder would be wasteful. To accommodate the needs of hardware designers, SystemC provides a rich set of hardware-oriented data types.

For example, fixed-precision integral type **sc_int**<W> and **sc_uint**<W> encode numbers from -2^{W-1} to $(2^{W-1}-1)$ and from 0 to $(2^{W}-1)$ accordingly. These types are limited to 64 bits. Arbitrary-precision integral types `sc_bigint<W>` and **sc_biguint**<W> can fit and operate on integers bigger than 64 bits, at the cost of worse performance.

The native C++ type **bool** is often implemented as a **short int**, so SystemC provides a replacement **sc_bv**<W> which can also encode bit-vectors. 4-valued logic values can be encoded using **sc_logic** and a vector of logic values can be encoded using **sc_lv**<W>. SystemC also provides a library for fixed-point arithmetic with corresponding data types.

2.2 SystemC as a simulation environment

The simulation semantics of SystemC is similar to that of hardware-design languages like VHDL. It is based on the principle that new events are generated as a result of the execution of processes, which were triggered by earlier events. This defines partial order on the executing processes and the notification of SystemC events. This section explores in detail the simulation semantics of SystemC.

2.2.1 Parallel execution

In order to simulate parallel execution, SystemC executes processes sequentially, but hides the effect of each process until all "parallel" processes have been executed and each one has reached a synchronization point: either a call to `wait()` or process termination. For example, values written to signals are not immediately readable by other processes executing "in parallel" with the writing process. Execution of processes is controlled by the SystemC *kernel*.

The (sequential) execution of "parallel" processes defines one phase of the overall execution of the model, called *evaluation phase*. Before the evaluation phase begins, the SystemC kernel creates a list of all processes that it must run. Processes on this list are called *runnable* processes. Nondeterministically and sequentially, the kernel removes a process from the list and gives it execution control; the process is now *running*. As soon as the running process suspends itself it becomes *waiting*. Control is transferred back to the kernel which then removes another process from the list and gives it execution control. The simulation semantics imposes *non-preemptive cooperative execution* of processes, that is, once the kernel gives a process execution control the kernel cannot take it back. The process must explicitly give back the control by reaching a synchronization point.

During its execution, a running process may request the notification of an event. If the request is for an *immediate notification* (see 2.1.4), all processes waiting on the event become runnable immediately, and are added to the pool of runnable processes.

If the event notification is time-delayed or delta cycle-delayed, it is collected by the kernel but not acted upon during the evaluation phase.

It is also possible to have two or more processes requesting the notification of the same event during the same evaluation phase. In those cases the request with earlier effect survives, and the other one is canceled. For example, if a delta-delayed and a time-delayed notification are requested in the same evaluation phase, the time-delayed notification is canceled while the delta-delayed notification remains. Likewise, an immediate notification cancels any pending time- or delta-delayed notification requests.

2.2.2 Signals and channels

After the end of the evaluation phase the kernel makes visible any changes of values of signals and channels, during the so called *update phase*. If a process wrote a new value to an **sc_signal**, the new value becomes the value of the signal. This change of value notifies the signal's `value_changed_event()` using a call to **notify**`(SC_ZERO_TIME)`. If there are processes that are sensitive to changes of the signal, the effect of event notification is to make those processes runnable. The event notifications are collected by the kernel but are not acted upon in this phase.

SystemC distinguishes a type of channel called *primitive channels*. Those are channels that perform simple communications or synchronization, and in particular, do not define their own processes. **sc_signal** is one example of a primitive channel, but the user can define their own by extending **sc_prim_channel**. Every primitive channel inherits the function **sc_prim_channel**`::request_update()`. During the evaluation phase of the simulation, a primitive channel may `request_update()`, which indicates to the kernel that the channel should be placed in the kernel's update queue. During the update phase the kernel removes nondeterministically channels from the update queue and calls `update()` on them. This gives user-defined primitive channels the opportunity to delay the effect of any updates of the values in the

channel until the end of the evaluation phase.

2.2.3 Delta notifications and delta-cycles

After updating all channels in the update queue, the SystemC kernel handles all pending delta-delayed notification requests. For each such request the kernel determines which, if any, processes are triggered by the event; those processes are added to the list of runnable processes to be executed in the subsequent evaluation phase. If the list of runnable processes is non-empty, the kernel starts another evaluation phase.

The evaluation phase, the update phase and the delta notification phase together form a *delta-cycle*. A delta-cycle represents execution in suspended time within the same clock cycle. In other words, the simulation clock does not advance during the execution of delta-cycles. It is quite common to have multiple delta-cycles within the same cycle of the simulation clock. SystemC, like VHDL, uses delta-cycles to impose a partial order on simultaneous actions. It is a common device for interpreting zero-delay semantics [GLMS02].

2.2.4 Timed notifications and advance of simulation time

If at the end of a delta-cycle there are no runnable processes, the kernel checks for any pending time-delayed event notifications. If there are none, the simulation is finished. Otherwise, the kernel advances the simulation time to the earliest time-delayed event and makes runnable all processes that are waiting for the event. It then starts another delta cycle. Notice that the only way to advance the simulation time is through timed events. If a model does not use timed events, simulation time will remain at 0 regardless of how many delta cycles are executed. Timeless models are often used during the early stages of design, and will be discussed further in Chapter 3.

2.2.5 The start of a simulation

The simulation of a SystemC model starts with an *initialization phase.* All primitive channels execute their `update()` methods, which may trigger delta-delayed events. All processes are made runnable with the exception of processes that explicitly request not to be initialized. The delta-delayed events then take effect, potentially making runnable some of the processes that requested not to be initialized. This concludes the initialization phase, and the kernel enters into the first delta-cycle.

2.2.6 Simulation semantics of SystemC

The simulation semantics of SystemC, which is defined in natural language in [IEE06], is presented in pseudo code below.

```
1: PC ← all primitive channels
2: P ← all processes
3: R ← ∅ /* Set of runnable processes */
4: D ← ∅ /* Set of pending delta notifications */
5: U ← ∅ /* Set of update requests */
6: T ← ∅ /* Set of pending timed notifications */
7: /* Start initialization */
8: /* Collect all update requests in U */
9: for all chan ∈ U do
10:   run chan.update()
11: end for
12: for all p ∈ P do
13:   if p is initializable and p is not clocked thread then
14:     R ← R ∪ p /* Make p runnable */
15:   end if
16: end for
17: for all d ∈ D do
18:   D ← D \ d
19:   for all p ∈ P do
20:     if p is triggered by d then
21:       R ← R ∪ p /* Make p runnable */
22:     end if
23:   end for /* End of Initialization phase */
24: end for
```

```
25: repeat
26:   while R ≠ ∅ do /* New delta-cycle begins */
27:     for all r ∈ R do /* Evaluation phase */
28:       R ← R \ r
29:       run r until it invokes wait() or terminates
30:     end for
31:     for all chan ∈ U do /* Update phase */
32:       run chan.update()
33:     end for
34:     for all d ∈ D do /* Delta notification phase */
35:       D ← D \ d
36:       for all p ∈ P do
37:         if p is triggered by d then
38:           R ← R ∪ p /* p is now runnable */
39:         end if
40:       end for
41:     end for
42:   end while /* End of delta-cycle */
43:   if T ≠ ∅ then
44:     Advance the clock to the earliest timed delay t.
45:     T ← T \ t
46:     for all p ∈ P do /* Timed notification phase */
47:       if t triggers p then
48:         R ← R ∪ p /* p is now runnable */
49:       end if
50:     end for
51:   end if
52: until end of simulation
```

The execution of a SystemC application starts with the *Elaboration phase*, during which all modules are instantiated and channels are bound to ports, and some channels may register a `request_update()` (line 8). Then the kernel enters the *Initialization phase* (lines 9–23). During the Initialization phase all channels with pending updates are updated (lines 9–11), all initializable SC_THREADs and SC_METHODs are made runnable (lines 12–16), and pending delta notifications cause their dependent processes to become runnable (lines 17–23). Next the kernel starts a delta-cycle and runs all runnable processes one at a time (*Evaluation phase*, lines 27–30). During this phase pending channel updates are collected in U, and pending event notifications

are collected in D. The evaluation phase is followed by an update phase (lines 31–33) where all collected channel update requests are executed and writes to signals take effect. After that the kernel enters the delta-notification phase (lines 34–42) where notified events trigger their dependent processes. Note that immediate notifications may make new processes runnable during the execution of lines 27–30.

If at this point there are runnable processes the kernel loops back to line 26 and starts another evaluation phase and a new delta-cycle. Alternatively, if there are no more runnable processes, the kernel advances the simulation clock to the earliest timed-delay notification (essentially, a notification that is explicitly set to be notified after some delay). All processes sensitive to this event are triggered (lines 46–50) and the kernel loops back to line 25 and starts a new delta-cycle. This process is repeated indefinitely, unless the designer has specified a fixed simulation time or all processes have terminated.

2.3 Summary and discussion

This chapter presents a brief introduction to the fundamentals of SystemC. It describes the main building blocks of SystemC models (modules), and the fundamental communication primitives (ports, interfaces, and channels). The simulation semantics of SystemC, consisting of the phases of the kernel and the connection between events and processes, is also presented. There are more details about SystemC than can be included in a single chapter; the interested reader should consult Grötker et al. [GLMS02] for further information. The Language Reference Manual [IEE06] is also a great resource.

One key point in this chapter is that the execution of a SystemC model is an interplay between the user code and the kernel, mediated by SystemC events. This interplay forms the basis of the monitoring framework presented in the remainder of this thesis.

Chapter 3

Specification Primitives for SystemC

The first requirement for an assertion-based verification framework for SystemC is a formal specification language that can describe the expected behavior of the model's execution. In the past, design specifications have been given in natural language documents [Var07], but natural language is inherently ambiguous and it is easy to miscommunicate or misinterpret the intended functionality of the design. In current design flows the specification provides a baseline for many implementations developed by different organizations, and a formal language with precise semantics is a must [BGM04]. This chapter describes some existing specification languages and points out why they are not sufficient to express properties of SystemC models. It then shows how those languages can be augmented to express SystemC properties.

3.1 Existing languages

3.1.1 Brief history of assertion language standards

Industrial languages that can specify not only Boolean conditions (such as the C++'s `assert(ptr != NULL)`) but also their relationship over time are a fairly modern invention. Their theoretical foundations, however, go back to the days of Aristotle [OH95].

Ancient Greeks and medieval philosophers were interested in the connection between time and logic, but the interest disappeared during the Renaissance. Development of modern temporal logic did not start in earnest until Prior's work (see, e.g., [Pri57]), who extended propositional logic with two temporal connectives, one for the past and one for the future. Using logic to specify the behavior of circuits

was envisioned by Church [Chu57]. He observed that the execution of a circuit can be represented by an infinite word, where each letter represents the set of all Boolean variables that hold at each step of the circuit. In modern terms, Church was describing a trace of execution. Church posed a decision problem: given a circuit and a formula, does the formula hold in all executions of the circuit. To answer this problem, Büchi [Büc62] showed a non-elementary construction, which involved automata on infinite words (Büchi automata). Applying temporal logic to specify correctness of non-terminating programs was proposed by Pnueli [Pnu77]. Pnueli introduced Linear Temporal Logic (LTL): propositional logic with next (**X**) and until (**U**) operators. An alternative way of specifying temporal properties, called Computation Tree Logic (CTL), was proposed by Clarke and Emerson [CE81]. Vardi, Wolper, and Sistla [WVS83, VW94] opened the way for practical use of LTL by replacing the then-current non-elementary translation from LTL to (nondeterministic) Büchi automata with an exponential construction.

In the early 1990s, researchers at IBM developed the temporal language Sugar, which was a syntactic simplification (or sugaring) of CTL [Fos08]. Researchers at Intel developed the ForSpec temporal language [AFF+02], what was based on LTL. Both Sugar and ForSpec were donated to Accellera Formal Verification Technical Committee as candidate languages for standardization. This led to the creation of the Property Specification Language (PSL) [EF06], which became IEEE Standard 1850-2007. PSL adopted the syntax of Sugar and the linear temporal semantics of ForSpec. As a nod to Sugar, a branching-time extension was also included [Var09].

In a parallel thread, in 2002 Accellera was working on creating a new version of Verilog, that would combine hardware verification and hardware description in one language [Fos08]. This effort led to the creation of SystemVerilog, which became IEEE Standard 1800-2005. A major component of SystemVerilog was an assertion language called SystemVerilog Assertions (SVA) [VR05]. SVA adopted many of the features of PSL, so the remainder of this thesis focuses on PSL as a representative of

both specification languages.

3.1.2 Overview of PSL

PSL is a powerful assertion language, consisting of four layers of descriptions. The *Boolean layer* is used to build expressions that are, in turn, used by the other layers. It consists of Boolean expressions of the underlying modeling language, as well as the symbols ***true*** and ***false***. These expressions describe the system state in a single instance. Since different modeling languages use different syntax, PSL defines different *flavors*. The most recent version of PSL, IEEE 1850-2005, defines VHDL, Verilog, SystemVerilog, SystemC, and General Description Language (GDL) flavors.

The *temporal layer* describes the temporal relationships between expressions. It defines *Sequential Extended Regular Expressions* (SEREs), which describe chains of events or Boolean primitives. Some aspects of SEREs are equivalent to conventional regular expressions, for example, the `*` operator. Others are SERE-specific, for example, the *fusion* operator `;`, which requires that both SEREs much hold together, and the length-matching operator `&&`, which requires that both argument SEREs occur, and both SEREs start and terminate at the same time. The temporal layer also includes the temporal operators from LTL like `always`, `eventually`, etc.

The *verification layer* includes directives which instruct the verification tool how to use the properties and sequences, for example, `assert` and `cover`.

The *modeling layer* is used to model the behavior of design inputs and to model auxiliary hardware (e.g., a state machine) that is not part of the design, but is needed for verification.

One of the most important features of PSL is the concept of clock context. Every property, sequence, and built-in function is evaluated only in instances in which the clock context holds. Thus, *clock expressions*, distinguished by the operator `@`, allow the user to set the granularity of time. Any Boolean expression can be used as a clock expression. The *base clock context* is ***true***, i.e., the granularity of time is determined

by the verification tool. An event-driven simulation typically has a fine-grained model of time, while a cycle-based simulation typically has a more coarse-grained model of time.

3.1.3 Overview of SystemC Verification Standard

The SystemC Verification Standard (SCV) [IS03] was proposed by the SystemC Verification Working Group as a "robust standard for developing test benches and verification [intellectual property] for [system-on-chip] designs." [IS03] The standard suggests a methodology for creating test benches by connecting the MUV to a test-generating object via a special channel called *transactor*. The transactor is responsible for translating each test vector to a sequence of read/write calls using the interface provided by the MUV. SCV also includes a library of objects that facilitate recording of values of variables as they are sent by the transactor to the MUV, and an API that allows the transactor to indicate when a new transaction starts and when a transaction ends. The recorded values can be stored in a database or output to a user-selected file. SCV does not provide a mechanism for generating the transactor automatically, so it needs to be constructed manually by the user following the guidelines proposed by SCV.

One of the main contributions of SCV is the ability to generate constrained and weighted randomized variables. The user can specify a range and a particular distribution of the random variables, as well as to indicate that certain values should never be generated. SCV then provides a mechanism for getting an unbounded number of values subject to the criteria specified by the user. The current version of the standard does not provide support for specifying or checking temporal properties. Its focus is on test generation and recording.

3.1.4 Overview of NSCa and TLA

Kasuya et al. from Jeda Technologies argued that "PSL and SVA mainly target temporal transitions at the cycle-accurate modeling layer. The primitives in both

languages are designed for checking the cycle-by-cycle behavior of signals." [KTZ06] Jeda Technologies went on to develop temporal languages for SystemC that allow specifications at higher levels of abstraction. They describe two languages for SystemC inspired by SVA. NSCa is an adaptation of SVA and is aimed at cycle-level verification. TLA is a variant aimed at transaction-level verification [KT07].

The NSCa (short for "native SystemC assertions") library defines an SVA-like language for specifying temporal properties of SystemC models. At the atomic level the properties can refer to the values of any signal or variable. The properties can be placed anywhere in the code and the values referred to in the property must be in scope. In addition, NSCa allows the notification of any signal to be used as an atomic expression in temporal formulas.

NSCa is not applicable to clock-less models, so Kasuya and Tesfaye propose another language (TLA, short for "transaction-level assertions") that exposes the sequence of steps of the model without depending on the clock. This language requires the user to setup callbacks (essentially, function calls) that "report an event occurrence at the point of transaction processing" [KT07].

Both languages require that the assertions be added to the source code of the model at the location where the assertion is expected to hold.

3.2 Related work

There have been a few attempts to adapt temporal languages to SystemC. Ecker et al. [EES+05] describe an implementation of a SystemC Assertion Library inspired by Accellera's Open Verification Library. The library defines 11 properties, most of which are invariance properties and a few are simple temporal templates. The library does not provide a mechanism for defining new temporal templates.

Große and Drechsler [DG02, GD03] use a C++ representation of bounded LTL formulas, which are then compiled together with the SystemC model. Their approach is limited to gate-level models. Before the start of each simulation, the SystemC

kernel polls each gate and obtains its type and primary inputs. Große and Drechsler's approach is to use this information to build a Binary Decision Diagram [Bry86, Bry92] representation of the circuit and to use a fixpoint algorithm to check if an illegal state is reachable. Since most SystemC models are designed several levels of abstraction above gate-level, this approach has very limited practical application.

Traulsen et al. [TCMM07] translate SystemC models into Promela models, which enables them to use the model checker Spin to verify LTL properties. Habibi, Gawanmeh, and Tahar advocate using PSL [HGT04] and SVA [HT04] for SystemC, but do not propose an adaptation.

Karlsson et al. [KEP06] use Petri-nets [Pet81] to create a formal representation of the SystemC model at the statement level (i.e., each statement is represented by one place and one transition). Ecker et al. [EES+06] propose an extension of PSL and SVA to express properties of **sc_event**s. A disadvantage for both of these two approaches is that the formal model has one level of abstraction.

Pierre and Ferro's framework [PF08, PF10] samples at the statement level, and then only considers those states which are relevant to the property, thus providing a somewhat more flexible temporal resolution. This approach shares the same drawback as all existing approaches: the state of the library code and the state of the simulation kernel are not taken into consideration.

3.3 Deficiencies in existing languages

Current works on temporal languages for SystemC are lacking in several respects. This section identifies the weaknesses and gives a brief sketch how they can be addressed, and the rest of this chapter contains a detailed discussion.

3.3.1 Inflexible abstraction levels

A temporal language should be adaptable to different levels of abstraction in the design of the model. One of the strengths of SystemC is modeling at different levels

of abstraction; during the design process the model typically gets refined, evolving from a system-level model to a gate-level model. Using multiple languages, one for each abstraction level, makes it difficult to reuse or adapt the specifications as the model is refined. Jeda Technologies' solution to use NSCa and TLA for different levels of abstractions is not flexible enough for a model under active development.

SystemC also allows modules developed at different levels of abstraction to be executing together in the same model. For example, it is possible to execute a Producer module driven by a clock signal, communicating with a Consumer module driven only by the module's inputs. The specification language should be able to specify properties that relate to the execution of a model with mixed abstraction levels.

This work identifies a new set of atomic propositions that allow the specification to refer to the execution of the SystemC kernel. It is shown how exposing the kernel's operation is essential for specifying properties at different temporal resolution.

3.3.2 Lack of mechanisms for user-code specification

The existing specification languages are approaching the issue mostly from a hardware perspective and are ignoring the fact that a SystemC model is, fundamentally, a C++ program. SCV, for example, adopts a "black-box" view of the MUV and only considers the input/output variables passing through the transactor. There is a large body of work on specification and model checking of Java, C++ and C code (e.g., [BR02, BCC+05, CDHR02, HJMS03, BCH+04]) and the specification primitives used there should be adapted for SystemC.

This work addresses these concerns by extending the Boolean layer of PSL-like specification languages with primitives that refer to the state of the user code, for example, values of module variables, and Boolean atomic propositions that refer to the control flow of the execution, for example, the call or return of a particular function.

3.3.3 Lack of definition of execution trace

All (linear) temporal languages are interpreted over execution traces, therefore before we can define the semantics of temporal properties for SystemC we need a precise definition of an execution trace. None of the existing languages addresses this issue. Traditionally, a trace has been defined as a sequence of states in the execution of the model, but there has been remarkably little discussion in the literature about the definition of SystemC traces.

Hardware-oriented languages such as PSL or SVA usually assume an underlying notion of a clock-cycle-level trace. However, such an approach fails to take into account the unique simulation semantics of SystemC, which allows for a much finer grained temporal resolution. For example, algorithmic-level SystemC models are often timeless, with the simulation being completely driven by events and the simulation clock making no progress during the whole simulation [GLMS02, MMMC06]. In fact, the whole simulation can consist of a single delta cycle, if the simulation is driven solely by immediate event notifications. Thus, clock-cycle-level temporal resolution is clearly inappropriate for such models. Also note that PSL's default sampling rate is tool dependent, so the semantics of a PSL specification may change between two different implementations of the SystemC kernel.

This work gives a precise definition of a SystemC trace. Intuitively, a trace is a sequence of states in the execution of the model. First, we abstract the simulation kernel and define its state with respect to this abstraction. Second, we recognize that one needs to distinguish between the SystemC model developed by the designer and the set of SystemC libraries used by this model. While the state of the model is fully detailed, the libraries are modeled only at the level of their exposed interfaces. Finally, we define the notion of a trace with respect to these abstractions of the kernel and the model's code (both user code and library code).

3.4 Kernel-level primitives

3.4.1 Kernel phases

It is not immediately clear why the state of the kernel needs to be referenced in SystemC specifications. For example, in the work of Kroening and Sharygina [KS05], the kernel is abstracted away completely. Each process is modeled as a labeled transition system, and the global system is defined as a product of these local transition systems. The transitions of the global system are defined according to the simulation semantics, which requires that "components must synchronize on shared actions and proceed independently on local actions" [KS05]. Under this model synchronization occurs when a process encounters a **`wait()`** or a **`notify()`** instruction. The observable behavior of their abstraction of execution matches well the execution of a SystemC model.

Similar philosophy has been adopted by Karlsson et al. [KEP06], who abstract the processes defined by the user code to Petri nets, and the SystemC scheduler is emulated by connecting the Petri nets in such a way that a process that terminates or suspends itself automatically triggers the next process. Ecker et al. [EES+06], and Pierre and Ferro [PF08], likewise do not model the kernel. Thus, on the surface, it may seem that taking into account the state of the kernel would only complicate the semantics.

This may sound reasonable at first, but one soon realizes that many important properties require some knowledge of the state of the kernel. A consistency property may be required to hold all the times, at the evaluation-phase boundary, at the delta-cycle boundary, or at a timed-cycle boundary. If the kernel is abstracted away completely, then there is no way to make these distinctions and specify the consistency requirement properly. We conclude, therefore, that the state of the kernel must be exposed to a certain extent, in order to enable the user to specify properties at different levels of abstraction. (For a discussion how such exposure is to be implemented, see

Chapter 4.) This approach of exposing the state of the kernel to some extent was taken by Moy et al. [Moy05, MMMC06]. Their work formalizes SystemC models in terms of communicating state machines, where the kernel is modeled as a particular state machine (Figure 3.1). Thus, the state of the kernel is exposed at an abstraction level corresponding to this specific state machine.

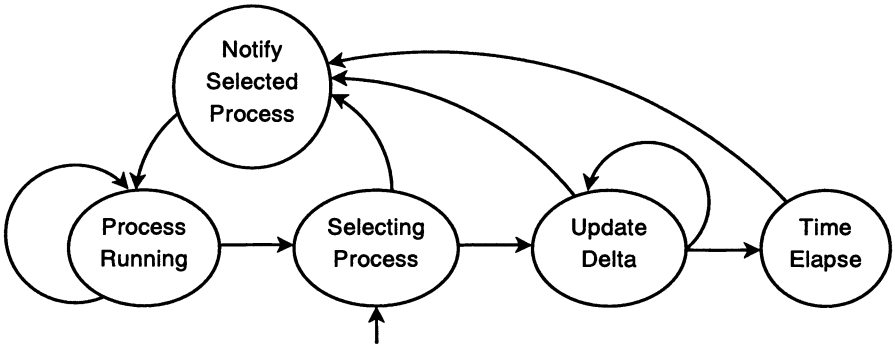


Figure 3.1 : Kernel states proposed by Moy et al.

Once one accepts the principle of exposing the kernel state, the question remains at what abstraction level to expose the kernel. Moy et al. offer a specific abstraction, but their choice is open to criticism. Their formalism is somewhat less detailed than the simulation semantics in SystemC's Language Reference Manual (LRM) [IEE06]. One could offer other abstractions of the kernel, but without some guiding principle such abstractions are also open to criticism. The guiding principle of this work is that the abstraction should abstract away the kernel implementation, but expose fully SystemC's simulation semantics, as described in Figure ?? in Chapter 2. A coarse abstraction might hide details that may be of importance to some users. Thus, an abstraction at the level of the simulation semantics is as generic as possible, enabling further abstraction if required by specific applications.

This thesis abstracts the simulation semantics, as described in Figure ?? in Chapter 2, by the state machine shown in Figure 3.2. This abstraction may seem, at first sight, to be somewhat too detailed. A simpler abstraction of the kernel would consider

each phase (*Initialization*, *Evaluation*, *Update*, *Delta notification*, *Timed notification*) as a separate state in a state machine. Why does the abstraction presented here has more states? The answer is that the simpler model does not expose the start and the finish of individual processes (all processes are run while the kernel is in the Evaluation phase, and similarly for the Update phase). This abstraction exposes the transfer of control between the kernel and the processes by splitting the Update end Evaluation phases into two sub-phases. Since SystemC uses the interleaving approach to concurrency, exposing the transfer of control is important. The abstraction of the kernel presented here is more detailed than Moy et al.'s, therefore giving the specification writer more expressive power.

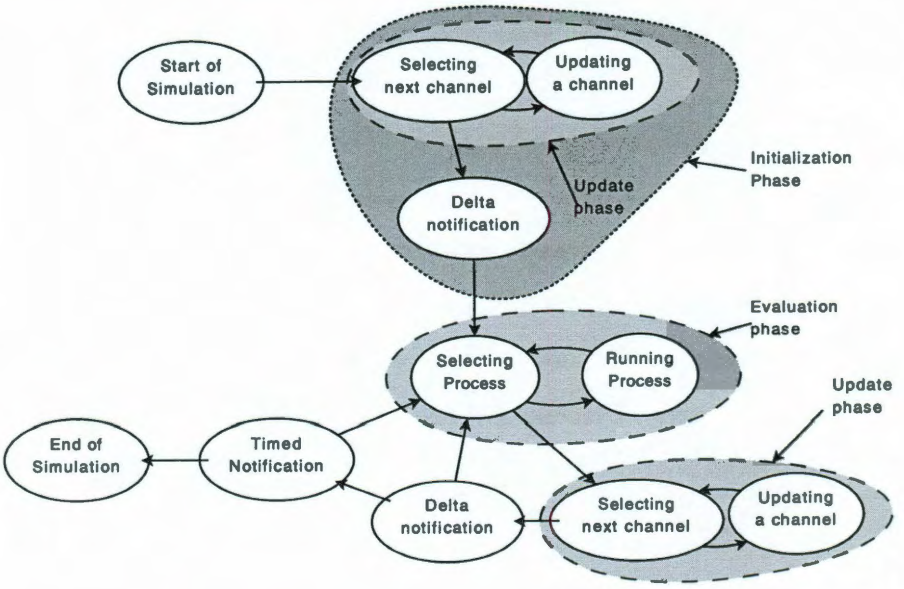


Figure 3.2 : Proposed Kernel States

One might argue that keeping track of all phases of the simulation semantics is unnecessary because very few properties relate to those specific phases. This thesis takes a generic approach and models all phases of the kernel that are described by the simulation semantics rather than try to pass judgment on which ones are the most important. As shown at the end of this chapter, the approach presented here

enables users to use coarser abstractions if needed. Rather than trying to anticipate all possible uses of SystemC, exposing the semantic fully is the most justifiable approach.

3.4.2 SystemC events

Recall that SystemC events are objects derived from the pre-defined class **`sc_event`**. A particular "waiting" process does not becomes "runnable" until the event on which the process is waiting is *notified*. For example, if a TLM channel is full, a thread that wishes to write to the channel may suspend itself by calling **`wait`**`(ok_to_put)`. As soon as there is free space on the channel, the channel notifies the `ok_to_put` event, and the waiting thread is moved to the pool of "runnable" processes among which the kernel selects the next process to run.

Most core SystemC objects have an associated event that indicates that some change has occurred. For example, an **`sc_signal`** has an event that is notified when the signal changes; an `sc_fifo` has an event for writing to and an event for reading from the channel; an **`sc_clock`**'s positive and negative edges are represented by events. Thus, events are the fundamental synchronization mechanism in SystemC, and keeping track of when a particular event is notified allows to pinpoint the instant in time when something important happens. In the particular example mentioned before, the user might want to specify that every time `ok_to_put` is notified the number of items in the channel is strictly smaller than the capacity of the channel.

PSL's and Jeda Technologies' treatment of events is to allow them to be used as clock expressions, though the issue when events are actually notified is not discussed explicitly. The position adopted by this work is that the fundamental role played by events in the execution of SystemC models justifies fully exposing event notification in the kernel's state. In essence, the instance in time when event notification takes place is elevated to the Boolean layer. This means that properties can refer directly to event notification, for example, specifying that `ok_to_put` is notified at least once every clock cycle.

3.5 User model primitives

3.5.1 Class data members

The state of the user model is the full state of the C++ code of all processes in the model, which includes the values of all class/module variables, the location counter, and the call stack. The perspective adopted in this work is that of "white-box validation", which means that the state of the model should be fully exposed. Thus, the property languages should consider all class members to have **`public`** access, including those that are declared as **`private`** or **`protected`**.

3.5.2 Statement-level primitives

As argued earlier, a property specification language for SystemC ought to consider SystemC as a system-level language, rather than a hardware-level language. This requires that the execution of a SystemC model be exposed also at the source-code level [BCH+04]. The specification writer should be allowed to refer explicitly to statements being executed by their syntax; for example, a specification should be able to refer to invocations of statements matching a particular regular expression.

3.5.3 Function calls

A popular style of modeling and a widely used library of pre-defined interfaces and objects, collectively referred to as *Transaction Level Modeling* (TLM) [Ghe06], is particularly important during the early stages of design exploration and prototyping. TLM abstracts communication by focusing on what data is transferred and which entities are communicating with each other, disregarding such details of the communication as specific protocols and timing. The duration of the function call, during which execution context is transferred from the initiating object to the callee, is often referred to as *transaction.* Exposing function calls is thus crucial.

Furthermore, as method invocation is central to the execution of object-oriented

code, the values of arguments passed to and returned from invoked methods should also be exposed, by exposing the values of the formal parameters of the method upon invocation. In essence, we are requiring traces of SystemC model to include both state and transition descriptions, in contrast to standard models of temporal logics that are typically state based [Pnu77]. By exposing both semantics and syntax of the model, we enable properties that relate source code and execution; for example, we can specify that for every port connected to a specific channel, the `write()` interface method call is passed only positive numbers as arguments. In order to simplify the process of referring to individual statements in what could be many thousands of lines of code, we provide several pre-defined labels for important locations. This is discussed further in Section 3.8.

3.6 Library code state

Most SystemC models rely heavily on external libraries (the TLM library, for example). These libraries encapsulate crucial components of SystemC models, such as signals and channels. When formalizing the notion of SystemC state, it needs to decide how to formalize the state of libraries. One approach would be to extend the white-box approach to library code, but users need to be familiar with libraries only at the API level, and not at the implementation level. Furthermore, while in many cases the code of the library is available, in others the library may be supplied as compiled code, thereby hiding the internal state.

Kroening and Sharygina [KS05] do not discuss how they handle library code. Moy et. al's approach [MMMC05, Moy05] is to provide specific state-machine models reflecting the functionality of TLM constructs. The benefit of this approach is that it preserves important information about the structure and behavior of the design. A major drawback is that this requires manual effort to develop formal models for libraries. These formal models may have to be revised when libraries are revised.

To attain generality, this thesis adopts the philosophy that library code should be

treated as a black box. For example, when specifying the behavior of a pre-defined library object, for example, `tlm_fifo`, the state of the queue should be exposed without exposing implementation details. Furthermore, the state of a library should be exposed only in terms of the API of that library. Consider, for example, the TLM 1.0 library. Properties should not have access to the state of the `tlm_fifo` other than via side-effect-free function calls, for example, via the `peek` method. Of course, when library source code is available, users can choose to treat it as a part of the user model and view it from a white-box perspective.

3.7 Execution trace

A SystemC trace is a sequence of states corresponding to the execution of the model. Such execution consists of an alternation of control between the kernel, on one hand, and the model and the libraries on the other hand. We have discussed so far how we formalize the state of the kernel, the model, and the libraries. It remains to discuss at what level of granularity we formalize the transition from one state to its successor.

When the kernel is executing, we follow transitions in the state machine described in Figure 3.2. When the kernel selects a process to run or a channel to update, control passes to that process, which then runs until it terminates or is suspended via a **`wait`** statement. With respect to transitions of processes, we follow the "large-step semantics" approach [Win93]. Under this approach we focus only on the overall effect of each statement, as opposed to considering the individual subexpressions. For example, `y = x++;` consists of two subexpressions (`y = x;` and `x = x + 1;`), but we ignore the valuations of the variables during the execution of the subexpressions. We believe that this matches the level at which programmers and verification engineers think about the source code. By following large-step semantics our framework may miss rare cases where a property is violated in a subexpression. For example, suppose that a program invariant requires that x must always be positive, and suppose that $x = 1$. During the execution of the expression `y = (x--) + (x++);` the value of

x is temporarily set to 0 by the `x--` subexpression, but since the value of x is restored back to 1 by the `x++` subexpression, no violation of the property will be reported. Modern design practices discourage the use of complex subexpressions that change the valuation of variables, therefore the choice of large-step semantics over small-step semantics is justified.

Finally, each invocation of a library method, for example, invoking a channel-interface method, is modeled to return in one step. This is consistent with the black-box view of libraries adopted here.

3.8 New specification primitives

From a high level point of view the execution of a SystemC model alternates between user code and the kernel, and the events provide a bridge between the two. Having given a definition of an execution trace, this thesis now defines a set of specification primitives for SystemC models that make explicit the transfer of control between the kernel and the user-code.

Notice that specification languages like PSL and SVA are already quite expressive temporarily, so no extension is necessary for their temporal layer in SystemC. Thus, the focus in this section is on extending the Boolean layer. The new set of primitives introduced here not only allows the specification of a richer set of properties, but also provides a uniform mechanism for controlling the temporal resolution of the specification language. Unlike Kasuya and Tesfaye's approach [KT07], which requires a separate language for clock-based and transaction-based models, the framework presented here allows for greater flexibility in the temporal granularity of the specification, and is general enough that it can readily be adapted for any temporal language with the notion of clock expressions.

Table 3.1 summarizes the new specification primitives.

specification_primitive	::=	*usercode_expression* | *kernel_expression*
usercode_expression	::=	*location_proposition* | *argument_primitive*
location_proposition	::=	[**before** | **after**] {code_label | *syntax_expression* }
		| *function_identifier:*{**entry** | **exit** | **call** | **return**}
argument_primitive	::=	*function_identifier* : non-negative_integer
syntax_expression	::=	$function_{\text{String} \rightarrow \text{Boolean}}$ (**curr_statement**, ...)
function_identifier	::=	*return_type* `class_name::function_name(`*param_list*`)`
		| `% class_name::function_name(`*param_list*`)`
		| `% %::function_name(`*param_list*`)`
param_list	::=	`[ param_type1, param_type2, ... | % ]`
kernel_expression	::=	*phase_expression* | *event_expression*
phase_expression	::=	**kernel_phase** == [`MON_INIT_PHASE_BEGIN`
		| `MON_INIT_PHASE_END`
		| `MON_INIT_UPDATE_PHASE_BEGIN`
		| `MON_INIT_UPDATE_PHASE_END`
		| ...
		| `MON_METHOD_SUSPEND`
		| `MON_THREAD_SUSPEND`]
event_expression	::=	*event_name*.**notified**

Table 3.1 : Proposed specification primitives

3.8.1 Kernel-level primitives

Kernel phases

A *kernel_expression* is an expression about the state of the kernel. This work introduces primitives for exposing the current phase (*phase_expression*) and when events are notified (*event_expression*).

When the kernel has the thread of control, the execution trace makes transitions that reflect the changing phases of the kernel. A primitive **kernel_phase** is added that exposes the current phase. The primitive returns a value in the set of sample points presented in Table 3.2 corresponding to our abstraction of the kernel in Figure 3.2. The primitive **kernel_phase** allows the user to define properties whose evaluation is triggered by different phases of the kernel.

Example 1 (Stable states) *Variable p in module* `mod` *must be 0 in all stable states*

MON_INIT_PHASE_BEGIN
MON_INIT_PHASE_END
MON_INIT_UPDATE_PHASE_BEGIN
MON_INIT_UPDATE_PHASE_END
MON_INIT_DELTA_NOTIFY_PHASE_BEGIN
MON_INIT_DELTA_NOTIFY_PHASE_END
MON_DELTA_CYCLE_BEGIN
MON_EVALUATION_PHASE_BEGIN
MON_EVALUATION_PHASE_END
MON_UPDATE_PHASE_BEGIN
MON_UPDATE_PHASE_END
MON_DELTA_NOTIFY_PHASE_BEGIN
MON_DELTA_NOTIFY_PHASE_END
MON_DELTA_CYCLE_END
MON_TIMED_NOTIFY_PHASE_BEGIN
MON_TIMED_NOTIFY_PHASE_END
MON_METHOD_SUSPEND
MON_THREAD_SUSPEND

Table 3.2 : Sample points corresponding to the kernel phases according to the abstraction presented in Figure 3.2

(i.e., states when no process is executing):

```
ALWAYS ((mod::p == 0) @ ((kernel_phase == MON_METHOD_SUSPEND) ||
(kernel_phase = MON_THREAD_SUSPEND))).
```

Example 2 (Stable states – coarser sampling rate) *Variable p in module* mod *must be 0 at the end of delta cycles*

```
ALWAYS ((mod::p == 0) @ (kernel_phase == MON_DELTA_CYCLE_END)).
```

Event notifications

During the execution of the user code a process may request an event to be *notified.* Event notifications (*event_expression*) allow the specification to refer to the instance when the notifications actually takes place. Note that the mechanisms described earlier expose function calls at the source-code level, and event notification requests and cancellations (i.e., calls to **notify**() and **cancel**()) are exposed via the user-code primitives. However, these primitives do not expose the particular state when the ac-

tual notification is carried out (i.e., when the dependent processes are made *runnable* by the kernel). For each event we propose a primitive **notified** which is true whenever the kernel carries out the actual notification. For immediate notifications this happens concurrently with the function call to **`notify()`**; for delta-delayed notifications it happens during the earliest delta-notification phase; for time-delayed notifications it happens during the corresponding timed-notification phase. Note that both delta-delayed and time-delayed notification requests can be subsequently canceled, therefore a call to **`notify()`** with a non-negative argument does not guarantee that **notified** will be true in the future. The role of this primitive is particularly important when referring to events that are notified implicitly, e.g. when an **`sc_signal`** changes value, a built-in event returned by the function call `value_changed_event()` is notified by the kernel in the delta notification phase that immediately follows.

Example 3 *The requirement that a signal changes in every delta cycle can be expressed as*

```
ALWAYS ((kernel_phase == MON_DELTA_NOTIFY_PHASE_BEGIN) ->
( (!(kernel_phase == MON_DELTA_NOTIFY_PHASE_END)) UNTIL
                                signal_name.value_changed_event.notified)).
```

Example 4 *Variable p in module* `mod` *must be 0 at the rising edge of clock* `cl`*:*

```
ALWAYS (cl.posedge.notified -> mod::p == 0).
```

3.8.2 User-code primitives

A *usercode_expression* is a specification primitive about the state of the user model. Under this category are included Boolean atomic propositions about the location counter (*location_proposition*): execution of a specific statement, call and return of functions, and start and end of execution of functions. Also included are non-Boolean primitives referring to the arguments and return values of functions (*argument_primitive*).

The definition of a trace explicitly keeps track of the location counter during the execution of the user model. With each location in the source code of the model we associate a Boolean variable that is true precisely in those states where the statement that is about to be executed corresponds to the *location_proposition*. Two optional modifiers, **before** and **after**, allow the specification to refer, respectively, to the state immediately before and immediately after that statement is executed (the default behavior corresponds to specifying **before**). Using this primitive allows the specification of forbidden or mandatory paths in the execution of the compiled model, e.g., if execution reaches `location1` it should not reach `location2`. It also allows the specification of properties that must hold at particular locations in the code.

Identifying functions

Some of the primitives proposed in this work refer to function calls, return values, and values of function parameters. The specification identifies the required functions by means of *function_identifier*s, adopted from the framework of aspect-oriented programming (discussed in Chapter 5). A *function_identifier* may list the return type of the function (or "**void**" for functions without a return type); the specification can also use "`%`" to match any return type. Thus, `% foo()` refers to all functions named `foo()`, while **int** `foo()` refers to the subset containing all functions named `foo()` that return **int**.

A *function_identifier* may also contain the module or class name where the function is defined, or use "`%`" to indicate that the function may be defined in any class. For example, `% bar::foo()` matches all functions named `foo()` in module or class `bar`, and `% %::foo()` indicates that the function `foo()` may be defined in any class. Global functions are identified by omitting the class, e.g., **int** `foo()`.

Functions can be distinguished further by specifying the list of parameter types (*param_list)* within the *function_identifier*, or using "`%`" to match any list of parameter types. For example, `% foo(int)` and `% foo(%)` both match the function

void `foo(int)`, while the latter also matches the functions **void** `foo(char)` and **void** `foo(float)`.

Syntax matching atomic propositions

Inspired by BLAST [BCH+04], we also propose adding a primitive **curr_statement** of type `string` that exposes the syntax of the statement that is about to be executed. As mentioned earlier in Section 3.5, this mechanism enables properties that relate syntax and semantics. PSL and SVA allow using functions defined in the underlying HDL language, which in the context of SystemC means that we can use a number of C++ functions that operate on strings and return Booleans (*syntax_expression*) and pass **curr_statement** as an argument. Of particular interest are regular expression matching and string comparison functions, because they allow the user to quickly identify a set of "important" locations in the source code without having to introduce labels manually. As an example, one can use this mechanism to identify all locations in the user model where a particular statement is executed.

Example 5 (Matching source code via regular expression) *Variable pointer* `a` *in module* `mod` *should not be NULL before dereferencing it:*

```
ALWAYS (mod::a = NULL) @ (''\*a'' || ''a->'')
```

Atomic propositions exposing start and end of function execution

The specification of pre- and post-conditions requires evaluating assertions at specific locations in the source code that are difficult to identify automatically via the mechanisms described so far. Inspired by SLIC [BR02], we introduce two additional primitives, **entry** and **exit**, that refer to the location immediately before the first executable statement, and the location immediately after the last executable statement, in a function. In some cases the pre-condition may need to refer to the values of the formal parameters passed on to the function. If the function is a part of the user model, one can use the names of the variables on the parameter list. We also

propose an alternative mechanism (previously used in both BLAST [BCH+04] and SLIC [BR02]) to refer to the value of each parameter according to its order. For a function *func(type1 param1, type2 param2, ...)*, we define implicit variables *func:1*, *func:2*, etc., whose values (and types) are equal to the values (and types) of the formal parameters of the function at the entry point (i.e., the values of the variables before the first statement in the function has been executed).

Example 6 (Precondition) *One desirable precondition for a function* `float long_division(double` dividend, `double` divisor`)` *in module* `FPU` *is that the second parameter should not be 0 at the time when the function starts execution:*

```
ALWAYS (``float FPU::long_division(%)'':2 != 0) @
                                (``float FPU::long_division(%)'':entry).
```

Atomic propositions exposing call and return of functions

This mechanism is inadequate for the specification of pre- and post-conditions of functions defined in a proprietary library because the source code is not exposed in the execution trace. For cases like this we introduce another set of primitives that we adopt from SLIC [BR02]. For each function call to *function_identifier* we introduce the primitives *function_identifier* : **call** and *function_identifier* : **return** to refer, respectively, to the location in the source code that contains the function call and to the location immediately after the function call. (Note that here we assume that function calls are not nested.) The values of the arguments can be accessed via implicit variables *function_identifier* : **1**, *function_identifier* : **2**, etc., whose types match the types of the arguments to the function, and whose values are precisely the values of the actual parameters at *function_identifier* : **call**. Another implicit variable, *function_identifier* : **0**, is defined as the value returned by the function, and it is only defined at *function_identifier* : **return**. This mechanism allows the specification of properties of proprietary functions and objects even if the library does

not expose their states directly (e.g., a proprietary channel). For example, we can ensure that a channel contains only positive values by specifying that the arguments to all relevant calls to `write()` are always positive. As a second example, we can express the property that the channel behaves like a queue by using PSL's modeling layer to temporarily remember two values written to the channel, and then verifying that the values are returned in the same order via the channel's `read()` method.

3.9 Using primitives as clock expressions

Notice that the clock-sampling mechanism available in PSL and SVA offers a way to express temporal properties at different levels of abstractions. A fine sampling would correspond, say, to subcycle-level abstraction, while coarser sampling would correspond, say, to transaction-level abstraction. Since in PSL and SVA any Boolean expression can be used as a clock expression, the additions to the Boolean layer proposed here also provide a much finer control over the temporal resolution. We show that this approach enables us to tailor temporal languages to SystemC in a uniform way; all that is needed is to adapt the underlying syntax for state assertions. The main feature of the resulting framework is the ease with which properties can be expressed at different levels of abstractions, without having to use different languages.

Traditionally (e.g., [EF06,VR05,KT07]) sampling is done at the boundary of clock cycles. The framework presented here can easily provide the same functionality by using the event notification primitive described earlier. Note that an **`sc_clock`** exposes two events, `posedge_event` and `negedge_event`, which are notified every time the value of the clock changes and the new value is, respectively, 1 and 0. Using `posedge_event.`**`notified`** and/or `negedge_event.`**`notified`** as clock expressions we can sample at the boundaries of half-clock or clock cycles. Clearly, the user is not limited to the simulation clock. If finer grained resolution is required, one can sample at the boundary of delta cycles by using (**`kernel_phase`** `== MON_DELTA_CYCLE_END`) as a clock expression (sampling at the end of delta cy-

cles), or at the end of execution of each process by sampling at `( (kernel_phase == MON_THREAD_SUSPEND) || (kernel_phase == MON_THREAD_SUSPEND) )`, which corresponds to the phases where SystemC threads and methods suspend execution. One can even sample at the boundary of the individual statements in the source code (which is the default sampling rate).

Example 7 (Clock expressions) *For this example we borrow some of PSL's syntax. The property that every call to function* `req()` *is followed within 3 clock cycles (of clock* `cl`*) by a notification of event* `ack` *can be expressed as*

```
default clock = cl.posedge.notified;
ALWAYS (req():call -> next[3] ack.notified).
```

Example 8 (Clock expressions – coarser temporal resolution) *If the acknowledgment needs to be received within 3 delta cycles instead, all we need to do is change the clock expression:*

```
default clock = (kernel_phase == MON_DELTA_CYCLE_END);
ALWAYS (req():call -> next[3] ack.notified).
```

The same mechanism allows specifying coarser sampling rates as well. In a transaction-level or system-level model one is typically interested in its behavior at event notification instances or at function calls. Jeda Technologies' framework provides this functionality in a separate language (TLA), but they require the user to setup callbacks (essentially, function calls) that "report an event occurrence at the point of transaction processing" [KT07]. In our framework this can be done by using as clock expressions the event-notification primitives introduced earlier. For example, in a transaction-level model we can sample at the instances when the an `sc_fifo` is written to (by sampling at `data_written_event.notified`), or when a signal changes value (by sampling at `value_changed_event().notified`), etc. The advantage of our framework is that it is using the same language throughout

the refinement process as the model is transformed from higher to lower levels of abstraction.

3.10 Summary and discussion

This chapter points out three major deficiencies in existing temporal languages for SystemC: 1) lack of definition of an execution trace, 2) lack of flexibility to handle modeling at different abstraction levels, and 3) failure to take advantage of well-known and widely used primitives for software specification. This chapter proposes a precise definition of SystemC traces, which captures the alternation between the user code and the kernel. It also defines a systematic way for enriching existing specification languages with a set of Boolean properties, which, together with existing clock-sampling mechanisms in PSL and SVA, allow the sampling of the execution trace with flexible temporal and transactional resolution. This framework enables the specification of properties at different levels of abstraction.

The notion of a state presented here encompasses information about the kernel (current phase and notification of events), as well as statement-level information about the user model, and publicly exposed state of the libraries. The level of details preserved in the states makes it possible to define a rich set of new properties about the execution of the SystemC model. Moreover, the user can specify a range of sampling rates, from the most coarse (transaction- and system-level) to the most detailed (statement level) by combining clock expressions with the primitives introduced in this chapter. The framework presented here is general enough that it can be adopted by most existing temporal specification languages by simply enriching the set of allowed atomic expressions.

Bringing techniques from software verification to the SystemC world is the second contribution of this chapter. The fact that SystemC models should be viewed as software models has been ignored so far. The result is a minimal yet highly expressive extension of PSL/SVA.

The framework proposed here is equally applicable to dynamic verification and formal verification. Enabling a dynamic verification path would require a minimal "one-time" addition to SystemC's simulation kernel source code to expose a part of SystemC kernel's internal state and data structures. The user code will have to be instrumented to allow the monitors to observe the behavior of the relevant components, and the monitors will be compiled and executed together with the model. These issues are addressed in Chapters 4 and 5.

Applying formal methods to SystemC is an active area of research with several different approaches (e.g. using communicating state machines [MMMC06, Moy05], Petri-nets [KEP06], or leveraging Promela/SPIN [TCMM07]). All of these works propose some FSM-like abstraction of the SystemC kernel, and no two abstractions are the same. The model presented in this paper corresponds directly to the simulation semantics as described in the SystemC LRM [IEE06] and is the most detailed model without making any assumptions about the particular kernel implementation. The FSM in Figure 3.2 can easily be adopted by existing and future formal verification approaches. Exposing the syntax further allows the analysis of the model from a purely software point of view. The techniques used in SLIC [BR02] and BLAST [BCH+04] can and should be applied to formal verification of SystemC.

Chapter 4

Monitoring Framework for SystemC

4.1 Introduction and motivation

4.1.1 Exposing the simulation semantics

The specification primitives proposed in Chapter 3 require that the simulation semantics of SystemC be exposed as a part of the system state. Specifically, they require exposing the phase of the simulation kernel and event notification. Moy et al. [MMMC06, Moy05] also proposed exposing information from the kernel, but their abstraction is motivated by the types of properties they want to check. This thesis argues that the abstraction should expose fully SystemC's simulation semantics, as described in [IEE06]. A coarse abstraction might hide details that may be of importance to some users. Thus, an abstraction at the level of the simulation semantics is as generic as possible, enabling further abstraction if required by specific applications.

In addition to proposing the exposure of the kernel phases, Chapter 3 proposes exposing the notification of SystemC events. Recall that event notifications can be requested in the user model by calling the **`notify()`** method of class **`sc_event`**. The actual moment when the event is notified is determined by the kernel depending on the type of each event notification, the status of the other processes in the model, and the kernel phase. There are three types of event notification:

1. **`notify()`** with no arguments: immediate notification. Notification happens upon execution.

2. **`notify`**`(SC_ZERO_TIME)` or **`notify`**`(0, SC_SEC)`: delta notification. Notification is postponed until the delta-notification phase.

3. **notify**(time) with a non-zero time argument: timed notification. Notification happens during a subsequent timed-notification phase.

Pending event notifications can be canceled using the **cancel**() method, pending timed notifications are canceled by delta notifications, and pending delta notifications are canceled by immediate notifications.

Below we present two SystemC models and some properties that cannot be expressed (and monitored) without reference to kernel phases and events. The same two models and the temporal properties described here are also used to evaluate empirically the performance of our proof-of-concept implementation.

4.1.2 A model implementing squaring via addition

The first SystemC model implements a squaring function by using repeated incrementing by 1. The system consists of an Adder module that implements addition $a + b$ by triggering b copies of `add_1()` process, each of which adds 1 to a.

We use a delta-delayed notification of a `driver_event` to suspend the driver and allow the `add_1()` processes to initialize. Then the driver uses immediate notification of an `add1_activate_event` to activate the `add_1()` processes, which then proceed to execute sequentially within the same delta cycle. At the end of execution, each `add_1` process (immediate–) notifies an `addition_event`. Thus, the result $a+b$ is calculated within two delta cycles and using b (immediate) notifications of the `addition_event`.

The Adder is embedded inside a Squarer module, which implements c^2 by repeatedly calculating the sum of c and `running_total`. The squarer waits until the next clock cycle before calculating the next addition. It takes c clock cycles to complete the calculation.

This simple model is intentionally inefficient. Notice that it is driven by all three types of event notifications (immediate, delta-delayed, timed). It also allows us to vary the size of the model by varying the number of processes. A small piece of code

illustrating the functionality of the Adder is presented below in Listing 4.1; the full source code is available in Appendix A.

```
1  void adder::driver() {
2    while(true) {
3      //Suspending until values on the inputs change
4      wait(input1.value_changed_event() |
5                        input2.value_changed_event());
6      i1 = input1.read();
7      i2 = input2.read();
8      _a = i1;
9
10     for (int i=0; i < i2; i++) {
11       sc_spawn( sc_bind(&adder::do_add1, this) );
12     }
13
14     // Allow the do_add1() processes to initialize
15     // by suspending until the next delta-cycle
16     driver_event.notify(SC_ZERO_TIME);
17     wait(driver_event);
18     add1_activate_event.notify();  // immediate notification
19
20     // Suspend for a delta-cycle to allow all
21     // computations to complete
22     driver_event.notify(SC_ZERO_TIME);
23     wait(driver_event);
24     result.write(_a);
25   }
26 }
27
28 void adder::do_add1() {
29   wait(add1_activate_event);
30   (_a) = (_a) + 1;
31   addition_event.notify(); // immediate notification
32 }
```

Listing 4.1: A code snippet from the Adder model

A correct implementation of the Adder must satisfy the following property:

```
ALWAYS (adder.add1_activate_event.notified && adder._a > 0) ->
((adder.addition_event.notified -> (adder._a > 0))
UNTIL ``result.write(_a)'')
```
(1)

i.e., if $a > 0$ at `add1_activate_event`, then $a > 0$ at every instance when the `addition_event` is notified until the result is pushed to the output wire. Sampling at such low temporal resolution is not possible under the current SystemC standard. The best we can do is to check the value of a before the `driver_event` is notified, and to check it again when the result is being written to the wire. There is no mechanism to check an assertion at particular event notifications, so the intermediate steps cannot be verified.

Another property of the Adder is

```
default clock = (kernel_phase == MON_DELTA_CYCLE_END);
ALWAYS ((``i1 = input1.read();'':after) ->
(within [2] ( (``result.write(_a);'') &&
adder::_a == adder::i1 + adder::i2)))                    (2)
```

i.e., the correct result is always returned within 2 delta cycles of receiving the inputs (a and b). This property also cannot be monitored using the current SystemC standard. A monitoring process can count the delta cycles in which it is triggered, but there might be delta cycles when the monitor is not triggered, and the monitor would not be able to count those. Thus, there is no way of determining that precisely two delta cycles have passed.

These limitations stem from the way the SystemC kernel is designed. The kernel makes the *effect* of event notifications visible only to the processes waiting for those events. While this is sufficient for simulation purposes, it makes monitoring some important properties impossible. What is missing is a mechanism for alerting monitors immediately after an event and for alerting monitors that a delta cycle is about to start or end. However, event notification (and delta cycle determination) is done by the kernel and is not exposed to the rest of the system. The solution to this issue is to expose some of the internal state of the kernel for *monitor-only* privileged access.

4.1.3 A model implementing an airline reservation system

The second SystemC model implements a system for reserving and purchasing airplane tickets. The users of the system submit requests by specifying the starting and the ending airports of the trip, the dates of travel, and a few other pieces of data. The system uses a randomly generated flight database to find a direct flight or a sequence of up to three connecting flights. Those are returned back to the user for approval. If the user would like to purchase a ticket, she submits payment information that is processed and stored, and the individual legs of the trip are booked.

Internally the system uses several modules connected by finite-capacity channels. Each module has a fixed amount of local memory, implemented as bounded queues, to store the requests that are currently pending processing or are waiting to be sent to another module. The modules use events to synchronize reading and writing to the internal memory. All modules except the I/O modules are connected to the same (slow) clock, and the I/O modules are connected to another (faster) clock. This allows stress-testing the behavior of the system when there are more requests than it can process. This model is intended to run forever. It approximates actual subsystems currently used in hardware design. A piece of code from the `flight_planner` module is presented below.

```
1  /**
2   * Receives requests from the master module. Adds
3   * new requests to the new_planning_requests queue
4   * if there is space available, otherwise blocks
5   * until space becomes available.
6   */
7  void flight_planner::receive_new_requests() {
8    while(true) {
9      tlm_tag_t t;
10     if (! in_from_master->nb_can_get( &t )) {
11       tlm_tag_t tag;
12       wait(in_from_master->ok_to_get(&tag));
13     }
14     token_t* req = new token_t();
15     in_from_master->nb_get(req);
```

```
16      if (req->get_payload() == PLAN) {
17        unsigned int curr_size =
18                          new_planning_requests.size();
19        if (curr_size >= queue_size) {
20          wait(new_requests_nonfull);
21        }
22        new_planning_requests.push(req->get_request());
23        new_requests_nonempty.notify(SC_ZERO_TIME);
24        wait();
25      } // it was a new request
26      else {
27        handle_special_request(req->get_request(),
28                                  req->get_payload());
29      }
30    } // receiving loop
31  } //receive_new_requests()
```

Listing 4.2: A code snippet from the Airline Reservation System model

One safety property of the system is that whenever some process notifies the event `new_requests_nonfull`, the corresponding queue (`new_planning_requests`) must have capacity for storing at least one request. Formally,

```
ALWAYS (new_requests_nonfull.notified ->
(new_planning_requests.size() < capacity))                    (3)
```

Notice that placing this assertion at the locations where the event notification is requested may lead to false negatives. Even if the assertion fails in that location, a subsequent process may *cancel* the event notification and the property would still be satisfied. This example demonstrates further the need for sampling at event notifications.

In order to meet performance objectives, the system must propagate each request through each channel (or through each module) within 5 cycles of the slow clock. This property is a conjunction of 16 bounded liveness assertions similar to the one shown below.

```
// Modeling layer:  keep track of requests as they come in
std::map< int, sc_time > requesttime;
token_t* t;
```

```
int request_id;
default_clock = slow_clock.posedge.notified;

// Propagate through io_module within 5 clock ticks
ALWAYS ((``status_t io_module::receive_transaction(%)'':entry) ->
(({t = ``status_t io_module::receive_transaction(%)'':1;
request_id = t->get_request()->get_request_id();
requesttime[ request_id ]= sc_time_stamp();}) &&
( within [5] (``% io_module::send_to_master(%)'':entry &&
({t =``% io_module::send_to_master(%)'':1;
request_id = t->get_request()->get_request_id();
requesttime[ request_id ] - sc_time_stamp() <=
                    5 * slow_clock.get_period();}))))
) AND ...                                                    (4)
```

Monitoring Property (4) requires a process that is aware of the slow clock and can be triggered from multiple processes in a non-deterministic sequence. Implementing a monitor of this type using the existing SystemC kernel would require major instrumentation of the model in order to store and propagate the required information. A more scalable and easier to use approach is to allow the creation of monitors that are accessible by all processes and at the same time have access to the kernel's internal information. The framework presented here solves these and many other problems to allow monitoring of important and previously untestable properties of SystemC models.

4.2 Related work

Several groups have proposed modifying the standard SystemC kernel in order to expose race conditions that may occur under alternative schedulings.

Helmstetter et al. [HMMCM06] apply dynamic partial-order reduction techniques

to explore alternative schedulings during simulation. They distinguish communication operations on events (**wait()** and **notify()**) and variables (`read()` and `write()`), and use them to define partial ordering of process execution. Alternative schedulings are produced by permuting the execution of processes subject to the partial ordering. They modify the standard SystemC scheduler by replace the process election code with an interactive version, and adding code to keep track of communication actions [HMMCM06].

Blanc and Kroening [BK10] point out that the SystemC standard allows implementations of the kernel to adopt a deterministic scheduling policy. Consequently, a model with an inherent data race may not exhibit it even after multiple simulations. The combinatorial explosion in the number of interleavings between processes makes it intractable to check all possible schedules. Instead, [BK10] employ formal methods to statically pre-compute dependencies between **SC_THREAD**s and **SC_METHOD**s and use those dependencies to prune the exploration of concurrent behaviors. Then they generate a static scheduler that replaces the dynamic scheduler in the reference implementation. The new scheduler leverages partial-order reduction and explores the remaining possible interleavings exhaustively [BK10].

Sen et al. [SOA08] propose a technique they call "predictive runtime verification" that preserves concurrency information about the execution and exploits it to find both actual and potential errors in the execution. Like [BK10], Set et al. exploit a partial-order execution trace instead of the total order execution trace produced by the reference scheduler. [SOA08] augment the scheduler with a new object that keeps track of event notifications and derives process dependencies as the model is executed. The resulting partial-order trace is then passed to an external verification tool, which determines whether the assertion holds or not [SOA08].

Braun et al. [BGR02] evaluate different strategies for checking temporal specification properties in a SystemC model. They consider two fundamentally different approaches: 1) an add-on library (a collection of SystemC objects) that implements

functions for checking temporal properties, and 2) an interface module that connects the SystemC model with an external test-bench environment (in particular, TestBuilder). The properties are limited to Finite LTL properties and the temporal resolution is fixed to the resolution of the simulation clock.

A number of proprietary specification languages for SystemC come with a monitoring framework. One of the more serious industrial efforts is by Kasuya and Tesfaye (Jeda Technologies) [KT07]. This work provides a set of primitives to express cycle-accurate and TLM-based temporal primitives, but no mechanism for adapting to different levels of abstraction.

4.3 Modifications of the kernel

Our first goal is to introduce minimal changes to the reference implementation in order to expose the actions of the kernel in a systematic way, while the behavior of the kernel (and thus the simulation semantics) remain unchanged. Only those steps described by the SystemC standard [IEE06] are exposed. Any implementation that follows the standard can be modified in a similar way.

One way to expose the state of the kernel is to implement an API that returns the current phase of execution of the kernel and relevant data, and another way is to modify the kernel to send updates about its execution. Notice that in the first case it is not clear how the monitors will be alerted when the kernel reaches a particular sample point. Busy waiting of the monitor will not allow other code (including kernel code) to execute, and using multi-threading inside the simulation does not guarantee that the monitor's thread (OS-level, as opposed to SystemC-level) will be active while the kernel is in a particular phase. On the other hand, if the kernel sends updates (via function calls), the monitor will be triggered and will execute as soon as the relevant sample point is reached. This is the mechanism that we chose to implement.

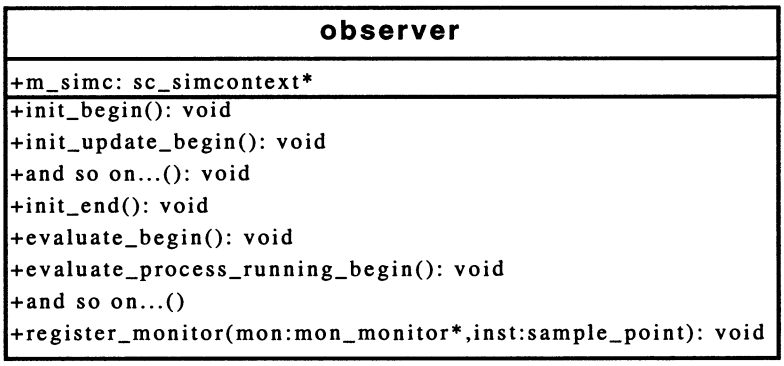


Figure 4.1 : Partial class diagram for `observer`

4.3.1 Determining when monitors are activated

One immediate problem is that this approach requires the kernel to have access to all monitors. While it is conceivable to add the necessary data structures to the existing code, it would require extensive modifications. Our intention is to add as few new lines of code as possible so that our framework can be applied to a wide range of SystemC implementations. To that end, we encapsulate all additional functionality in a new object, `observer`, and connect the existing code to it via callbacks. `observer` stores references to the monitors, receives updates of the kernel state, and then notifies the monitors that need to execute at the current sample point (Figure 4.1). The observer implements a callback for each phase of execution of the kernel. The code in Listing 4.3 below illustrates the main idea.

```
1  enum sample_point {
2    ...
3    MON_DELTA_CYCLE_END,
4    ...
5  };
6
7  class mon_observer {
8   public:
9
10   void delta_cycle_end() {
11     unsigned int num_elements =
```

```
12                    arr_mon_sets[MON_DELTA_CYCLE_END]->size();
13      if (num_elements > 0) {
14        monitor_set::const_iterator it;
15        for (it = arr_mon_sets[MON_DELTA_CYCLE_END]->begin();
16             it != arr_mon_sets[MON_DELTA_CYCLE_END]->end();
17             it++) {
18          mon_prototype* mp = *it;
19          mp->callback_delta_cycle_end();
20        }
21      }
22    }
23
24    void register_monitor(mon_prototype* mon, sc_event* eve) {
25      if (events_to_monitor_sets[eve] == 0) {
26        std::set<mon_prototype*>* n =
27                        new std::set<mon_prototype*>();
28        eve->register_observer(this);
29        n->insert(mon);
30        events_to_monitor_sets[eve] = n;
31      }
32      else {
33        (events_to_monitor_sets[eve])->insert(mon);
34      }
35    }
36
37    void event_notified(sc_event* event) {
38      monitor_set::const_iterator it;
39      for (it = events_to_monitor_sets[event]->begin();
40           it != events_to_monitor_sets[event]->end();
41           it++) {
42        mon_prototype* mp = *it;
43        mp->callback_event_notified(event);
44      }
45    }
46  } // class observer
```

Listing 4.3: Partial listing of the source code of the Observer object

The kernel source code is then modified to call the `observer` callback functions at the locations where a change of phase occurs. (In the OSCI reference implementation, the particular file that is modified is `sc_core::sc_simcontext.cpp`.) For example, Listing 4.4 shows a snippet of actual code (from `sc_simcontext.cpp`) with our modification:

```
1  while ( true ) {
2    // EVALUATE PHASE
3    m_execution_phase = phase_evaluate;
4
5    // One new line of code added below
6    if (observer != 0) { observer->evaluate_begin(); }
7
8    while( true ) {
9      // execute method processes
10     sc_method_handle method_h = pop_runnable_method();
11     ...
```

Listing 4.4: A code snippet from `sc_core::sc_simcontext.cpp`

4.3.2 Handling communication with monitors

The communication between the observer and the monitors is also via callbacks. However, that also raises another programming issue. The observer needs to be designed to communicate with objects that it knows nothing about at the time when the observer is compiled. To resolve that issue we define an abstract class, `mon_prototype`, that serves as a base class for all monitors (Figure 4.2). This class declares a **`virtual`** callback function for each type of sample point on the execution trace, for example, **`virtual`** `callback_init_phase_begin()` and **`virtual`** `callback_evaluate_phase_begin()`. Each monitor implements the callback functions that are relevant to its execution and that implementation is executed instead of the empty `virtual` implementation defined in `mon_prototype`.

Monitors request to be notified by issuing a call to the `observer`'s `register_monitor()` function. For example, a monitor for the Adder might use `register_monitor(`**`this`**`, MON_DELTA_CYCLE_BEGIN)` in order to be alerted at the start of each delta cycle. For each kernel phase, `observer` maintains a list of monitors that have requested to be alerted when the kernel reaches that particular phase. As soon as the kernel notifies (via a function call) the `observer` that the kernel is entering another phase, the `observer` calls the callback function corresponding to this kernel phase for each monitor that has requested to be notified.

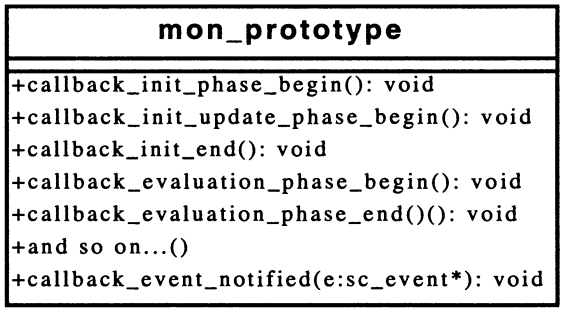


Figure 4.2 : Partial class diagram for `mon_prototype`

Monitors register with specific events directly and are alerted only when those events are notified. As an example, a monitor for the Adder would use `register_monitor(this, addition_event)` to request to be alerted as soon as the kernel notifies the `addition_event`. The communication mechanism is the same as the communication mechanism for kernel-level sample points, with the only difference that communication is initiated from the `sc_event` object. Implementing this idea requires minimal changes to the code of `sc_core::sc_event.cpp`.

Notice that the changes to the kernel are intended to be compiled once, together with the rest of the SystemC code, into a static library (for example, `libsystemc.a` in the case of the OSCI reference implementation) and linked with the user code. It is possible for a commercial implementation to adopt all of the changes proposed here and provide the simulator in binary without revealing proprietary source code.

The `observer` is instantiated from the user code at the end of elaboration in `sc_main()` and the `observer` instantiates all monitors before the simulation starts. This allows `observer` to pass references to user-code modules to monitors that monitor user-code properties, for example, values of variables. It is not until the end of elaboration that these references become valid, so instantiating the monitors earlier is not possible.

In case the model does not contain any properties to be monitored, there is negligible overhead in the modified kernel. If the user does not instantiate `observer`, the

kernel's pointer to `observer` defaults to 0. Before issuing any callback, the kernel checks if the `observer` is non-zero, and only then it issues the callback, for example,

```
if (observer != 0) { observer->update_begin(); }
```

Thus, in a simulation without monitors the kernel's overhead consists of checking a conditional at every sample point and event notification.

One may object to our decision to modify the kernel by arguing that there are several implementations of the kernel. Our response is that the language proposed in Chapter 3, which enables the expression of rich temporal properties, requires some kernel-level information to be exposed. Our modifications, however, only expose details that are described in the LRM [IEE06] and should be portable to any implementation that follows the standard. Furthermore, our changes of the existing code (of the OSCI implementation) are minimal and localized, and we believe that other implementations would be easily modified.

We want to note that there are many alternative ways of modifying the kernel (see, e.g. [BK10, SOA08, HMMCM06]) but none of the previous works has achieved the temporal resolution and kernel–monitor communication provided by our modifications. Other reasonable approaches for monitoring temporal SystemC properties have been explored by Broun et al. [BGR02], one of which requires no modifications of the kernel at all (at the cost of 4 times slower execution speed than a comparable approach with a modified kernel). The novelty of our approach is that it introduces a generic monitor object that can be refined to check any safety LTL property [AKT+06] and can sample at much finer (e.g., at the boundaries of delta cycles), as well as much coarser temporal resolution (e.g., at the boundaries of timeless transactions), than any existing approach.

In future releases of SystemC, the simulation semantics and the kernel may change, for example, adding new simulation phases. The modular framework that we describe in this work can be modified easily to handle changes in the kernel. For each new phase in the kernel, the observer will need to be extended to handle one additional

callback, and the modified kernel will need to be instrumented with one additional line of code. Similarly, removing a phase from the kernel requires deleting a callback from the observer, and deleting (rather, not adding) a line of code in the instrumented kernel.

4.4 Instrumentation of the MUV

Each monitor inherits from the base class `mon_prototype` the declarations of all virtual functions, and implements only those callbacks that are relevant to the property that is monitored. Each monitor then *registers* itself with the `observer` indicating which sampling points it is interested in. For example, if the property is

```
ALWAYS (p==0) @ (kernel_phase == MON_DELTA_CYCLE_BEGIN)
```

(i.e., `p==0` must hold at the beginning of each delta cycle) the corresponding monitor overwrites the function `callback_delta_cycle_begin()` to check if p is equal to 0. The monitor then registers itself with the `observer` using the following function call:

```
observer->register_property(this, MON_DELTA_CYCLE_BEGIN);
```

for the sampling point `MON_DELTA_CYCLE_BEGIN`.

In some cases properties refer to variables that are either **`private`** or **`protected`**. We adopt the perspective of "white-box validation", which means that the state of the model should be fully exposed to the monitors. This is consistent with the view in Chapter 3, which considers all class members to have public access for verification purposes. We use the C++ **`friend`** class declaration to give the monitors access to the data members of the monitored modules, while still preserving the encapsulation and the limited access that the designer intended.

Monitoring user-code variables or other user-code data structures is done at the boundaries of statements. This brings up the question of how modules can receive references to the monitors. The difficulty stems from the fact that the monitors are instantiated after the user-code modules, so they cannot be passed to the modules as

arguments. Our solution is to have the `observer` maintain a list of monitors that need to be triggered at user-code sample points. The `observer` does not handle the communication with those monitors. Instead, it defines a `mon_prototype* get_monitor_by_index(int)` function that returns a particular monitor by its index in the list. The instrumentation in the MUV then handles the communication with the monitor by issuing pre-defined callbacks. Each callback depends on the variable or data structure being monitored, and must be defined in the monitor.

For example, in the following code snippet, we are monitoring the execution of statements matching the regular expression `''*. = input?.read()''`:

```
1  ...
2  while(true) {
3    wait(input1.value_changed_event() |
4                        input2.value_changed_event());
5    int i1 = input1.read();
6
7    // Callback to notify monitor 42 that a statement matching
8    // the RE ''*. = input?.read()'' has been reached
9    observer->get_mon_by_index(42)->location_loc21();
10   // End instrumentation
11
12   int i2 = input2.read();
13
14   // Callback to notify monitor 42 that a statement matching
15   // the RE ''*. = input?.read()'' has been reached
16   observer->get_mon_by_index(42)->location_loc21();
17   // End instrumentation
18   ...
```

Listing 4.5: Illustrating statement-level monitoring of of the Adder model

The approach described here may seem cumbersome when the monitored properties are static assertions. Indeed, in those cases a simple `assert()` statement placed at the relevant location in the code will "monitor" the property more efficiently. Notice, however, that our framework is designed for monitoring temporal properties that `assert()`'s cannot handle. Moreover, with automated user code instrumentation

(see Chapter 5) and automated monitor generation (see Chapter 6) the framework can handle a large number of properties without significant manual effort from the user.

The framework presented here is backward-compatible with existing SystemC models, which can be linked against the modified `libsystemc.a` library without linking errors. In order to activate the monitoring framework the user model needs to instantiate an object derived from **`class`** `observer`, and instantiate one or more monitors that are derived from **`class`** `mon_prototype`.

4.5 Experimental results

We modified version 2.2.0 of the OSCI simulator and compiled it using the default settings in the Makefile. The empirical results below were measured on a Pentium 4 / 3.20GHz CPU / 1 GB RAM machine running GNU Linux.

4.5.1 Framework overhead

First we compiled each of the two models described below, without monitors and without an observer, using both the modified kernel and the original OSCI kernel. We ran a simulation of each version separately and measured a decrease of performance of less than 0.5% when using the modified kernel. We also compiled the models with an observer and no monitors (using the modified kernel) and measured an additional slow-down of the execution time of less than 0.25%. We did not observe any significant memory increase in either of those cases. Thus, our modifications of the kernel do not lead to a significant loss of performance compared to the unmodified kernel.

4.5.2 Monitoring overhead

For the rest of our experiments we measured the effect of running with a different number of monitors and monitoring the properties that we introduced in Section 4.1. Each data point represents the median of 10 measurements. In each case we first ran

the model without an observer, monitors, or user code instrumentation to establish the baseline, and then ran several simulations with instrumentation and an increasing number of copies of the same monitor. The results we report are the cost of each copy of the monitor as a percentage of the baseline.

When using the testing methodology described above it is important to consider the possibility of caching effects: if the system were reusing the results of previous computations, averaging the execution time would be meaningless. Our implementation avoids these issues because we create each copy of the monitor as a separate instance of the same class. Since we are not using **`static`** variables, each instance contains its own copy of the class data, thus memory use is proportional to the number of copies of the monitors. Furthermore, each instance is handled as a generic `mon_prototype` object, so it is impossible for the SystemC kernel to reuse computations. Finally, the behavior of each monitor is determined by the communication it receives from the SystemC kernel, and the C++ compiler cannot determine statically that each monitor is doing identical computations, therefore compile-time optimizations will not prevent each monitor from executing.

Squaring via addition

The first property checked was Property (1): the value of the Adder's a variable is always positive, monitored whenever `addition_event` is notified. Since the overhead of checking the property $a > 0$ is minimal, the results mostly expose the overhead of the monitoring framework. For comparison purposes, we measured separately the performance when sampling at the `addition_event` notifications, and when sampling at the end of each delta cycle. The following code snippet shows the key parts of the monitor that checks the safety property at `addition_event`:

```
1 /**
2  * ALWAYS (adder.add1_activate_event.notified && adder._a > 0) ->
3  * ((adder.addition_event.notified -> (adder._a > 0))
4  * UNTIL ''result.write(_a)'')
```

```
 */

// The constructor
mon1(observer* obs, adder* obj1) : mon_prototype() {
  observer = obs;
  object1 = obj1;
  // Register with the events
  object1->add1_activate_event.register_property(this);
  object1->addition_event.register_property(this);
}

// Overwrite the default virtual void function
virtual void callback_event_notified(sc_event* e) {
  if (e == &(object1->addition_event)) {
    adder_addition_event_notified = true;
    step();
    adder_addition_event_notified = false;
  }
  else if (e == &(object1->add1_activate_event_notified)) {
    add1_activate_event_notified = true;
    step();
    add1_activate_event_notified = false;
  }

  else {
    std::cout << ``Unknown event in monitor '' << std::endl;
    exit(1);
  }
} // callback_event_notified()


// State machine corresponding to the temporal property
void step() {
  current_state = next_state;
  next_state = -1;

  if (current_state == 0) {
    if (add1_activate_event_notified) {
      next_state = 1;
    }

    ...
  }
```

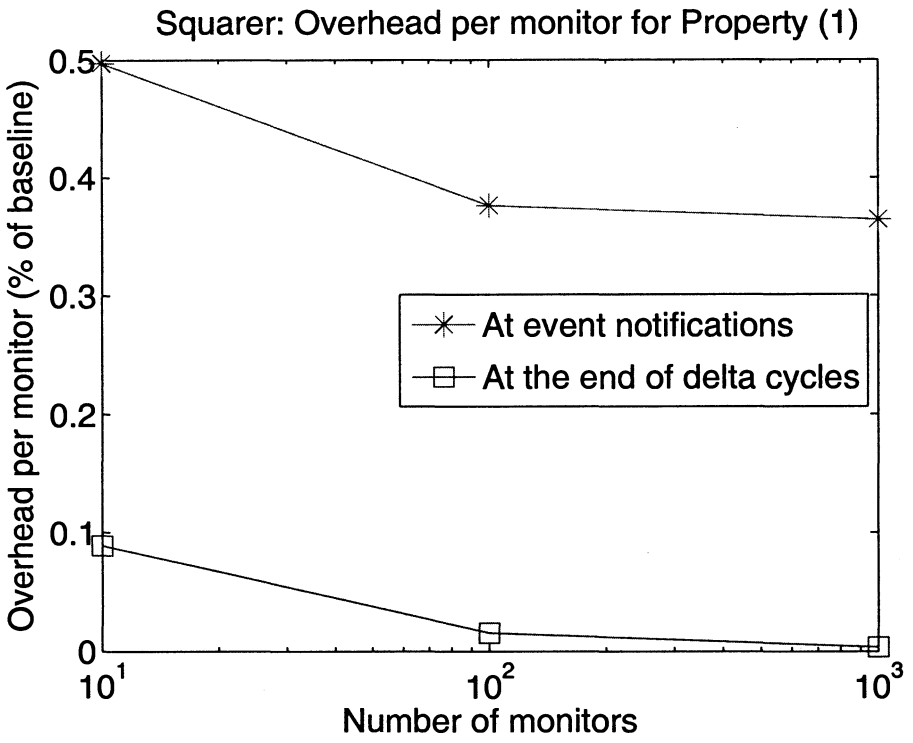


Figure 4.3 : Monitor overhead as a percentage of baseline (i.e., execution without monitors) for Property (1) using two different temporal resolutions

```
48 }
```

Listing 4.6: Partial implementation of a monitor for Property (1)

In each case we instantiated between 10 and 1000 copies of the monitor. The execution time to calculate 1000^2 without monitors is ~14 seconds. The results are reported as percentage overhead per monitor (Figure 4.3).

As the number of monitors increases, the overhead of the framework is amortized over more monitors, thus the average overhead per monitor decreases. Also notice that sampling at event notifications is much more expensive than sampling at the end of delta notifications. Each delta cycle involves (for 1000^2) 1000 event notifications, so the monitor is invoked 1000 times more often. We would also like to point out that while the average overhead per monitor is negligible (~0.5%–0.003%), the cumulative

effect of running 1000 monitors is significant. In the first case (sample at event notification) the execution is slowed down 363% when running with 1000 copies of the monitor (such overhead is not uncommon in industrial applications). The effect is less pronounced when sampling at the end of delta cycles, incurring a total performance penalty of 3.6% when running with 1000 monitors.

Property (2) asserts that the adder correctly calculates the sum and returns the result within two delta cycles of reading the input. Notice that this property combines information from the user code (getting the values of the relevant variables) and the kernel (getting information about each delta cycle). We evaluated this property for different sized models – 500, 1000, 1500, and 2000 processes, calculating, respectively, 500^2, 1000^2, 1500^2 and 2000^2. The results are in Figure 4.4. The behavior we observe is that increasing the number of processes in general reduces the overhead per monitor. The more processes we have, the more work the system needs to do in each delta cycle, thus the effect of the monitoring becomes a smaller fraction of the overall execution. The overhead per monitor averages around 0.01% and the worst cumulative slow-down we observed was by 12.9%, when using 1000 monitors on a 500-process model.

4.5.3 Airline reservation system

We checked two properties of the system:

1. The incoming queue has capacity for another request whenever the `incoming_req_nonfull` event is notified (Property (3)), and

2. Every request is propagated through a channel within 5 clock cycles of the slow clock, and it is sent out from each module within 5 slow clock cycles of its receipt by the module (Property (4)).

Since the system is designed to operate indefinitely, we measured the performance by simulating for 1 million cycles of the slow clock. The wall-clock execution time of the system without monitors is ~27 seconds. The monitoring overhead is presented

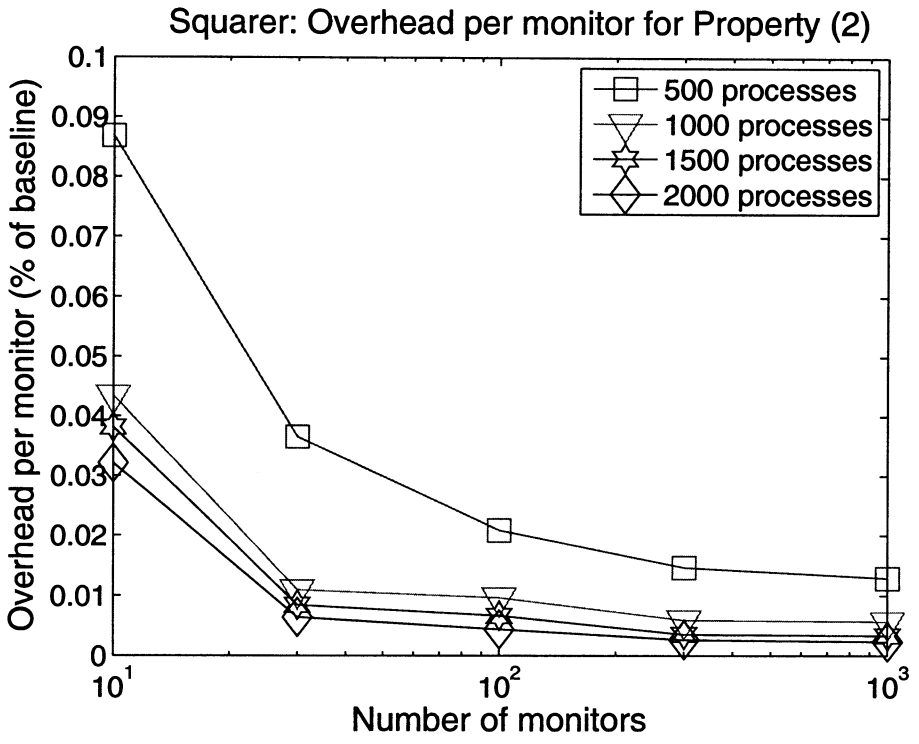


Figure 4.4 : Monitor overhead as a percentage of baseline for Property (2)

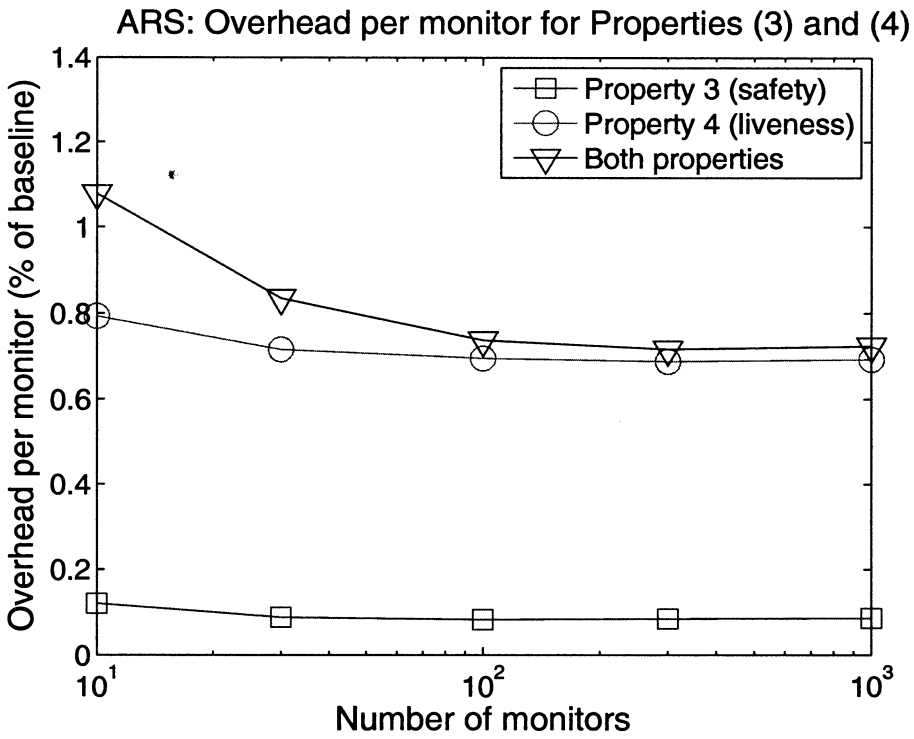


Figure 4.5 : Experimental results for monitoring Properties (3) and (4) in the airline reservation system

as percentage of that baseline (Figure 4.5). Checking Property (3) is relatively inexpensive, and the results are consistent with the previous results. Property (4) is much more expensive. The monitor contains a state machine that tracks the arrival and the departure of each request as it travels through the model. This requires a lot of communication from the model to the monitor, as well simulating a state transitions inside the monitor. Running the system with 1000 copies of both monitors slows it down by 715%. Although this is quite significant, it is not unusual. A ten-fold slowdown of the simulation when monitoring complicated properties is often observed in the industry.

4.6 Summary and discussion

This chapter introduces a monitoring framework for the specification language proposed in Chapter 3, that allows monitoring of temporal properties of SystemC at different levels of resolution, both in clocked and clockless models. The framework requires very small changes to the existing code of the kernel and encapsulates the new functionality in two new objects. A SystemC installation augmented with the new objects allows temporal properties to refer to event notifications and simulation phases. The changes to the kernel cause a negligible slow-down (less than 0.5%) of the simulation speed.

As proof of concept, several types of properties of two SystemC models, involving components from different modules, were implemented and tested at sub-clock-cycle and sub-delta-cycle resolution. The user code was instrumented to allow observation of the relevant parts of the model state. The experimental results show that for most properties the overhead is quite small and running hundreds or thousands of monitors does not have a prohibitive cost (usually less than 0.2% per monitor). More complex properties are naturally more expensive, but even in this case the overhead is typically less than 1% per monitor and further optimizations may improve the performance.

The primary of focus of this chapter is on creating an efficient monitoring framework, therefore the monitors and the user-code instrumentation was done by hand. Doing so for more complicated temporal properties would require substantial manual effort from the user. Thus, it is important to automate the process of instrumenting the user code to allow monitoring of statement-level primitives. Chapter 5 addresses this issue. Automated generation and integration of specification monitors with the framework presented here is the subject of Chapter 6.

Chapter 5

Aspects of Temporal Monitoring of SystemC

5.1 Introduction and motivation

The specification primitives defined in Chapter 3 propose enriching the Boolean layer of PSL and SVA, making specification languages more expressive. The primitives can be split roughly into two groups: primitives related to the execution of the kernel, and primitives related to the execution of the user code. The user code primitives that have to be exposed are meant to enable *white-box validation*, which means that the C++ code of all processes in the model, the values of user-define variables, location counter, and the call stack are first-class members of the property specification language [BCH+04]. This allows a very flexible temporal resolution of the *execution trace*. In particular, it enables specification of properties across a wide spectrum of temporal granularities, from cycle level to transaction level.

Chapter 4 shows how to expose the semantics of the SystemC kernel in a modular way without a serious performance hit. This chapter shows how to expose to the monitoring framework the execution flow, the C++ syntax and the state of the user code. The difficulty is that the very philosophy of object orientation is an obstacle to white-box validation. The benefits of object-oriented encapsulation and data hiding of modules, ports, and channels are deterrents for effective monitoring, because access to internal variables is limited to the objects' own processes. Good software-engineering practices suggest keeping internal data private and exposing it only via dedicated function calls or ports. Most other works (see discussion later) focus on using only the publicly available data members, exposing the execution of user code only at the level of function calls and returns, or requiring manual instrumentation. In contrast,

our work automate low-level monitoring of user-code primitives.

Exposing the state of the model and the flow of its execution is a cross-cutting aspect of the model's behavior, and some of the desired exposure can be achieved using the Aspect-Oriented Programming (AOP) paradigm [KIL+97] (see Subsection 5.2.1). Directly using AOP, however, requires users to manually generate fairly sophisticated AOP code. We get around that by adding a layer of abstraction above the level of AOP, so that verification engineers can use a simple declarative language to describe the desired primitives to be exposed. Our framework pre-processes user declarations and automatically generates AOP *advices*, which are then *woven* into the user code automatically using `AspectC++` [SGSP02]. We show how to expose the values of local module-level data members, even those marked `protected` or `private`, all invocations of a specific function, arguments passed to and returned from user-code functions, and statements being executed, without requiring additional annotations or instrumentation by the user or knowledge of AOP.

Some exposure, for example, exposing all locations in the code where a variable `a` is divided by or is incremented, cannot be achieved using AOP. We have identified some of those limitations and propose an alternative method, based on pattern matching, for identifying and instrumenting locations in the source code.

Detecting violations of the property requires constructing a deterministic monitor that uses the exposed values and locations. The user can generate such monitor by hand or using an automated tool. The monitor is integrated with the instrumented code via simple function callbacks implemented in the monitor. The techniques presented in Chapter 6 show how the monitor construction can be automated, but the framework presented here is applicable to any monitor construction.

5.2 Preliminaries

5.2.1 Aspect-Oriented Programming

Aspect-Oriented Programming [KIL+97] is based on the idea of separation of concerns at source code level. In many systems there are requirements that do not partition cleanly into objects, for example, synchronization, logging, and locking/unlocking. AOP allows such concerns (called *aspects*) to be programmed separately and then composed with the rest of the source code (by an *aspect weaver*) into a coherent program [EFB01].

The insertion points where the aspects are weaved are referred to as *join points*. A join point may refer to a function, an attribute, a type, or a variable. A *pointcut* is a set of join points, described by a *pointcut expression*. An *advice* declaration specifies the code that should run when the join points specified by a pointcut expression are reached. There are four types of advices: *before* advices, which are executed before the join point, *after* advices, executed after the join point, and *around* advices, which are executed in place of the join point. The fourth type of advice, *introduction*, extends classes with new functions or data members. All advices are declared inside an *aspect declaration*. For a detailed discussion of AOP and `AspectC++` we refer the reader to [SGSP02].

Two key features of AOP are 1) *quantification*: i.e., the same aspect code can have effect in many locations, and 2) *obliviousness*, or the idea that the places (join points) where the aspects have effect need not be specially prepared to receive these enhancements [TLSS10, EFB01]. These features make AOP particularly well suited for instrumenting code for monitoring, because they do not require additional efforts from the designer of the model.

5.2.2 Monitoring framework

In order to get references to specific monitors we leverage the monitoring framework presented in Chapter 4 for handling all monitors and for activating them at the necessary sample points. Each monitor that we construct in this chapter derives from `mon_prototype`, and in addition defines callbacks that are called from the instrumented code to communicate with the monitor. We also take advantage of the `monitor_observer` object. Recall that at instantiation, each monitor registers with the `mon_observer`, which builds a list of all monitors. `mon_observer` provides a function, `get_monitor_by_index()`, that returns a (generic) `mon_prototype*` pointer to any of the monitors. Our implementation uses this mechanisms to obtain pointers to monitors from the instrumented code and to call the appropriate callback from the instrumented code.

5.3 Related Work

AOP has been applied to instrumenting Java programs (see, e.g., [dH05, Bod05, HJ08]). The AOP weaver for Java, `AspectJ`, is very mature and widely used. A lot of ideas from Java-based AOP have been transferred to `C++`-based AOP, but adoption is still lagging.

Niemann and Haubelt [NH06] use AOP to expose function calls in the user code, and then check (off-line) the trace against the specification using SVA. They associate with each monitored function an atomic proposition that is true iff the function has been called but has not yet returned. Our solution exposes function calls and returns, which allows us to emulate the functionality of [NH06], but we also expose the function parameters and return value, and arbitrary code statements based on pattern matching. Unlike the off-line post-processing mechanism used by Niemann and Haubelt, we monitor the specification online. As soon as illegal behavior is detected the execution of the model can be terminated or reset to a known good state.

Endoh et al. [EIIK08] propose using AOP join points directly as Boolean atomic propositions. Intuitively, the atomic proposition associated with a join point holds iff the execution of the model reaches the join point. They do not expose internal variables of the model, nor do they expose parameters of functions or return values of functions. Their approach requires the user to generate the AOP code manually. We expose a richer set of primitives and we use a higher level of abstraction for declaring primitives, from which the AOP advice code is generated automatically by our pre-processor.

Déharbe and Medeiros [DM06] use `AspectC++` to instrument SystemC for metrics collection, injecting different algorithms into processes (e.g., substituting in different cache policies), fault injection, and hardware-level polymorphism. Kasuya et al. [KHT04] adopt AOP to the Jeda programming language to facilitate adding debugging messages and measuring code coverage. Tartler et al. [TLSS10] deal with instrumentation of a running program using the AOP paradigm (i.e., *dynamic weaving*). None of these works applies AOP to monitoring.

5.4 User-code primitives

Our approach provides a mechanism for referring to a rich set of user code primitives in property specifications, without requiring the user to instrument the code manually or to write AOP advices. Primitives are declared by the user via a high-level language, and after that they can be used in any of the properties. We use a configuration file to store all primitive declarations, properties, and optional parameters for our preprocessor and the monitor generator. A command-line interface allows the options specified in the configuration file to be overridden. The primitives that can be used are described below.

5.4.1 Exposing function calls

Certain assertions need to be checked immediately before a particular function call is made, or immediately after a particular function call returns. The declaration

```
location loc1 ``\% bar::foo()'':call
```

declares a Boolean atomic proposition `loc1` that holds immediately before the execution of the model reaches a call-site of a function `foo()` of class `bar`. Similarly, a Boolean `loc2` that holds immediately after the return of the function is declared using

```
location loc2 ``% bar::foo()'':return.
```

Example 9 *Suppose that we have a model consisting of two modules: producer and consumer, that are connected by a channel. The producer defines a blocking call* `send()` *to push tokens to the channel, and the consumer defines a non-blocking call* `receive_nb()` *to read from the channel. We want to specify that* `send()` *remains blocked until* `receive_nb()` *has returned. We declare the following primitives:*

```
location send_start ``% producer::send()'':call
location received ``% consumer::receive_nb()'':return
location send_done ``% producer::send()'':return
```

and use them to specify the expected behavior:

```
ALWAYS (send_start -> ( !send_done UNTIL received ))
```

∎

5.4.2 Exposing function execution

Exposing the start and end of execution of user-defined functions allows the specification of pre- and post-conditions and is done by the declarations

```
location loc3 ``% bar::foo()'':entry
location loc4 ``% bar::foo()'':exit
```

Both the `call` primitive and the `entry` primitive signal that the function `foo()` is about to execute, but they hold in different locations in the user text. The `call` primitive holds at the call-site of `foo()`, while the `entry` primitive holds immediately before the execution of the first statement of `foo()`. Similar is the distinction between `return` and `exit`: `exit` holds immediately after the last statement of the function, while `return` holds immediately after the function returns.

Another key distinction is that `entry` and `exit` can only be used with user-defined functions. This restriction is motivated by the property language presented in Chapter 3. To attain generality, library code is treated as a black box, and the state of library objects is allowed to be exposed only through publicly declared interfaces.

5.4.3 Exposing function parameters and return values

Exposing the values of function parameters according to their location in the parameter list allows the specification of pre-conditions without requiring the user to know the name of the actual parameters used in the function body declaration. The primitive declaration

```
value int var1 ''float bar::foo(...)'':2
```

declares an (integer) variable `var1` whose value is equal to the 2-nd parameter of function `foo()` at the time when the function starts executing. Notice that the function may be defined in a library, but the function call is a part of the user code. Following the specification framework presented in Chapter 3, we would like to expose the function parameters for both user-defined and library-defined functions. However, due to a limitation of `AspectC++`, this primitive is available only for user-defined functions.

Exposing the return values of functions allows the specification of post-conditions for functions. The variable `ret` in the following declaration is assigned the return value of `foo()`:

```
value ret ''float bar::foo(...)'':0
```

Unlike the previous primitive, this one is available for both user-defined and library-defined functions.

Example 10 *Referring to the producer-consumer model described earlier, suppose that if the channel is full,* `send()` *is supposed to block for a few cycles and then timeout and return* NULL*. We want to assert that if the return value is* NULL *then the channel does not have free space. Here we assume that the channel defines function* `num_available()`*, and the channel instance is called* `my_channel`*. We declare the following primitives:*

```
location send_done ``% producer::send()'':return
value return_value  ``STATUS_T producer::send()'':0
```

and use them in the property:

```
ALWAYS ((send_done && ``(ret_val == NULL)'') ->
     ``( my_channel->num_available() == 0 )'' )
```

∎

Notice that the monitor generator interprets every quoted string as an atomic proposition, hence the second "->" is not interpreted as an implication.

5.4.4 Exposing syntax

Sometimes it may be desirable to assert that a particular C++ statement (or a set of C++ statements) is reached during the execution of the model. In other cases, assertions may need to be checked immediately before or immediately after some statements. This requires exposing the syntax of the user code to the monitoring framework. The primitive declaration

```
plocation loc5 ``/ *a'':before
```

declares a Boolean atomic proposition `loc5` that holds immediately before the execution of all statements where we divide by the variable `a`. The dual,

```
plocation loc6 ``balance *= *.*;'':after
```

holds immediately after all statements matching the regular expression ``balance *= *.*;''.

5.4.5 Exposing private variables

Referring to values of private or protected class variables (i.e., local storage) of modules is critical for white-box validation of models. The declaration

```
makevisible my_class
```

is issued after a particular property, and declares a SystemC module or a C++ class `my_class` fully visible to the monitoring framework. This enables references to its class variables in the monitor corresponding to the property.

5.5 Implementation

Our implementation uses the monitoring framework described in Chapter 4 to obtain references to the monitors from the instrumented user code. The monitors are agnostic about the semantics of the primitive Booleans used in the property: these primitives are treated as Boolean expressions that determine state change in the monitors. The monitors expect these Boolean primitives to be assigned correct values prior to the execution of monitor steps. In this section we show how the primitives described in Section 5.4 are assigned values.

5.5.1 Exposing function calls

Exposing `location` primitives, e.g.,

```
location loc1 ``% bar::foo()'':call
```

is done by creating a communication interface between the user code and the monitor, and then instrumenting the user code to communicate with the monitor. The monitor defines a callback function `callback_loc1()` and a local Boolean variable `loc1`. The monitor expects that the callback function `callback_loc1()` is be called from

the user code as soon as the execution of the user code reaches the function call `bar::foo()`.

During initialization the monitor sets all Boolean variables corresponding to user-defined `locations` to ***false***. The execution of a callback function `callback_loc1()` consists of the following sequence of steps: 1) The associated Boolean variable `loc1` is set to ***true***; 2) The monitor executes one step; and 3) `loc1` is set to ***false***.

The instrumentation of the user code must call the monitor's `callback_loc1()` function immediately before the function call to `bar::foo()`. Our implementation creates an AOP advice that carries out the communication with the monitor from the user code:

```
1  advice call("% bar::foo()"): before() {
2    // Start new inner scope
3    {
4      extern sc_core::mon_observer* observer;
5      mon_prototype* mp = observer->get_monitor_by_index(42);
6      my_monitor42* mon42 = (my_monitor42*) mp;
7
8      // This callback implemented only by my_monitor42
9      mon42->callback_loc1();
10   }
11 } // advice
```

Listing 5.1: AOP advice to expose function calls of `bar::foo()`.

The AOP advice in Listing 5.1 uses an inner scope to prevent variable name conflicts. This also ensures that no variable declared during the execution of the advice code will remain in scope after the end of the execution of the advice code. A pointer to the `mon_observer` object `observer` is obtained using its external declaration. This example assumes that the 42-nd property refers to the `location` declaration `loc1`. The function call `observer->get_monitor_by_index(42)` returns a pointer to the 42-nd monitor as an abstract `mon_prototype` object (see Chapter 4). It is recast to the type `my_monitor42*` so that the callback function defined by the 42-nd monitor (i.e., `mon42->callback_loc1()`) can be used.

Exposing the locations immediately after the return of a function call is done in a similar way as in Listing 5.1, but replacing `before` with `after` in the generated AOP advice (see Listing 5.2). The advice is activated upon the function's return and it calls the monitor's callback function corresponding to the `location` primitive:

```
1  advice call("% bar::foo()"): after() {
2    // Start new inner scope
3    {
4      extern sc_core::mon_observer* observer;
5      mon_prototype* mp = observer->get_monitor_by_index(21);
6      my_monitor42* mon21 = (my_monitor21*) mp;
7
8      // This callback implemented only by my_monitor21
9      mon21->callback_loc1();
10   }
11 } // advice
```

Listing 5.2: AOP advice to expose the return of function `bar::foo()`.

5.5.2 Exposing function execution

Primitives associated with the start and the end of functions are handled by the monitors in the same way as call and return primitives: the monitor declares a Boolean variable corresponding to the `location` primitive, and this variable is set to ***true*** via a callback. In order to instrument the user code we generate an AOP advice that is activated when the monitored function starts or finishes executing.

As an example, if the user declares

`location loc3 ''% bar::foo()'':entry`

our implementation generates an AOP advice very similar to the one presented in Listing 5.1, but replacing `advice call()` by `advice execution()` (see Listing 5.3. This changes the location where the instrumented code is inserted: instead of instrumenting the call-site, the advice is inserted at the beginning of the function body.

```
1  advice execution("% bar::foo()"): before() {
```

```
2    // Start new inner scope
3    {
4      extern sc_core::mon_observer* observer;
5      mon_prototype* mp = observer->get_monitor_by_index(42);
6      my_monitor42* mon42 = (my_monitor42*) mp;
7
8      // This callback implemented only by my_monitor42
9      mon42->callback_loc1();
10   }
11 } // advice
```

Listing 5.3: AOP advice to expose start of execution of function `bar::foo()`.

For the declaration

```
location loc3 ``% bar::foo()'':exit
```

we generate an advice similar to the one presented in Listing 5.1, but changing the first line to `advice execution(``% bar::foo()''):after`:

```
1  advice execution("% bar::foo()"): after() {
2    // Start new inner scope
3    {
4      extern sc_core::mon_observer* observer;
5      mon_prototype* mp = observer->get_monitor_by_index(42);
6      my_monitor42* mon42 = (my_monitor42*) mp;
7
8      // This callback implemented only by my_monitor42
9      mon42->callback_loc1();
10   }
11 } // advice
```

Listing 5.4: AOP advice to expose end of execution of function `bar::foo()`.

5.5.3 Exposing function parameters and return values

For each monitored `value` primitive `myval`, e.g.,

```
value int myval ``% bar::foo(...)'':2
```

the monitor defines a callback function `callback_myval(`T `v)`, where T is the type of `myval`. The monitor also declares a local variable `value_of_myval` of type T. The monitor expects that `callback_myval()` is called upon execution of the function `bar::foo()`.

The function `callback_myval()` sets the value of `val_of_myval` equal to the value of the parameter `v` and returns without running the monitor. The callback function here serves only as a communication channel to expose the value of the monitored parameter, but the property is not automatically evaluated at this callback. If the user wishes a property to be evaluated when the monitored function is executed, the user can add the execution of the function to the list of sample points for the property.

Instrumenting the user code is done by an automatically generated AOP advice. The advice for

```
value int myval ``% bar::foo(...)'':2
```

is presented in Listing 5.5. The advice uses the built-in AOP function call `tjp->arg(n)` which exposes the n-th parameter of the function (counting up from 0). `tjp->arg(n)` returns a `void*` pointer that needs to be cast to the type of `myval` before it is passed to the monitor via the callback.

```
1  advice execution("int driver::foo(...)"): before() {
2    // Inner scope
3    {
4      extern sc_core::mon_observer* observer;
5      mon_prototype* mp = observer->get_monitor_by_index(42);
6      my_monitor42* mon42 = (my_monitor42*) mp;
7
8      // Obtain the value to send to the monitor
9      int value_to_send = (int) *(int *)tjp->arg(1);
10     mon42->callback_myval(value_to_send);
11   }
12 }
```

Listing 5.5: AOP advice to expose function parameters of `bar::foo()`.

5.5.4 Exposing syntax

Declarations of `plocation` primitives, e.g.,

```
plocation loc6 ``balance *= *.*;'':after
```

are handled by the monitor as `location` primitives: for each `plocation` the moni-

tor declares a callback function and a local Boolean variable. The value of the variable is set to ***true*** by the callback, the monitor executes a step, and the variable is set to ***false*** before the callback returns.

Instrumenting the user code to expose statements that match regular expressions cannot be done using the AOP framework. Thus, our implementation pre-processes all user-code files and identifies the source code locations that need to be instrumented, using pattern matching. At each such location we insert code that obtains a reference to the correct monitor and makes the callback. For example, the injected code corresponding to the `plocation` defined above is presented in Listing 5.6:

```
1   // Code fragment injected without using the AOP framework.
2   {
3     extern sc_core::mon_observer* observer;
4     mon_prototype* mp = observer->get_monitor_by_index(42);
5     my_monitor42* mon42 = (my_monitor42*) mp;
6     mon42->callback_loc6();
7   }
```

Listing 5.6: Code fragment injected in the source code to expose syntax.

This code is identical to the AOP advice code presented earlier in Listing 5.1. However, when using the AOP-based instrumentation the advice code executes with the support of the AOP framework. In particular, AOP allows `C++` header files to be inserted automatically in the instrumented source code files. The AOP aspect that is generated by our implementation carries an `#include` directive that injects the monitor header file in all instrumented user-code files; casting the `mon_prototype` objects to the derived monitor types (e.g., `my_monitor42`) would otherwise be impossible. When injecting instrumentation code without the help of the AOP framework, our implementation injects the required `#includes` in the user code where `plocation` primitives are exposed.

5.5.5 Exposing private variables

All modules and channels in SystemC extend the pre-defined objects `sc_module` and `sc_channel`, which are implemented internally as C++ classes. To expose their **`private`** and **`protected`** data members we use C++'s **`friend`** mechanism. Intuitively, a monitored module declares the monitor class as a **`friend`** class, which gives the monitor unrestricted access to all internal data members. We show how to do this automatically via an aspect introduction.

AOP *introductions* allows adding new data members and functions to a class [GSPS01]. However, AOP does not restrict the advice code that can be weaved via an introduction. Since introductions extend the static structure of classes, an introduction advice can also be used to declare the monitoring class as a **`friend`** class. Our implementation generates a named pointcut `reveal()`, in respect to which we define the introduction (Listing 5.7):

```
1 pointcut reveal() = ``bar'' || ``bas'';
2
3 advice reveal() : slice class {
4   friend class monitor0;
5   friend class monitor1;
6   ...
7 };
```

Listing 5.7: AOP advice to expose private and protected data members of modules "bar" and "bas".

Notice that Listing 5.7 specifies explicitly all monitors that require privileged access to modules `bar` and `bas`. It is tempting to use the superclass instead and limit the list to a single declaration, i.e.,

```
friend class mon_prototype
```

and avoid listing the individual monitor classes. This declaration, however, would not have the desired effect. In C++, a subclass cannot claim friendship by inheritance. One of the reasons for this restriction is that otherwise a malicious class would be able to extend a trusted class, and would gain trivially unrestricted access.

5.6 Experimental evaluation

We used version 2.2.0 of the OSCI simulator, which was modified using the monitoring framework presented in Chapter 4 and compiled using the default settings. We ran all experiments on Ada, Rice's Cray XD1 compute cluster (`rcsg.rice.edu/ada`). Each of Ada's nodes has two dual core 2.2 GHz AMD Opteron 275 CPUs and 8GB of RAM.

We used the SystemC model implementing a system for reserving and purchasing airplane tickets, which was presented earlier in Chapter 4. Recall that the model simulates user-submitted requests for air travel and the system uses a randomly generated flight database to find a direct flight or a sequence of up to three connecting flights. Those are returned to the user for approval, payment and booking. Internally the system uses modules connected by bounded-capacity channels. This model is intended to run forever. It approximates actual subsystems currently used in hardware design.

We measured the performance by simulating for 1 million clock cycles. The average wall-clock execution time of the system over 10 runs without instrumentation is ~33 seconds. We call this the "baseline" execution. We next added a simple `assert true` specification that is checked at increasing number of locations in the source code. For each experiment we wrote a configuration file containing the specification and declared the locations at which the specification was to be checked. Our implementation generated the corresponding AOP advice and the monitor. The advice was then weaved into the user source code using `AspectC++`. The instrumented code and the monitor code were compiled and executed using the same input parameters as the baseline execution. At the end of execution the monitor reported how many times it had been called, which corresponds to the number of times the instrumentation had been exercised. Since we are using a very simple monitor, any slow-down of the execution is due to the instrumentation.

Figure 5.1 presents the number of times the monitor was called and the corre-

sponding execution overhead of the user-code as a percentage of the baseline execution time. We observe a linear increase in the overhead as we increase the number of calls.

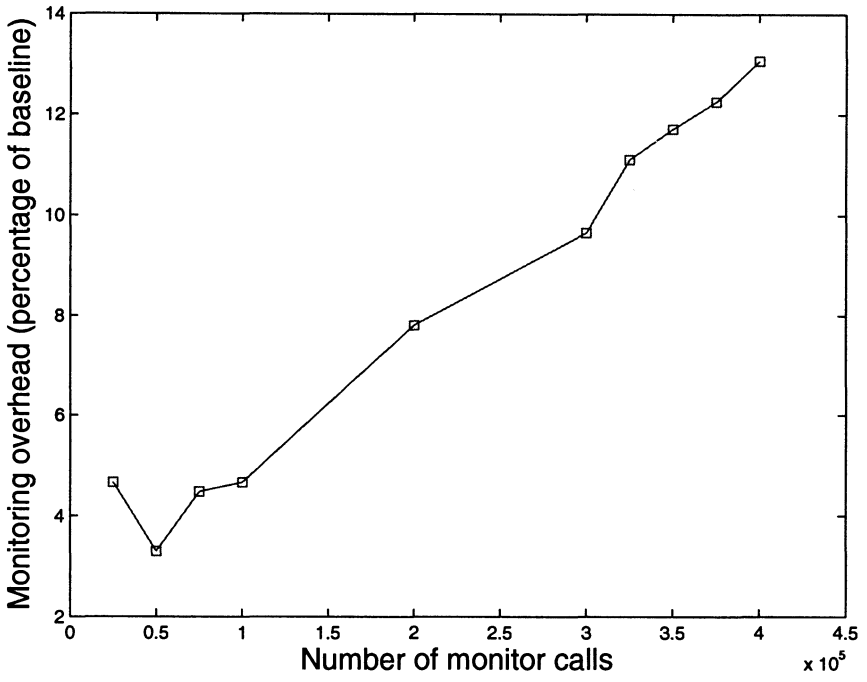


Figure 5.1 : Instrumentation overhead as a percentage of the baseline runtime increases linearly as we increase the number of calls to the monitor

Figure 5.2 shows the cost of the instrumentation per monitor call, as a percentage of the baseline execution. Our data suggest that there is a fixed cost of the instrumentation, which, when amortized over more and more calls, leads to lower average cost. The average cost per call stabilizes after 300,000 calls, and is less than $0.5 \times 10^{-4}\%$.

5.7 Summary and discussion

A successful monitoring framework for SystemC requires access to internal variables of modules and channels, and the ability to trace the execution of threads and methods.

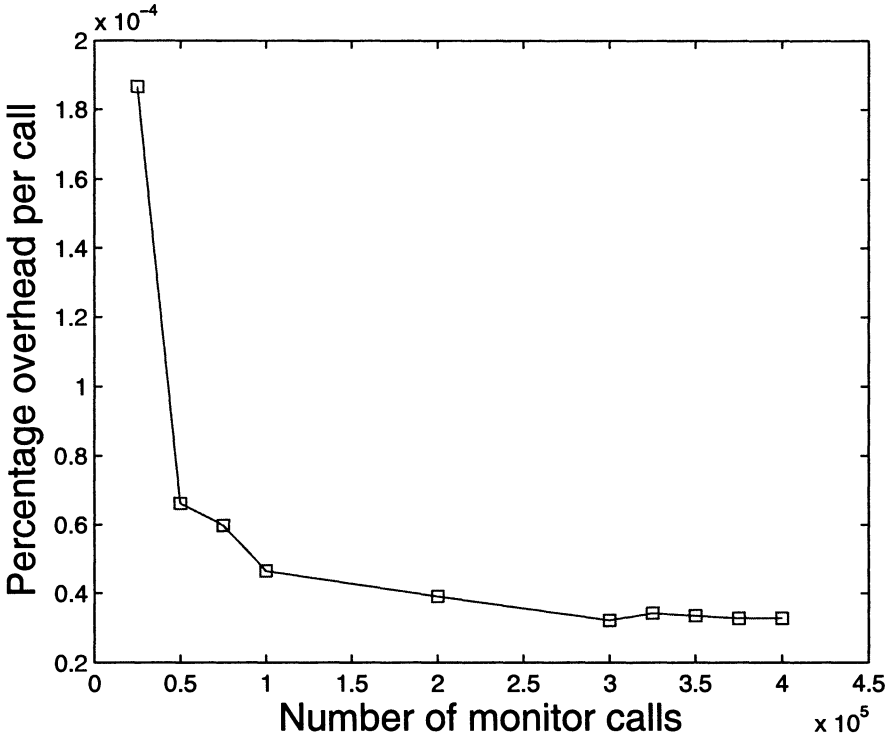


Figure 5.2 : Instrumentation overhead per monitor call as a percentage of the baseline runtime

This chapter describes a framework for exposing a rich set of user code primitives via automated generation of AOP advice and automated instrumentation of source code files. The mechanisms presented here are easy to use and do not require the users to instrument the code manually or to be experts in AOP.

The solution presented here allows the users to declare specification primitives referring to the value of internal variables, the values of parameters passed to function calls, and function return values. Tracing of the execution of processes is enabled by allowing statement execution or function call to be used as an atomic proposition. The correct behavior of the model can then be specified by forming temporal formulas and clock expressions using these primitives, without requiring manual instrumentation of the user code to expose the primitives. The experimental results show that the

automated instrumentation of the user code leads to very low execution overhead.

One limitation of the instrumentation approach presented is that arguments of functions calls are not exposed if the called function is not defined in the user code. This affects the monitoring of calls of library functions. We expect that future versions of `AspectC++` will include this functionality, thereby removing the limitation from the instrumentation approach presented here.

Chapter 6

Optimized Temporal Monitors

6.1 Introduction and motivation

Recall that assertion-based verification (ABV) allows the designer to asserts properties that capture design intent in a formal language, e.g., PSL [EF06] or SVA [VR05]. The model then is verified against the properties using dynamic verification or formal verification techniques. The focus of formal verification is the correctness of the model under verification (MUV), while the focus of dynamic verification is the correctness of a particular *execution trace* of the MUV.

Leucher and Schallhart give the following definition of dynamic verification.

Definition 1 (Dynamic verification [LS09]) Dynamic verification *is the discipline of computer science that deals with the study, development, and application of those verification techniques that allow checking whether an* execution trace *of an MUV satisfies or violates a given correctness property.*

Checking whether an execution trace meets the correctness property is typically performed by a monitor. As pointed out earlier in Chapter 1 and Chapter 4, for simple properties it may be feasible to write the monitors manually (c.f., [GBA+99]); however, in most industrial workflows, writing and maintaining monitors manually would be an extremely high-cost, labor-intensive, and error-prone process [ABG+00]. This has inspired both academia and industry to search for methods for automated generation of monitors from temporal properties.

Formal, automata-theoretic foundations for monitor generation for temporal properties have been laid out in [KV01], which showed how a deterministic finite word

automaton (DFW) can be generated from a temporal property such that the automaton accepts the finite traces that violate the property. Many works have elaborated on that approach, cf. [AKT+06, BLS06, dR05, FS04, Gei01, GH01]); see the discussion below of related work. Many of these works, e.g. [AKT+06], handle only safety properties, which are properties whose failure is always witnessed by a finite trace. Here, as in [dR05], we follow the framework of [KV01] in its full generality and we consider all properties whose failure may be witnessed by a finite trace. For example, the failure of the property "eventually q" can never be witnessed by a finite trace, but the failure of the property "always p and eventually q" may be witnessed by a finite trace.

Apriori it is not clear how monitor size is related to performance, and most works on this subject have focused on underlying algorithmics or heuristics to generate smaller monitors or on fast monitor generation. This work shifts the focus toward optimizing the runtime overhead that monitor execution adds to simulation time. We believe that this reflects more accurately the priorities of the industrial applications of monitors [AKT+06]. A large model may be accompanied by thousands of monitors [BZ08], most of which are compiled once and executed many times, so lower runtime overhead is a crucial optimization criterion, much more than monitor size or monitor-generation time. In this paper we identify several algorithmic choices that need to be made when generating temporal monitors for monitoring frameworks implemented in software. We conduct extensive experimentation to identify the choices that lead to superior performance.

Identified in this chapter are four issues in monitor generation: *state minimization*, should nondeterministic automata be determinized online or offline; *alphabet representation*, should alphabet letters be represented explicitly or symbolically; *alphabet minimization*, should inconsistent alphabet letters be eliminated; and *monitor encoding*, how should the transition function of the monitor be expressed. These options give us a *configuration space* of 27 different ways of generating a monitor from

nondeterministic automata.

We evaluate performance using a model representing an adder that was described in Chapter 4. Its advantages are that it is scalable and creates events at many different level of abstractions. For temporal properties we use linear temporal formulas. We use a mixture of pattern and random formulas, giving us a collection of over 1,300 temporal properties. Our experiments identify a specific configuration that offers the best performance in terms of runtime overhead.

6.2 Related work

Most related papers that deal with monitoring focus on simplifying the monitor or reducing the number of states. Using smaller monitors is important for in-circuit monitoring, say, for post-silicon verification [BZ08], but for pre-silicon verification, using lower-overhead monitors is more important. Very few prior works focus on minimizing runtime overhead.

Several papers focus on building monitors for *informative prefixes*, which are prefixes that violate input assertions in an "informative way." Kupferman and Vardi [KV01] define informative prefixes and show how to use alternating automata to construct nondeterministic finite word automata (NFW) of size $2^{O(\psi)}$ that accept informative prefixes of an LTL formula ψ. Kupferman and Lampert [KL06] use a related idea to construct NFW automata of size $2^{O(\psi)}$ that accept at least one prefix of every trace that violates a safety property ψ. Two constructions that build monitors for informative prefixes are by Geilen [Gei01] and by Finkbeiner and Sipma [FS04]. Geilen's construction is based on the automata-theoretic construction of [GPVW95], while that of Finkbeiner and Sipma is based on the alternating-automata framework of [KV01]. Neither provide experimental results.

Armoni et al. [AKT+06] describe an implementation based on [KV01] in the context of hardware verification. Their experimental results focus on both monitor size and runtime overhead. They showed that the overhead is significantly lower than

that of commercial simulators. Stolz and Bodden [SB06] use monitors constructed from alternating automata to check specifications of Java programs, but do not give experimental results.

Giannakopoulou and Havelund [GH01] apply the construction of [GPVW95] to produce nondeterministic monitors for **X**–free LTL formulas, and simulate a deterministic monitor on the fly. They provide one experimental result from the early testing of their implementation. A weakness of their approach is that their LTL semantics distinguishes between finite and infinite traces, which implies that LTL properties may have the different meaning in the context of dynamic and formal verification.

Morin-Allory and Borione [MAB06] show how to construct hardware modules implementing monitors for properties expressed using the *simple subset* [PSL07] of PSL. Pierre and Ferro [PF08] describe an implementation based on this construction, and present some experimental results that show runtime overhead, but do not present any attempts to minimize it. Boulé and Zilic [BZ08] show a rewriting-based technique for constructing monitors for the simple subset of PSL. They provide substantial experimental results, but focus on the monitor size and not on runtime overhead.

D'Amorim and Roşu [dR05] show how to construct monitors for *minimal bad prefixes* of temporal properties without any restrictions whether the property is a safety property or not. They construct a nondeterministic finite automaton of size $2^{O(\psi)}$ that extracts the safety content from ψ, and simulate a deterministic monitor on the fly. They present two optimizations: one reduces the size of the automaton, while the other searches for a good ordering of the outgoing transitions so that the overall expected cost of running the monitor will be smallest. They measure experimentally the size of the monitors for a few properties, but do not measure their runtime performance. A similar construction, but without any of the optimizations, is also described by Bauer et al. [BLS06].

6.3 Theoretical background

6.3.1 Bad prefixes

Let AP be a finite set of atomic propositions and let $\Sigma = 2^{AP}$ be a finite alphabet. Given a temporal specification ψ over AP, we denote the set of models of the specification with $\mathcal{L}(\psi) = \{w \in \Sigma^\omega \mid w \models \psi\}$. Intuitively, $\mathcal{L}(\psi)$ represents the set of valid executions given by property ψ.

Let $u \in \Sigma^*$ denote a finite word (in the context of dynamic verification, u represents a particular execution trace of the MUV). Clearly, checking the correctness of u with respect to a specification ψ is equivalent to checking if u is an element of $\mathcal{L}(\psi)$. Dynamic verification produces finite-length execution traces, and because of that it focuses on properties whose failure can be detected during finite execution. The concept of a *bad prefix*, first proposed by Kupferman and Vardi [KV01], formalizes this idea of failure after finite number of steps:

Definition 2 (Bad Prefix [KV01]) *A finite word $u \in \Sigma^*$ is a* bad prefix *for some language $\mathcal{L}$ iff for all $y \in \Sigma^\omega$, $x \cdot y \notin \mathcal{L}$.*

Intuitively, a bad prefix cannot be extended to an infinite word in $\mathcal{L}$. A *minimal bad prefix* does not have a bad prefix as a strict prefix.

6.3.2 Automata on infinite words

Temporal properties of non-terminating systems are often compiled to non-deterministic automata on infinite words; such automata were first formalized by Büchi [Büc62] and are named *Büchi automata* in his honor.

Definition 3 (Büchi automaton [Büc62]) *A* nondeterministic Büchi automaton on words *(NBW) is a tuple $\mathcal{A} = \langle \Sigma, Q, \delta, Q^0, F \rangle$, where Σ is a finite alphabet, $Q \neq \emptyset$ is a finite set of states, $\delta : Q \times \Sigma \to 2^Q$ is a transition function, $Q^0 \subseteq Q$ is a set of initial states, and $F \subseteq Q$ is a set of accepting states.*

Given a NBW $\mathcal{A} = \langle \Sigma, Q, \delta, Q^0, F \rangle$, if $q' \in \delta(q, \sigma)$ then we say that we have a transition from q to q' labeled by σ. We extend the transition function $\delta : Q \times \Sigma \rightarrow 2^Q$ to $\delta : 2^Q \times \Sigma^* \rightarrow 2^Q$ as follows: for all $Q' \subseteq Q$, $\delta(Q', a) = \cup_{q \in Q'} \delta(q, a)$, and for all $\sigma \in \Sigma^*$, $\delta(q, a\sigma) = \delta(\delta(q, a), \sigma)$.

A *run* of $\mathcal{A}$ on a word $w = a_0 a_1 \ldots \in \Sigma^\omega$ is a sequence of states $q_0 q_1 \ldots$, such that $q_0 \in Q^0$ and $q_{i+1} \in \delta(q_i, a_i)$ for some $a_i \in \Sigma$. For a run r, let $Inf(r)$ denote the states visited infinitely often. Notice that since $\mathcal{A}$ has a finite number of states, $Inf(r)$ is always non-empty. A run r of $\mathcal{A}$ is called *accepting* iff $Inf(r) \cap F \neq \emptyset$.

The word w is accepted by $\mathcal{A}$ if there is an accepting run of $\mathcal{A}$ on w. For a given Linear-Time Logic (LTL) or PSL/SVA formula ψ, we can construct an NBW that accepts precisely $\mathcal{L}(\psi)$ [VW94]. This work uses SPOT [DLP04], an LTL-to-NBW tool, which is among the best available in terms of performance [RV07]. Analogous translators are available for PSL and SVA (see, e.g., [BFH05]).

6.3.3 Automata on finite words

A *nondeterministic automaton on finite words* (NFW) is a tuple $\mathcal{A} = \langle \Sigma, Q, \delta, Q^0, F \rangle$. An NFW can be determinized by applying the *subset construction*, yielding a *deterministic automaton on finite words* (DFW) $\mathcal{A}' = \langle \Sigma, 2^Q, \delta', \{Q^0\}, F' \rangle$, where $\delta'(S, a) = \bigcup_{s \in S} \delta(s, a)$ and $F' = \{S : S \cap F \neq \emptyset\}$. For a given NFW $\mathcal{A}$, there is a canonical minimal DFW that accepts $\mathcal{L}(\mathcal{A})$ [HU79]. In the remainder of this chapter, given an LTL formula ψ, we use $\mathcal{A}_{NBW}(\psi)$ to mean an NBW that accepts $\mathcal{L}(\psi)$, and $\mathcal{A}_{NFW}(\psi)$ (respectively, $\mathcal{A}_{DFW}(\psi)$) to mean a an NFW (respectively, DFW) that rejects the minimal bad prefixes of $\mathcal{L}(\psi)$.

Building a monitor for a property ψ requires building $\mathcal{A}_{DFW}(\psi)$. Our work is based on the construction by d'Amorim and Roşu [dR05], which produces $\mathcal{A}_{NFW}(\psi)$. Their construction assumes an efficient algorithm for constructing $\mathcal{A}_{NBW}(\psi)$ (e.g., [DLP04], when the specification is expressed in LTL, or [BFH05], when the specification is in PSL). Below we sketch the construction of [dR05] and then we show how we construct

$\mathcal{A}_{DFW}(\psi)$.

Given an NBW $\mathcal{A} = \langle \Sigma, Q, \delta, Q^0, F \rangle$ and a state $q \in Q$, define $\mathcal{A}^q = \langle \Sigma, Q, \delta, q, F \rangle$. Intuitively, $\mathcal{A}^q$ is the NBW automaton defined over the structure of $\mathcal{A}$ but replacing the set initial states with $\{q\}$. Let $empty(\mathcal{A}) \subseteq Q$ consist of all states $q \in Q$ such that $\mathcal{L}(\mathcal{A}^q) = \emptyset\}$, i.e., all states that cannot start an accepting run. The states in $empty(\mathcal{A})$ are "unnecessary" in $\mathcal{A}$, because they cannot appear on an accepting run. We can compute $empty(\mathcal{A})$ efficiently using nested depth-first search [CVWY92]. Deleting the states in $empty(\mathcal{A})$ is done using the function call `spot::tgba::prune_scc()`, which is available in SPOT.

To generate a monitor for ψ, d'Amorim and Roşu build $\mathcal{A}_{NBW}(\psi)$ and remove $empty(\mathcal{A}_{NBW}(\psi))$. They then treat the resulting automaton as an NFW, with all states taken to be accepting states. That is, the resulting NFW is $\mathcal{A} = \langle \Sigma, Q', \delta', Q^0 \cap Q', Q' \rangle$, where $Q' = Q - empty(\mathcal{A})$, and δ' is δ restricted to $Q' \times \Sigma$. When started with $\mathcal{A}_{NBW}(\psi)$, we call the resulting automaton $\mathcal{A}^{dR}_{NFW}(\psi)$.

Theorem 1 [dR05] *$\mathcal{A}^{dR}_{NFW}(\psi)$ rejects precisely all bad prefixes of ψ.*

6.3.4 From NFW to monitors

From now on we refer to $\mathcal{A}^{dR}_{NFW}(\psi)$ simply as $\mathcal{A}_{NFW}(\psi)$. $\mathcal{A}_{NFW}(\psi)$ is not useful as a monitor because of its nondeterminism. One way to construct a monitor from $\mathcal{A}_{NFW}(\psi)$ is to determinize it explicitly using the subset construction. In the worst case the resulting $\mathcal{A}_{DFW}(\psi)$ is of size exponential of the size of $\mathcal{A}_{NFW}(\psi)$, which is why explicit determinization has rarely been used. We note, however, that we can minimize $\mathcal{A}_{DFW}(\psi)$, getting a minimal DFW. It is not clear, *a priori*, what impact this determinization and minimization will have on runtime overhead.

An alternative way of constructing a monitor from $\mathcal{A}_{NFW}(\psi)$ that avoids the potential for exponential blow up of the number of states is to use $\mathcal{A}_{NFW}(\psi)$ to simulate a deterministic monitor on the fly. d'Amorim and Roşu describe such a construction in terms of nondeterministic multi-transitions and binary transition

trees [dR05]. Instead of introducing these formalisms, here we use instead the approach in [AKT+06, TV05], which presents the same concept in automata-theoretic terms. The idea in both papers is to perform the subset construction on the fly, as we read the inputs from the trace. Given $\mathcal{A}_{NFW}(\psi) = \langle \Sigma, Q, \delta, Q^0, Q \rangle$ and a finite trace $a_0, \ldots, a_{n-1}$, we construct a run $P_0, \ldots, P_n$ of $\mathcal{A}_{DFW}(\psi)$ as follows: $P_0 = \{Q^0\}$ and $P_{i+1} = \bigcup_{s \in P_i} \delta(s, a_i)$. The run is accepting iff $P_i = \emptyset$ for some $i \geq 0$, which means that we have read a bad prefix. Notice that each P_i is of size linear in the size of $\mathcal{A}_{NFW}(\psi)$, thus we have avoided the exponential blowup of the determinization construction, with the price of having to compute transitions on the fly [AKT+06, TV05].

6.4 Monitor generation

We now describe various issues that arise when constructing $\mathcal{A}_{DFW}(\psi)$.

6.4.1 State minimization

As noted above, we can construct $\mathcal{A}_{DFW}(\psi)$ on the fly. We discuss in detail below how to express $\mathcal{A}_{DFW}(\psi)$ as a collection of `C++` expressions. The alternative is to feed $\mathcal{A}_{NFW}(\psi)$ into a tool that constructs a minimal equivalent $\mathcal{A}_{DFW}(\psi)$. We use the `BRICS Automaton` tool [Mø04]. Clearly, determinization and minimization, as well as subsequent `C++` compilation, may incur a nontrivial computational cost. Still, such a cost might be justifiable if the result is reduced *runtime* overhead, as assertions have to be compiled only once, but then run many times. A key question we want to answer is whether it is worthwhile to determinize $\mathcal{A}_{NFW}(\psi)$ explicitly, rather than on the fly.

6.4.2 Alphabet representation

In our formalism, the alphabet Σ of $\mathcal{A}_{NFW}(\psi)$ is $\Sigma = 2^{AP}$, where AP is the set of atomic propositions appearing in ψ. In practice, tools that generate $\mathcal{A}_{NBW}(\psi)$ (SPOT in our case) often use $\mathcal{B}(AP)$, the set of Boolean formulas over AP, as the

automaton alphabet: a transition from state q to state q' labeled by the formula θ is a shortcut to denote all transitions from q to q' labeled by $\sigma \in 2^{AP}$, when σ satisfies θ. When constructing $\mathcal{A}_{DFW}(\psi)$ on the fly, we can use formulas as letters. Automata-theoretic algorithms for determinization and minimization of NFWs, however, require comparing elements of Σ, which makes it impractical to use Boolean formulas for letters. We need a different way, therefore, to describe our alphabet[1] . Below we show two ways to describe the alphabet of $\mathcal{A}_{NFW}(\psi)$ in terms of 16-bit integers.

Assignment-based representation

The explicit approach is to represent Boolean formulas in terms of their satisfying truth *assignments*. Let $AP = \{p_1, p_2, \ldots, p_n\}$ and let $\mathcal{F}(p_1, p_2, \ldots, p_n)$ be a Boolean function. An *assignment* to AP is an n-bit vector $\mathbf{a} = [a_1, a_2, \ldots, a_n]$. An assignment $\mathbf{a}$ *satisfies* $\mathcal{F}$ iff $\mathcal{F}(a_1, a_2, \ldots, a_n)$ evaluates to 1. Let A^n be the set of all n-bit vectors and let $I : A^n \rightarrow \mathbb{Z}_+$ return the integer whose binary representation is $\mathbf{a}$, i.e., $I(a) = a_1 2^{n-1} + a_2 2^{n-2} + \ldots + a_n 2^0$. We define $sat(\mathcal{F}) = \{I(a) : a \text{ satisfies } \mathcal{F}\}$. Thus, the explicit representation of the automaton $\mathcal{A}_{NFW}(\psi) = \langle \mathcal{B}(AP), Q, \delta, Q^0, F \rangle$ is $\mathcal{A}^{ass}_{NFW}(\psi) = \langle \{0, \ldots, 2^n - 1\}, Q, \delta_{ass}, Q^0, F \rangle$, where $q' \in \delta_{ass}(q, z)$ iff $q' \in \delta(q, \sigma)$ and $z \in sat(\sigma)$.

BDD-based representation

The symbolic approach to alphabet representation leverages the fact that Ordered Binary Decision Diagrams (*BDD*s) [Bry86, Bry92] provide canonical representations of Boolean functions. A BDD is a rooted, directed, acyclic graph with one or two terminal nodes labeled **0** or **1**, and a set of variable nodes of out-degree two. The variables respect a given linear order on all paths from the root to a leaf. Each path

[1] `BRICS Automaton` represents the alphabet of the automaton as Unicode characters, which have 1-to-1 correspondence to the set of 16-bit integers.

represents an assignment to each of the variables on the path. For a fixed variable order, two BDDs are the same iff the Boolean formulas they represent are the same.

As an example, Figure 6.1 shows the BDD representation of the Boolean formula $(x_1 \leftrightarrow x_2) \vee x_3$. Solid edges represent assignment of ***true***, and dashed edges represent assignment of ***false***.

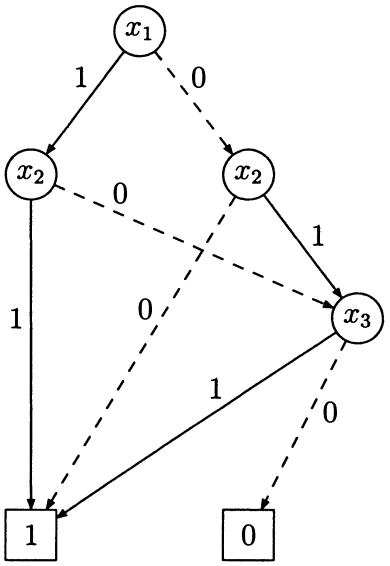


Figure 6.1 : A BDD representing the Boolean formula $(x_1 \leftrightarrow x_2) \vee x_3$

The symbolic approach uses SPOT's `spot::tgba_reachable_iterator_breadth_first::process_link()` function call to get a reference to all Boolean formulas that appear as transition labels in $\mathcal{A}_{NFW}(\psi)$. The formulas are enumerated using their BDD representation (using SPOT's `spot::tgba_succ_iterator::current_condition()` function call), and each unique formula is assigned a unique integer. We thus obtain $\mathcal{A}^{bdd}_{NFW}(\psi)$ by replacing transitions labeled by Boolean formulas with transitions labeled by the corresponding integers. While the size of $\mathcal{B}(AP)$ is doubly exponential in $|AP|$, the automaton $\mathcal{A}_{NBW}(\psi)$ is exponential in $|\psi|$, so the number of Boolean formulas used in the automaton is at most exponential in $|\psi|$.

From NFW to DFW

We provide both $\mathcal{A}^{ass}_{NFW}(\psi)$ and $\mathcal{A}^{bdd}_{NFW}(\psi)$ as inputs to `BRICS Automaton`, producing, respectively, minimized $\mathcal{A}^{ass}_{DFW}(\psi)$ and $\mathcal{A}^{bdd}_{DFW}(\psi)$. We note that neither of these two approaches is *a priori* a better choice. LTL–to–automata tools use Boolean formulas rather than assignments to reduce the number of transitions in the generated *nondeterministic* automata, but when using $\mathcal{A}^{bdd}_{DFW}(\psi)$ as a monitor, the trace we monitor is a sequence of truth assignments, and $\mathcal{A}^{bdd}_{DFW}(\psi)$ is not deterministic with respect to truth assignments. As a consequence, there is no guarantee that at each step of the monitor at most one state is reachable.

6.4.3 Alphabet minimization

While propositional temporal specification languages are based on Boolean atomic propositions, they are often used to specify properties involving non-Boolean variables. For example, we may have the atomic formulas `(a == 0)`, `(a == 1)`, and `(a > 1)` in a specification involving the values of a variable **`int`** `a`. Notice that in this example not all assignments in 2^{AP} are consistent. For example, the assignment `(a == 0) && (a == 1)` is not consistent. By eliminating inconsistent assignments we may be able to reduce the number of letters in the alphabet exponentially. Identifying inconsistent assignments requires calling an SMT (Satisfiability-Modulo-Theory) solver [dMB08]. Here we would need an SMT solver that can handle arbitrary `C++` expressions that evaluate to type **`bool`**. Not having access to such an SMT solver, we use the compiler as an improvised SMT solver.

A set of techniques called *constant folding* allow compilers to reduce constant expressions to a single value at compile time (see, e.g., [CT04]). When an expression contains variables instead of constants, the compiler uses *constant propagation* to substitute values of variables in subsequent subexpressions involving the variables. In some cases the compiler is able to deduce that an expression contains two mutually exclusive subexpressions, and issues a warning during compilation. We con-

struct a function that uses conjunctions of atomic formulas as conditionals for dummy `if/then` expressions, and compile the function (we use `gcc 4.0.3`). To gauge the effectiveness of this optimization we apply it using two sets of conjunctions. *Full alphabet minimization* uses all possible conjunctions involving atomic formulas or their negations, while *partial alphabet minimization* uses only conjunctions that contain each atomic formula, positively or negatively.

We compile the function and then parse the compiler warnings that identify inconsistent conjunctions. Prior to compiling the Büchi automaton we augment the original temporal formula to exclude those conjunctions from consideration. For example, if `(a == 0) && (a == 1)` is identified as an inconsistent conjunction, we augment the property ψ to $\psi \wedge \mathbf{G}(!((a == 0) \wedge (a == 1)))$.

6.4.4 Monitor encoding

We describe five ways of encoding automata as C++ monitors. Not all can be used with all automata directly, so we identify the transformations that need to be applied to an automaton before each encoding can be used.

The strategy in all encodings based on automata that are nondeterministic with respect to truth assignments (i.e., $\mathcal{A}_{NFW}(\psi)$ and minimal $\mathcal{A}^{bdd}_{DFW}(\psi)$) is to construct the run $P_0, P_1, \ldots$ of the monitor using two bit-vectors of size $|Q|$: `current[]` and `next[]`. Initially `next[]` is zeroed, and `current[j] = 1` iff $q_j \in Q^0$. Then, after sampling the state of the program, we set `next[k] = 1` iff `current[j] = 1` and if there is a transition from q_j to q_k that is enabled by the current program state. When we are done updating `next[]`, we assign it to `current[]`, zero `next[]`, and then repeat the process at the next sample point. Intuitively, `current[]` keeps track of the set of automaton states that are reachable after seeing the execution trace so far, and `next[]` maintains the set of automaton states that are reachable after completing the current step of the automaton.

Notice that when the underlying automaton is deterministic with respect to truth

assignments (i.e., $\mathcal{A}^{ass}_{DFW}(\psi)$), after each step there are precisely 1 or 0 reachable states. In those cases it is inefficient to use bit-vector encoding of the set of reachable states, because this set is guaranteed to be singleton. Thus, when constructing monitors from deterministic automata, we use `int current` and `int next` to keep track of the run of the automaton. Initially, `current = j` iff q_j is the initial state. Then we set `next = k` iff the transition from q_j to q_k is enabled at the first sample point; since the automaton is deterministic, at most one transition is enabled. We continue in this fashion until the simulation ends or until none of the transitions in the monitor is enabled, indicating a bad prefix.

The details of the way we update `current[]` (respectively, `current`) and `next[]` (respectively, `next`) are reflected in the different encodings. As a running example, we show how to construct a monitor for the property $\varphi = \mathbf{G}(p \rightarrow (q \vee \mathbf{X}q \vee \mathbf{XX}q))$. The first step is to use SPOT to construct a NBW automaton that accepts all traces satisfying φ. Next, we use SPOT to construct $\mathcal{A}_{NFW}(\varphi)$, which is presented in Figure 6.2.

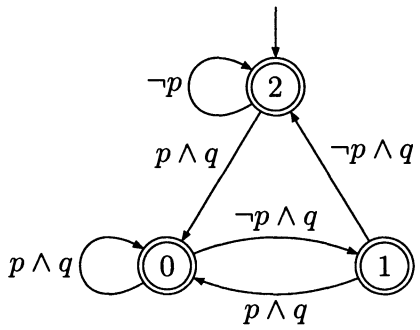


Figure 6.2 : $\mathcal{A}_{NFW}(\varphi)$ constructed from the specification $\varphi = \mathbf{G}(p \rightarrow (q \vee \mathbf{X}q \vee \mathbf{XX}q))$ using the algorithm of d'Amorim and Roşu. Double circles represent accepting states, and state 2 is the initial state.

Nondeterministic encodings

Two novel encodings, which we call `front_nondet` and `back_nondet`, expect that the automaton transitions are Boolean formulas, and do not assume determinism.

Thus, `front_nondet` and `back_nondet` can be used with $\mathcal{A}_{NFW}(\psi)$ directly. They can also be used with $\mathcal{A}^{ass}_{DFW}(\psi)$ and $\mathcal{A}^{bdd}_{DFW}(\psi)$, once we convert back the transition labels from integers to Boolean formulas as follows. In $\mathcal{A}^{ass}_{DFW}(\psi)$, we calculate the assignment corresponding to each integer, and use that assignment to generate a conjunction of atomic formulas or their negations. In $\mathcal{A}^{bdd}_{DFW}(\psi)$ we relabel each transition with the Boolean function whose BDD is represented by the integer label.

The `front_nondet` encoding uses an explicit **`if`** to check if each state s of `current[]` is enabled. For each outgoing transition t from s it then uses a nested **`if`** with a conditional that is a verbatim copy of the transition label of t to determine if the destination state of t is reachable from s. Listing 6.1 illustrates this encoding.

```
1  /**
2   * front_nondet encoding
3   */
4  test_monitor0::step() {
5    if (status == MON_UNDETERMINED) {
6      // Property has not been determined to fail so far
7      num_steps++; // Current length of execution trace
8
9      for (int i = 0; i < 3; i++) {
10       current[i] = next[i];
11       next[i] = 0;
12     }
13
14       if (current[0]) {
15         if(!(p) && (q))
16           next[1] = 1;
17         if((p) && (q))
18           next[0] = 1;
19       } // if
20
21       if (current[1]) {
22         if((p) && (q))
23           next[0] = 1;
24         if(!(p) && (q))
25           next[2] = 1;
26       } // if
27
28       if (current[2]) {
```

```
29          if((p) && (q))
30            next[0] = 1;
31          if(!(p) && (q))
32            next[2] = 1;
33          if(!(p) && !(q))
34            next[2] = 1;
35        } // if
36
37      // Check if there were enabled transitions
38      bool not_stuck = false;
39      for (int i = 0; i < 3; i++) {
40        not_stuck = not_stuck || next[i];
41      }
42
43      if (! not_stuck) {
44        // None of the transitions were enabled
45
46  #ifdef MONITOR_REPORT_FAIL_IMMEDIATELY
47        SC_REPORT_WARNING("Property failed", "Critical error");
48        std::cout << "Property failed after " << num_steps
49                  << " steps" << std::endl;
50  #endif
51        status = MON_FAIL;
52      }
53    } // if (status == MON_UNDETERMINED)
54  } // step()
```

Listing 6.1: Illustrating `front_nondet` encoding of the automaton in Figure 6.2.

The `back_nondet` encoding uses a disjunction that represents all of the ways in which a state in `next[]` can be reached from the currently reachable states. Listing 6.2 illustrates this encoding.

```
1  /**
2   * back_nondet encoding
3   */
4  test_monitor0::step() {
5    if (status == MON_UNDETERMINED) {
6      // Property has not been determined to fail so far
7      num_steps++; // Current length of execution trace
8
9      for (int i = 0; i < 3; i++) {
10       current[i] = next[i];
11       next[i] = 0;
```

```
12      }
13
14      next[0] = ( current[2] && ((p) && (q))) ||
15                ( current[1] && ((p) && (q))) ||
16                ( current[0] && ((p) && (q)));
17
18      next[1] = ( current[0] && (!(p) && (q)));
19
20      next[2] = ( current[2] && (!(p) && (q))) ||
21                ( current[1] && (!(p) && (q))) ||
22                ( current[2] && (!(p) && !(q)));
23
24      // Check if there were enabled transitions
25      bool not_stuck = false;
26      for (int i = 0; i < 3; i++) {
27        not_stuck = not_stuck || next[i];
28      }
29
30      if (! not_stuck) {
31        // None of the transitions were enabled
32
33  #ifdef MONITOR_REPORT_FAIL_IMMEDIATELY
34        SC_REPORT_WARNING("Property failed", "Critical error");
35        std::cout << "Property failed after " << num_steps
36                  << " steps" << std::endl;
37  #endif
38        status = MON_FAIL;
39      }
40    } // if (status == MON_UNDETERMINED)
41  } // step()
```

Listing 6.2: Illustrating `back_nondet` encoding of the automaton in Figure 6.2.

Deterministic encodings

Three novel deterministic encodings, which we call `front_det_switch`, `front_det_ifelse`, and `back_det`, expect that the automaton has been determinized using assignment-based encoding. Thus, these three encodings can be used only with $\mathcal{A}^{ass}_{DFW}(\psi)$. Note that we work with $\mathcal{A}^{ass}_{DFW}(\psi)$ directly and do not convert the automaton alphabet from integers back to Boolean functions. Instead, at the beginning of each step of the automaton we use the state of the MUV (i.e., the values

of all public and private variables, as exposed by the framework of [TVKS08]) to derive an assignment $\mathbf{a}$ to the atomic propositions in $AP(\psi)$. We then calculate an integer representing the relevant model state $mod\_st = I(\mathbf{a})$, where $\mathbf{a}$ is the current assignment, and use mod_st to drive the automaton transitions.

Referring to the running example automaton presented in Figure 6.2, we first show how to convert the Boolean expressions on the transitions to integers using assignment-based integer representation. Table 6.1 shows the integer encoding of all possible assignments of values to p and q. We then construct $\mathcal{A}^{ass}_{NFW}(\varphi)$ in Figure 6.3. Determinizing and minimizing $\mathcal{A}^{ass}_{NFW}(\varphi)$ using `BRICS Automaton` produces $\mathcal{A}^{ass}_{DFW}(\varphi)$, which in this case is identical to $\mathcal{A}^{ass}_{NFW}(\varphi)$.

p	q	**int**
0	0	0
0	1	1
1	0	2
1	1	3

Table 6.1 : Assignment-based encoding for the transitions of the $\mathcal{A}_{NFW}(\psi)$ in Figure 6.2

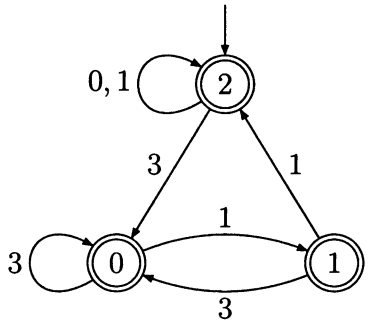


Figure 6.3 : $\mathcal{A}^{ass}_{NFW}(\varphi)$ for $\varphi = \mathbf{G}(p \rightarrow (q \vee \mathbf{X}q \vee \mathbf{XX}q))$. Determinizing $\mathcal{A}^{ass}_{NFW}(\varphi)$ using `BRICS Automaton` produces $\mathcal{A}^{ass}_{DFW}(\varphi)$, which is then minimized. The minimized $\mathcal{A}^{ass}_{DFW}(\varphi)$ in this case is identical to $\mathcal{A}^{ass}_{NFW}(\varphi)$.

The `back_det` encoding is similar to `back_nondet` in that it encodes the automaton transitions as a disjunction of the conditions that allow a state in `next[]` to be enabled. The difference is that here we use an integer instead of a vector to keep track of the (at most one) state that is reached in the current step of the au-

tomaton, and the transitions are driven by *mod_st* instead of by Boolean functions. See Listing 6.3 for an illustration of this encoding.

```
1  /**
2   * back_det encoding
3   */
4  test_monitor0::step() {
5    if (status == MON_UNDETERMINED) {
6      // Property has not been determined to fail so far
7      num_steps++; // Current length of execution trace
8
9      current = next;
10     next = -1;
11
12     // Calculate the system state index
13     int system_state_index = 0;
14     system_state_index += (p) ? (1 << 1) : 0;
15     system_state_index += (q) ? (1 << 0) : 0;
16
17       if (( ( current == 2) && ( system_state_index == 3)) ||
18         ( ( current == 1) && ( system_state_index == 3)) ||
19         ( ( current == 0) && ( system_state_index == 3)))
20           { next = 0;}
21       else if (( ( current == 0) && ( system_state_index == 1)))
22           { next = 1;}
23       else if (( ( current == 2) && ( system_state_index == 1))
             ||
24         ( ( current == 1) && ( system_state_index == 1)) ||
25         ( ( current == 2) && ( system_state_index == 0)))
26           { next = 2;}
27
28     // Check if there were enabled transitions
29     bool not_stuck = (next != -1);
30     if (! not_stuck) {
31       // None of the transitions were enabled
32
33 #ifdef MONITOR_REPORT_FAIL_IMMEDIATELY
34       SC_REPORT_WARNING("Property failed", "Critical error");
35       std::cout << "Property failed after " << num_steps
36                 << " steps" << std::endl;
37 #endif
38       status = MON_FAIL;
39     }
40   } // if (status == MON_UNDETERMINED)
```

```
41 } // step()
```

Listing 6.3: Illustrating back_det encoding of the automaton in Figure 6.3.

The front_det_switch and front_det_ifelse encodings are similar, but differ in the C++ constructs used to take advantage of the determinism in the automaton. Applying front_det_switch to the automaton in Figure 6.3 is illustrated in Listing 6.4 and front_det_ifelse is illustrated in Listing 6.5.

```
1  /**
2   * front_det_switch encoding
3   */
4  test_monitor0::step() {
5    if (status == MON_UNDETERMINED) {
6      // Property has not been determined to fail so far
7      num_steps++; // Current length of execution trace
8
9      current = next;
10     next = -1;
11
12     // Calculate the system state index
13     int system_state_index = 0;
14     system_state_index += (p) ? (1 << 1) : 0;
15     system_state_index += (q) ? (1 << 0) : 0;
16
17     switch (current) {
18       case 0:
19         switch ( system_state_index ) {
20           case 1: next = 1; break;
21           case 3: next = 0; break;
22         } // inner switch/case
23       break; // the outer case
24
25       case 1:
26         switch ( system_state_index ) {
27           case 3: next = 0; break;
28           case 1: next = 2; break;
29         } // inner switch/case
30       break; // the outer case
31
32       case 2:
33         switch ( system_state_index ) {
34           case 3: next = 0; break;
```

```
35            case 1: next = 2; break;
36            case 0: next = 2; break;
37          } // inner switch/case
38        break; // the outer switch/case
39      } // switch (current)
40
41      // Check if there were enabled transitions
42      bool not_stuck = (next != -1);
43      if (! not_stuck) {
44        // None of the transitions were enabled
45
46 #ifdef MONITOR_REPORT_FAIL_IMMEDIATELY
47        SC_REPORT_WARNING("Property failed", "Critical error");
48        std::cout << "Property failed after " << num_steps
49                  << " steps" << std::endl;
50 #endif
51        status = MON_FAIL;
52      }
53    } // if (status == MON_UNDETERMINED)
54 }
```

Listing 6.4: Illustrating `front_det_switch` encoding of the automaton in Figure 6.3.

```
1  /**
2   * front_det_ifelse encoding
3   */
4  test_monitor0::step() {
5    if (status == MON_UNDETERMINED) {
6      // Property has not been determined to fail so far
7      num_steps++; // Current length of execution trace
8
9      current = next;
10     next = -1;
11
12     // Calculate the system state index
13     int system_state_index = 0;
14     system_state_index += (p) ? (1 << 1) : 0;
15     system_state_index += (q) ? (1 << 0) : 0;
16
17     if (current == 0) {
18       if ( system_state_index == 1 ) { next = 1; }
19       else if ( system_state_index == 3 ) { next = 2; }
20     } // if (current == ...)
21
```

```
22      else if (current == 1) {
23        if ( system_state_index == 1 ) { next = 1; }
24        else if ( system_state_index == 3 ) { next = 2; }
25        else if ( system_state_index == 0 ) { next = 1; }
26      } // if (current == ...)
27
28      else if (current == 2) {
29        if ( system_state_index == 3 ) { next = 2; }
30        else if ( system_state_index == 1 ) { next = 0; }
31      } // if (current == ...)
32
33      // Check if there were enabled transitions
34      bool not_stuck = (next != -1);
35      if (! not_stuck) {
36        // None of the transitions were enabled
37
38  #ifdef MONITOR_REPORT_FAIL_IMMEDIATELY
39        SC_REPORT_WARNING("Property failed", "Critical error");
40        std::cout << "Property failed after " << num_steps
41                  << " steps" << std::endl;
42  #endif
43        status = MON_FAIL;
44      }
45    }
46  }
```

Listing 6.5: Illustrating `front_det_ifelse` encoding of the automaton in Figure 6.3.

6.4.5 Configuration space

The different options allow 27 possible combinations for generating a monitor, summarized in Table 6.2.

6.5 Experimental setup

6.5.1 SystemC model

Our experimental evaluation is based on the Adder model discussed in Chapter 4. As a reminder, the Adder implements a squaring function by using repeated incrementing

State Minimization	Alphabet Representation	Alphabet Minimization	Monitor Encoding
no	Not required	none partial full	`front_nondet`
yes	BDDs		`back_nondet`
	assignments		`front_nondet`
			`back_nondet`
			`front_det_ifelse`
			`front_det_switch`
			`back_det`

Table 6.2 : The configuration space for generating monitors.

by 1. We used the Adder to calculate 100^2 with 1,000 instances of a monitor for the same property. Since we are mostly concerned with monitor overhead, we focus on the time difference between executing the model with and without monitoring. We compiled the Adder model with a virgin installation of SystemC (i.e., without the monitoring framework presented in Chapter 4) and averaged the runtime over 10 executions. This established the baseline time.

To calculate the monitor overhead we averaged the runtime of each simulation over 10 executions and subtracted the baseline time. Notice that the overhead as calculated includes the cost of the monitoring framework and the slow-down due to all 1,000 monitors.

6.5.2 Properties

We used specifications constructed using both pattern formulas and randomly generated formulas. We used LTL formulas, as we have access to explicit-state LTL-to-automata translators (SPOT, in our case). The construction of NBWs is orthogonal to the issues we study here, as the framework is applicable to any specification lan-

guage that produces NBWs.

We adopted the pattern formulas used in [GH06] and presented below:

$$
\begin{aligned}
lu(n) &:= (\ldots(p_1 \mathbf{U} p_2))\ldots\mathbf{U}p_n)\mathbf{U}p_{n+1} \\
ru(n) &:= p_1 \mathbf{U}(p_2 \mathbf{U}(\ldots(p_n \mathbf{U} p_{n+1})\ldots)) \\
c1(n) &:= \bigvee_{i=1}^{n} \mathbf{GF}p_i \\
c2(n) &:= \bigwedge_{i=1}^{n} \mathbf{GF}p_i \\
qq(n) &:= \bigwedge_{i=1}^{n} (\mathbf{F}p_i \vee \mathbf{G}p_{i+1}) \\
rr(n) &:= \bigwedge_{i=1}^{n} (\mathbf{GF}p_i \vee \mathbf{FG}p_{i+1}) \\
ss(n) &:= \bigvee_{i=1}^{n} \mathbf{G}p_i
\end{aligned}
$$

In addition to these formulas we also used bounded **F** and bounded **G** formulas, and a new type of nested **U** formulas, presented below:

$$
\begin{aligned}
f1(n) &:= \mathbf{G}(p \rightarrow (q \vee \mathbf{X}q \vee \ldots \vee \mathbf{XX}\ldots\mathbf{X}q)) \\
f2(n) &:= \mathbf{G}(p \rightarrow (q \vee \mathbf{X}(q \vee \mathbf{X}(q \vee \ldots \vee \mathbf{X}q)\ldots))) \\
g1(n) &:= \mathbf{G}(p \rightarrow (q \wedge \mathbf{X}q \wedge \ldots \wedge \mathbf{XX}\ldots\mathbf{X}q)) \\
g2(n) &:= \mathbf{G}(p \rightarrow (q \wedge \mathbf{X}(q \wedge \mathbf{X}(q \wedge \ldots \wedge \mathbf{X}q)\ldots))) \\
uu(n) &:= \mathbf{G}(p_1 \rightarrow (p_1 \mathbf{U}(p_2 \wedge p_2 \mathbf{U}(p_3 \ldots (p_n \wedge p_n \mathbf{U} p_{n+1}))))\ldots)
\end{aligned}
$$

In our experiments we replaced the generic propositions p_i in each pattern formula with Boolean expressions of type `(a == 100^2 - 100( n - i - 1))`, where `a` is a variable representing the running total in the Adder. For each pattern we scaled up the formulas until all 27 configurations either timed out or crashed. Most configurations can be scaled up to $n = 7$, and the bounded properties $f1(n), f2(n), g1(n)$

and $g2(n)$ can be scaled to $n = 16$. We identified 127 pattern formulas for which at least one configuration could complete the monitoring task.

The random formulas that we used were generated following the framework of Daniele et al. [DGV99], using code provided by Kristin Y. Rozier. For each formula length there are two parameters that control the number of propositions used and the probability of selecting an **U** or a **V** operator (formula length is calculated by adding the number of atomic propositions, the number of logical connectives, and the number of temporal operators). We varied the number of atomic propositions between 1 and 5, the probability of selecting an **U** or a **V** was one of {0.1, 0.3, 0.7, 0.95}, and we varied the formula length from 5 to 30 in increments of 5. We used the same style of atomic propositions as in the pattern formulas. For each combination of parameters we generated 10 formulas at random, giving us a total of 1200 random formulas.

6.6 Experimental results

We ran all experiments on Ada, Rice's Cray XD1 compute cluster.[2] Each of Ada's nodes has two dual core 2.2 GHz AMD Opteron 275 CPUs and 8GB of RAM. We ran with exclusive access to a node so all 8GB of RAM were available for use. We allowed 8 hours (maximal job time on Ada) of computation time per configuration per formula for generating Büchi automata, automata-theoretic transformations, generating C++ code, compilation, linking with the Adder model using the monitoring framework presented in Chapter 4, and executing the monitored model 10 times.

We first evaluate the individual effect of each optimization. For each formula we partition the configuration space into two groups: those configurations that use the optimization and those that do not. We form the Cartesian product of the overhead times from both groups and present them on a scatter plot.

[2] `rcsg.rice.edu/ada`

6.6.1 State minimization

Fig. 6.4 shows the effect of determinization and state minimization on the automaton size. We observe that in most cases minimizing the automata (i.e., minimizing $\mathcal{A}^{ass}_{DFW}(\varphi)$ and $\mathcal{A}^{bdd}_{DFW}(\varphi)$) produces smaller automata than the equivalent $\mathcal{A}_{NFW}(\varphi)$. It is known [HU79] that in the worst case, nondeterministic automata are exponentially more succinct that the corresponding minimal deterministic automata. Our experimental results show that the worst case blow up is avoided for the types of formulas that are likely to be used in practice, and, in fact, for some formulas we see three orders of magnitude smaller deterministic automata. This observation goes against the traditional justification for constructing monitors from nondeterministic rather than deterministic automata.

In Fig. 6.5 we show the effect of state minimization on the runtime overhead. A few outliers notwithstanding, using state minimization lowers the runtime overhead of the monitor.

6.6.2 Alphabet representation

Figure 6.6 shows that using assignments leads to better performance than BDD-based alphabet representation. Our data shows that in the vast majority of cases, using assignments leads to smaller automata, which again suggests a connection between monitor size and monitor efficiency.

6.6.3 Alphabet minimization

Our data shows that partial– and full– alphabet minimization typically slow down the monitor (see Figure 6.7). We think that the reasons behind it are two-fold. On one hand, the performance of `gcc` as a decision engine to discover mutually exclusive conjunctions is not very good (in our experiments it was able to discover only 10%–15% of the possible mutually exclusive conjunctions). On the other hand, augmenting the formula increases the formula size, but SPOT does not take advantage of the extra

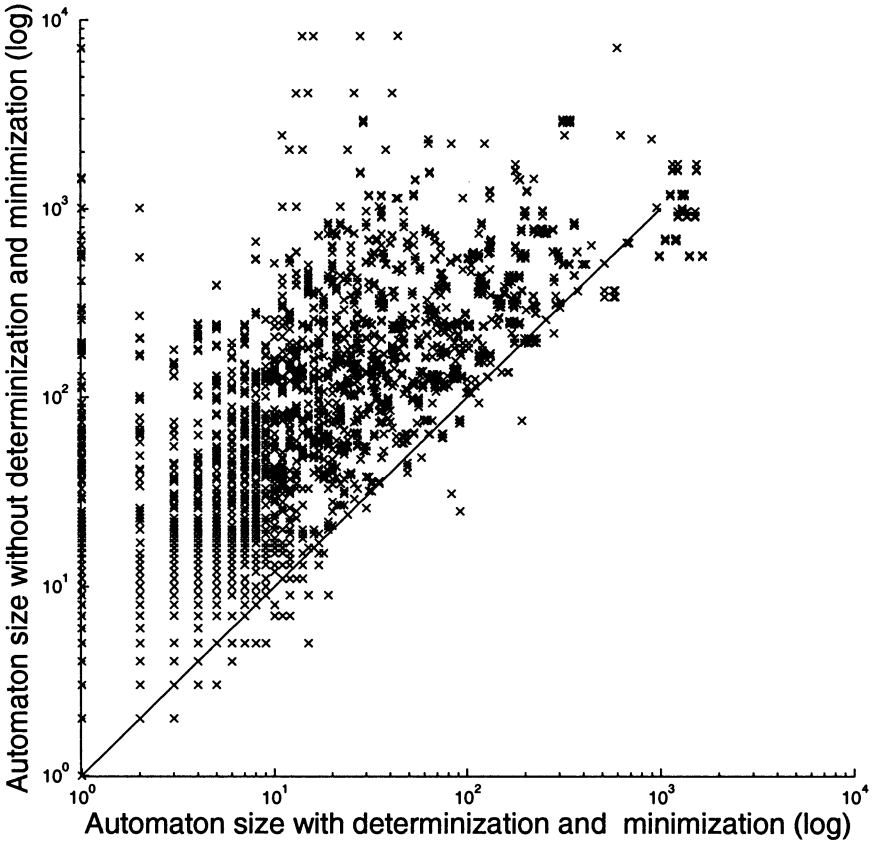


Figure 6.4 : Number of states in the automaton with and without state minimization.

information in the formula and typically generates bigger Büchi automata. If we manually augment the formula with *all* mutually exclusive conjunctions we do see smaller Büchi automata, so we believe this optimization warrants further investigation.

6.6.4 Monitor Encoding

Finally, we compared the effect of the different monitor encodings (Fig. 6.8). Our conclusion is that no encoding dominates the others, but two (`front_nondet` and `front_det_switch`) show the best performance relative to all others, while `back_det`

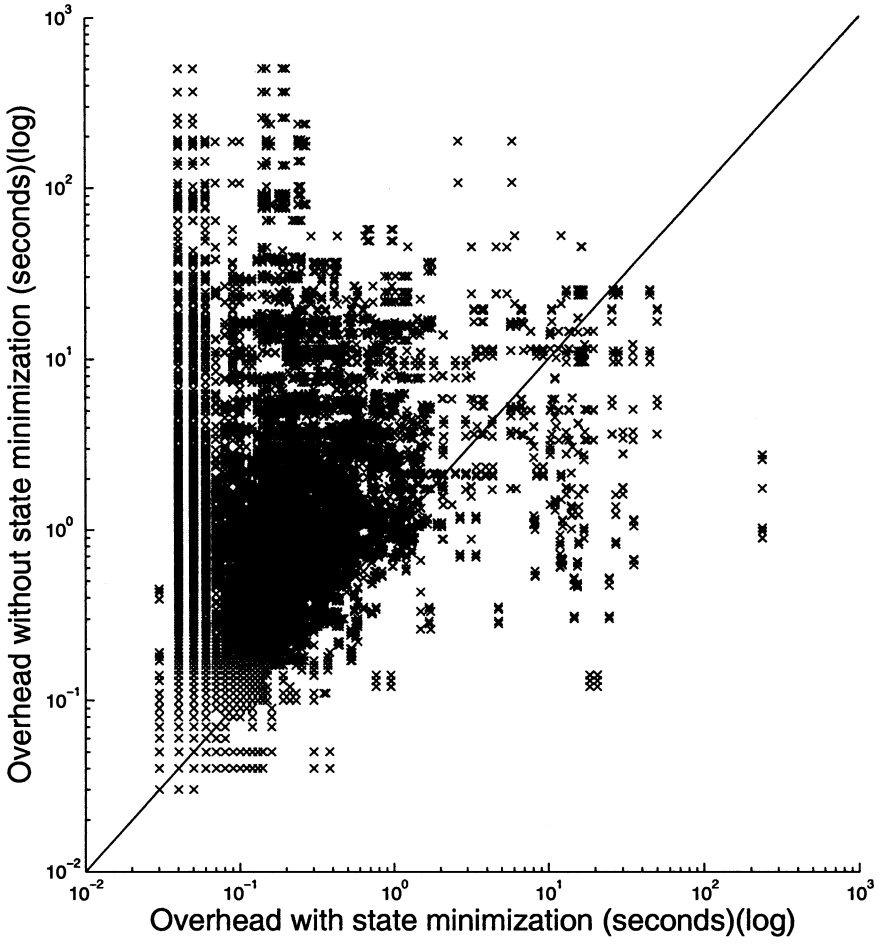


Figure 6.5 : The size of the determinized/minimized automaton in most cases is smaller than the size of the corresponding nondeterministic automaton.

has the worst performance. Comparing `front_nondet` and `front_det_switch` directly to each other (Fig. 6.9) indicates that `front_det_switch` delivers better performance for all but a few formulas.

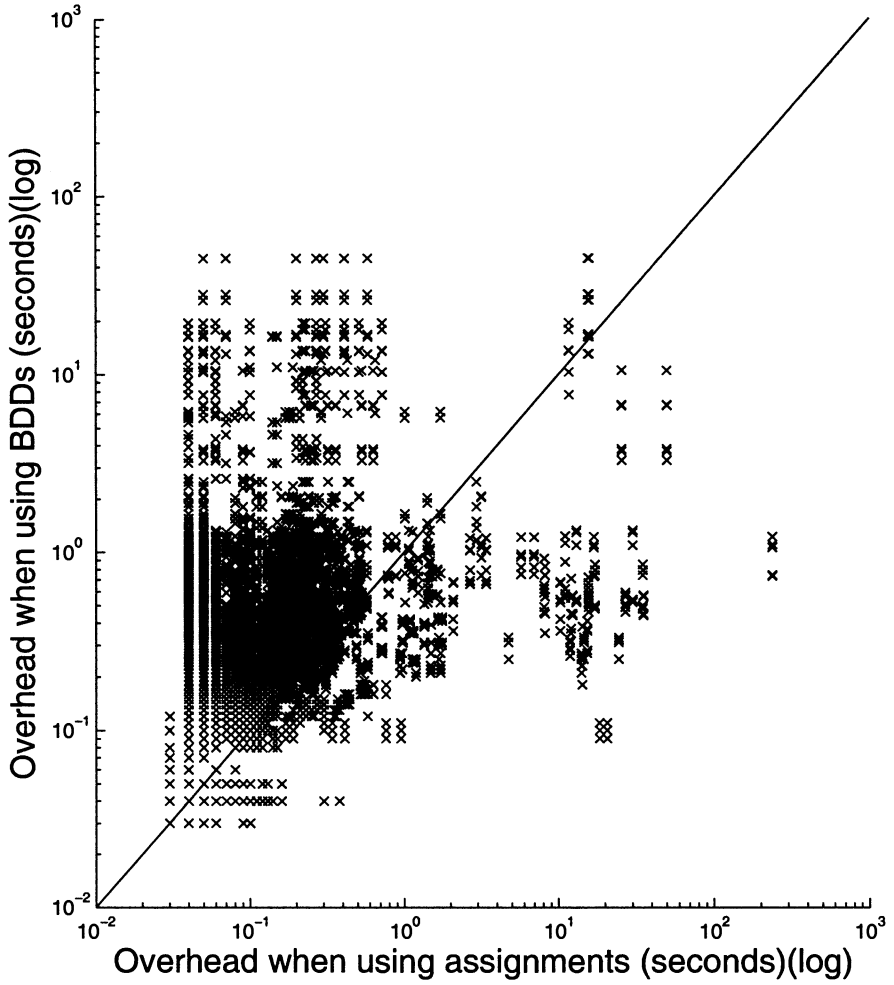


Figure 6.6 : Using assignments for alphabet representation leads to better performance than using BDDs.

6.6.5 Best configuration

The final check of our conclusion is presented in Figure 6.10, where we plot the performance of the winning configuration against all other configurations. There are a few outliers, but overall the configuration gives better performance than all others.

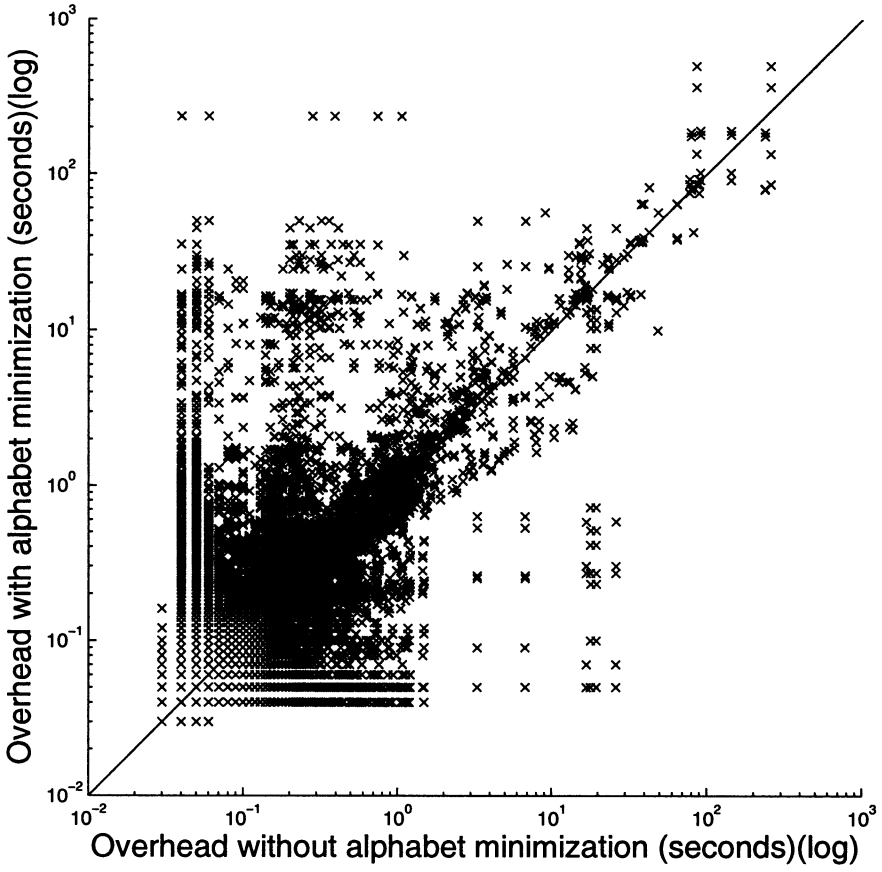


Figure 6.7 : Effect of alphabet minimization on monitor overhead. We do not see significant advantage in using alphabet minimization, but this may be due to the particular tool chain that we used.

Based on the comparison of individual optimizations we conclude that `front_det_switch` encoding with assignment–based state minimization and no alphabet minimization is the best overall configuration.

6.7 Summary and discussion

Together with the specification formalism proposed in [TVKS08], and the monitoring framework described in [TV10a], this work provides a general ABV solution for temporal monitoring of SystemC models. We have identified a configuration that generates low-overhead monitors and we believe that it can serve as a good default setting. We note, however, that practical use of our tool may involve monitoring tasks that are different than the synthetic load that we used for our tests. Recent developments in the area of self-tuning systems show that even highly optimized tools can be improved by orders of magnitude using search techniques over the configuration space (c.f., [Hoo08]). One possible extension of our work is to apply different optimizations to different types of formulas. For example, our data show that when the minimized automaton ($\mathcal{A}^{bdd}_{DFW}(\psi)$ or $\mathcal{A}^{ass}_{DFW}(\psi)$) has more states than the unminimized automaton ($\mathcal{A}_{NFW}(\psi)$), generating a monitor using $\mathcal{A}_{NFW}(\psi)$ leads to smaller runtime overhead. This observation can be used as a heuristic, and further investigation may reveal that for different classes of formulas different configurations yield the best results. Thus, we have left the user full control over the tool configuration.

The specification formalism proposed in Chapter 3, the monitoring framework described in Chapter 4, the techniques for automated instrumentation of the user code in Chapter 5, and the work presented in this chapter complete the necessary steps toward a general ABV solution for temporal monitoring of SystemC models. Chapter 7 summarizes the contributions of this dissertation and presents some possible extensions of this work.

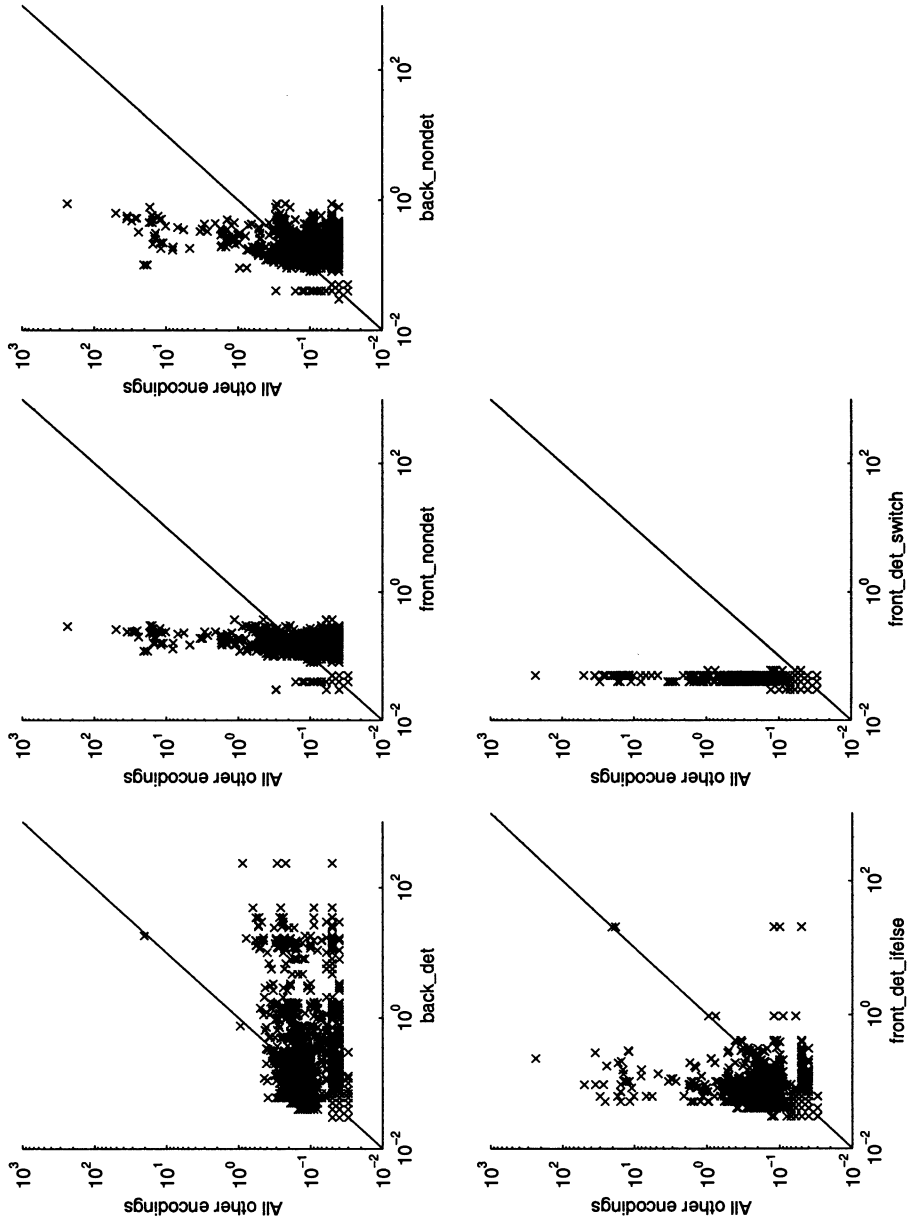


Figure 6.8 : Comparison of the monitor overhead when using different encodings. Each subplot shows the performance when using one of the encodings (x-axis) vs. all other encodings (y-axis).

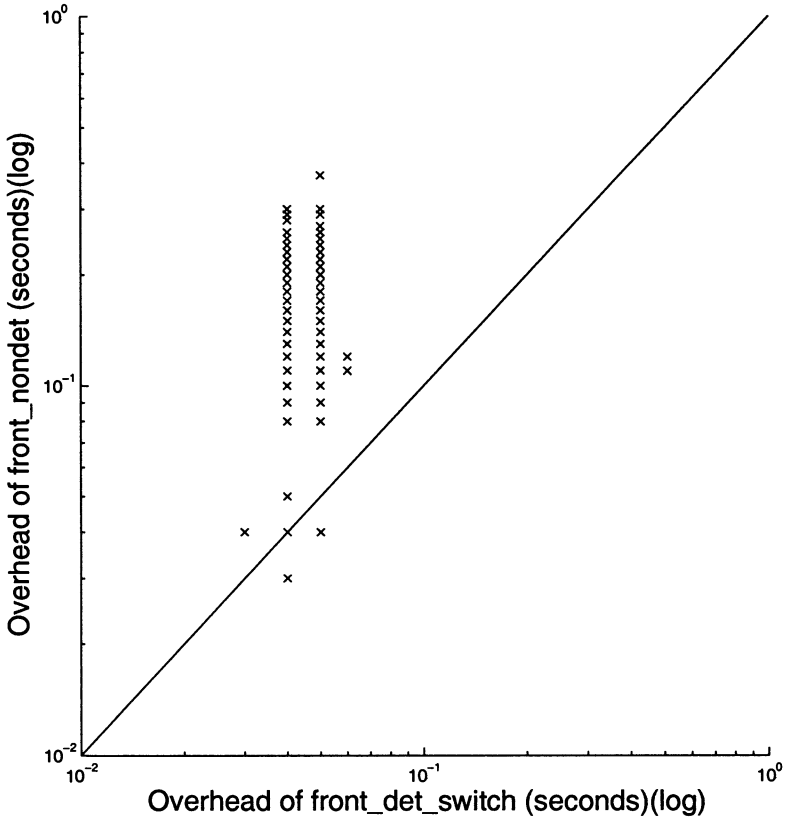

Overhead of front_nondet (seconds)(log)
Overhead of front_det_switch (seconds)(log)
10^0
10^{-1}
10^{-2}
10^{-2}
10^{-1}
10^0

Figure 6.9 :

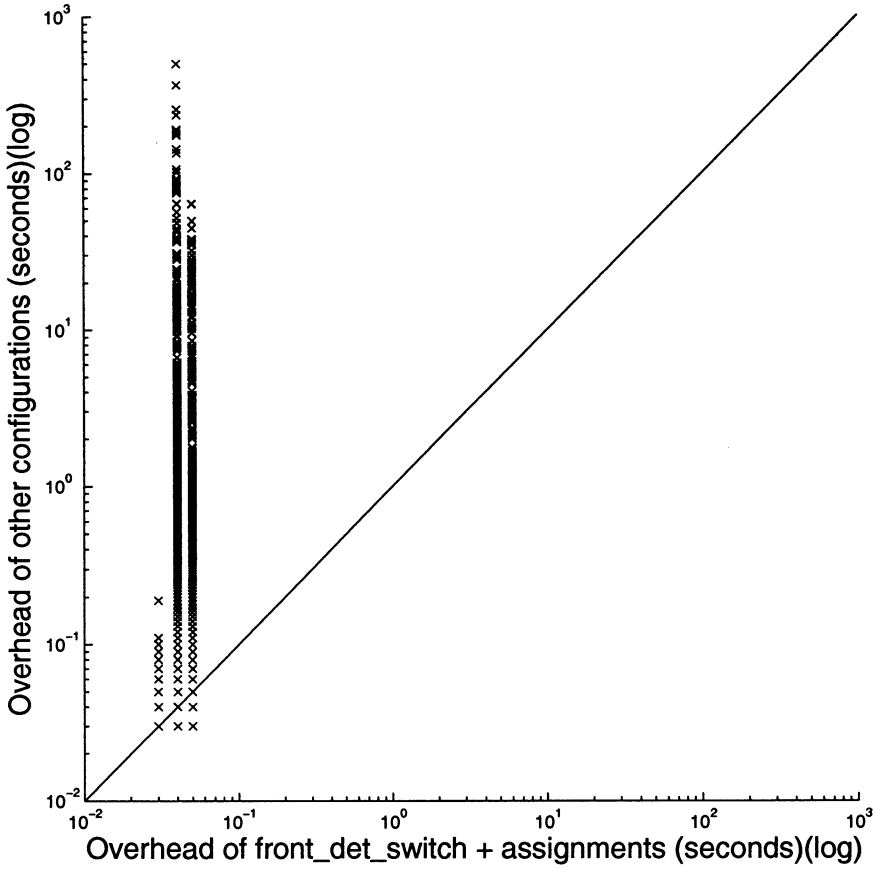


Figure 6.10 : Best overall performance

Chapter 7

Conclusion and Perspectives

7.1 Summary of contributions

The starting point of this dissertation is a detailed discussion of the simulation semantics of SystemC and of the types of properties that may need to be verified. The execution of a SystemC model is much more detailed than the execution of an RTL model, both in temporal resolution and in the amount of information that is accessible in the model. Existing specification languages fail to take advantage of this information, and, as a result, designers are limited in the way they can apply assertion-based verification to SystemC models. This dissertation identifies primitives that are essential to the execution of a SystemC model and elevates them to the Boolean layer of PSL-like languages. Specifically, the phases of the kernel and notification of events are exposed to the specification language, as well as the control flow, the call stack and the source code statements of the model. These primitives are derived from an abstraction of the SystemC simulation semantics, and are independent of the particular implementation of SystemC.

Simply allowing the specification to refer to these primitives is not sufficient by itself, because we also need a mechanism for determining if these primitives hold or not. This issue can be split in two parts: determining the values of specification primitives related to the execution of the kernel and notification of events, and determining the values of primitives related to the execution of user code. This dissertation describes a framework for exposing the actions of the kernel without prohibitively slowing down the simulation speed. This is done by adding an intermediary object that observes the actions of the kernel and also keeps track of the kernel phases (or event noti-

fications). When a sample point relevant to the execution of some set of monitors is reached, only those monitors are activated. The functionality required for storing and communicating with the monitors is incorporated in a few new objects that can be added to existing implementations of SystemC. Only a few lines of existing kernel code need to be modified to integrate the new functionality within the kernel. As a result, the operations of the kernel are exposed using low overhead and in a highly portable manner.

The second type of exposure required by the specification primitives is at the source-code level of the SystemC model. Exposing the execution of functions calls and their return is critical for specifying the behavior of SystemC models early in the design stage. Exposing the values of class and module variables to the monitoring framework without interfering with the data encapsulation provided by the object-oriented design allows using white-box verification transparently from the user and without requiring extra annotations. Exposing the syntax allows sampling the state of the user model at statement-level resolution. User-code exposure is achieved using low-overhead instrumentation of the code via automatically generated Aspect-Oriented Programming (AOP) advices.

The mechanisms for generating monitors for temporal properties present both theoretical as well as practical challenges. Dynamic verification can only detect failure of properties if such failure happens after a finite number of steps. This is done by detecting prefixes that cannot be extended to a correct execution trace. A very elegant construction by d'Amorim and Roşu allows the construction of a nondeterministic automaton on finite words that rejects precisely all bad prefixes for a particular temporal property. Building a deterministic monitor from a nondeterministic automaton, however, requires making several algorithmic choices. One point of concern is how to handle nondeterminism. This dissertation discusses two paths: explicit determinization of the automaton or using on-the-fly determinization in the monitor. Determinizing explicitly requires a way to represent as integers the Boolean formu-

las that guard the automaton transitions; this dissertation shows two mechanisms how to achieve this task. A new optimization that detects and removes inconsistent atomic propositions is also proposed. Five different options for encoding automata as C++ monitors are described. All combinations of these algorithmic choices are tested empirically on a wide variety of temporal formulas, and one configuration is distinguished as having the best overall performance. It should also be pointed out that the monitor generation techniques discussed here are applicable not only to SystemC, but also to any verification framework that requires constructing monitors in software.

This dissertation has made it possible to specify SystemC properties at fine-grained temporal resolution, has proposed a low-overhead exposure of kernel state and user-code state, has automated the instrumentation of user code and has demonstrated techniques for generating low-overhead monitors automatically. These contributions form a complete and efficient dynamic assertion-based verification framework for SystemC.

7.2 Adopting the framework

Adopting the framework presented in this dissertation requires very little effort. This section summarizes the necessary steps.

7.2.1 Adopting the new specification primitives

The framework presented in this dissertation exposes the operations of the SystemC kernel and the user code in accordance with the specification primitives presented in Chapter 3. One way one can take advantage of this framework is by incorporated our proposed Boolean primitives in a specification language that has a temporal layer, such as LTL, PSL, or SVA. Monitors can then be generated automatically from properties. If the specification language supports clock expressions, the primitives can be used further to control the temporal resolution of the execution trace. Another

way the user can use the framework is by constructing monitors manually and simply using the provided exposure of the operations of the kernel and the user code.

7.2.2 Exposing the operations of the SystemC kernel

Adopting the monitoring framework presented in Chapter 4 requires adding the source code of `mon_observer` and `mon_prototype` to the SystemC kernel source code. In addition, the SystemC scheduler needs to be augmented with fewer than twenty lines of code to make possible the communication between the scheduler and the `observer`. In the reference implementation of SystemC provided by OSCI, the source code files that will need to be modified are `sysc/kernel/sc_simcontext.cpp` and `sysc/kernel/sc_event.cpp`. After that, the kernel will need to be recompiled and thereafter the new functionality described in this dissertation will be available to the user. A patch that contains the new source code files and can apply the required changes to the existing source code is available from `http://www.cs.rice.edu/CS/Verification/`

7.2.3 Exposing user code primitives

The approach presented in Chapter 5 uses an AOP tool called `AspectC++` to instrument automatically the user code. At the time of writing of this dissertation, this tool is available for download for free from `http://www.aspectc.org/`. The approach presented in this dissertation involves generating AOP advices from user-provided declarations of primitives. We implemented a tool that parses the declarations and generates the corresponding advices; the tool is available for free online at `http://www.cs.rice.edu/CS/Verification/`.

7.2.4 Generating efficient monitors from properties

The work presented in Chapter 6 makes use of an LTL-to-automata tool called SPOT. At the time of writing of this dissertation, SPOT is available for free from `http:`

//spot.lip6.fr/wiki/. Alphabet minimization is done using the gcc compiler, available for free from http://gcc.gnu.org/. Determinization and minimization of nondeterministic automata on finite words was done using a tool called BRICS Automaton, available for free from http://www.brics.dk/automaton/. We wrote a tool we call monitor_master that integrates with SPOT to create the nondeterministic automaton from which the monitor is built. We used the LBT format described at http://www.tcs.hut.fi/Software/maria/tools/lbt/ as the output format. We extended BRICS Automaton to parse automata in LBT format, perform determinization and minimization, and output the resulting automaton in LBT format. We built into monitor_master the required functionality to send the automaton to BRICS Automaton and parse the minimized automaton without leaving monitor_master. The tool then synthesizes a C++ monitor according to the options selected by the user. monitor_master is available for free from http://www.cs.rice.edu/CS/Verification/.

7.3 Future directions

One possible extension of the work presented in this dissertation is on specifying the power requirements of circuits. Recent work by Liu et al. [LTSA10] shows how SystemC can be used for high-level power modeling. Dhanwada et al. [DLN05] augment SystemC transaction-level models to perform transaction-level power estimation. These and other existing techniques would benefit from a mechanism to specify formally the power requirements of the system and to monitor the power consumption estimation during simulation. Such specification can be done at different architectural resolutions (e.g., at the system level or at level of individual modules), as well as at different temporal resolution. A future direction of research is adapting th techniques presented in this dissertation to monitor such specifications.

Another possible extension is in monitoring analog and mixed signals in SystemC models. OSCI announced recently an integration of analog and mixed signals

(AMS) in SystemC [GBVE08]. The new extension provides functional modeling, architectural exploration, virtual prototyping, and integration validation for "embedded analog/mixed-signal systems." In order to maintain an acceptable simulation performance while modeling the architecture's behavior with sufficient accuracy, the AMS framework requires using dedicated simulation kernels synchronized with the standard SystemC kernel [GBVE08]. Thus, integrating the monitoring framework presented in this dissertation with the new AMS simulation framework would require establishing a new abstraction of the interplay between the different AMS simulation kernels, as well as defining a new notion of an execution trace.

Appendix A

Source code of the Adder model

```
1  #ifndef ADDER_H
2  #define ADDER_H
3
4  #include "global.h"
5
6  #ifdef WITH_OBSERVER
7  extern mon_observer* observer;
8  #endif
9
10
11 class adder : public sc_module {
12
13   SC_HAS_PROCESS(adder);
14
15  public:
16
17   // The constructor
18   adder(sc_module_name module_name, int num_mons);
19
20   sc_in < int > input1, input2;
21   sc_out< int > result;
22
23   void driver();
24   void do_add1();
25
26  protected:
27
28   int _a;
29   sc_event addition_event;
30   sc_event driver_event;
31   sc_event add1_activate_event;
32 }; //class
33
```

```
34 #endif
```

Listing A.1: Source code of `adder.h`

```
1  #include "adder.h"
2
3
4  // The constructor
5  adder::adder(sc_module_name module_name, int num_mons) :
6    sc_module(module_name) {
7
8    SC_THREAD( driver );
9    sensitive << input1 << input2;
10
11   _a = 0;
12 }
13
14
15 void
16 adder::driver() {
17   DP->dprint(4, "driver(): Start execution\n");
18
19   while(true) {
20     DP->dprint(5, "driver(): Suspending until the values on the
         inputs change\n");
21     wait(input1.value_changed_event() | input2.
         value_changed_event());
22
23     DP->dprint(5, "driver(): The values on the inputs have
         changed\n");
24     DP->dprint(5, "driver(): Reading from the input ports\n");
25     int addent1 = input1.read();
26     int addent2 = input2.read();
27
28     DP->dprint(5, "driver(): input1 = %d, input2 = %d\n", addent1
         , addent2);
29     DP->dprint(5, "driver(): Initialize the local memory to %d
         and spawn %d processes\n", addent1, addent2);
30     _a = addent1;
31
32     for (int i=0; i < addent2; i++) {
33       DP->dprint(5, "driver(): Spawning an add1 process (seq %d)\
           n", i+1);
34       sc_spawn( sc_bind(&adder::do_add1, this) );
35     }
```

```
36
37      // Allow the do_add1() processes to initialize
38      // by suspending until the next delta-cycle
39      DP->dprint(5, "driver(): Notifying the driver event (
           SC_ZERO_TIME delay)\n");
40      driver_event.notify(SC_ZERO_TIME);
41
42      DP->dprint(5, "driver(): Suspending until the driver event is
            notified\n");
43      wait(driver_event);
44
45      DP->dprint(5, "driver(): Resume execution after the driver
           event is notified\n");
46
47      DP->dprint(5, "driver(): notifying the addition event (
           immediate notification)\n");
48      add1_activate_event.notify();
49
50      // Suspend for a delta-cycle to allow all
51      // computations to complete
52      DP->dprint(5, "driver(): Notifying the driver event (
           SC_ZERO_TIME delay)\n");
53      driver_event.notify(SC_ZERO_TIME);
54
55      DP->dprint(5, "driver(): Suspending until the driver event is
            notified\n");
56      wait(driver_event);
57
58      DP->dprint(5, "driver(): Resume execution after the driver
           event is notified\n");
59
60      DP->dprint(5, "driver(): Writing the result (%d) to the \"
           result\" port\n", _a);
61      result.write(_a);
62    }
63
64    //Unreachable
65    DP->dprint(4, "driver(): Finished execution. The value of the
         addent is %d\n", _a);
66    assert(false);
67  }
68
69
70  void
71  adder::do_add1() {
```

```
72    DP->dprint(4, "do_add1(): Starting execution. The value of
         addent is %d\n", _a);
73    DP->dprint(6, "do_add1(): Suspending until the waiting event is
          notified\n");
74
75    wait(add1_activate_event);
76
77    DP->dprint(6, "do_add1(): Resuming execution after waiting
         event was notified\n");
78
79    (_a) = (_a) + 1;
80
81    DP->dprint(7, "do_add1(): The value of the addent was
         incremented\n");
82
83    DP->dprint(6, "do_add1(): notifying the addition event\n");
84    addition_event.notify(); //immediate notification
85
86    DP->dprint(4, "do_add1(): finishing execution. The value of
         addent is %d\n", _a);
87  }
```

Listing A.2: Source code of `adder.cc`

```
1  #ifndef DRIVER_H
2  #define DRIVER_H
3
4  #include "global.h"
5
6
7  class driver:public sc_module {
8
9    SC_HAS_PROCESS(driver);
10
11  public:
12
13   // The constructor
14   driver(sc_module_name module_name, int input);
15
16   // Input and output ports
17   sc_out <int> output1, output2;
18   sc_in  <int> result;
19
20   sc_in <bool> clk;
21
```

```
22  protected:
23    // Local memory
24    int initial_number;
25    int running_total;
26
27    void generate_task();
28    void get_result();
29  };
30
31
32  #endif
```

Listing A.3: Source code of `driver.h`

```
1   #include "driver.h"
2
3   /**
4    * The constructor
5    */
6   driver::driver(sc_module_name module_name, int input):sc_module(
       module_name) {
7     initial_number = input;
8     running_total = 0;
9
10    // Initialize the output ports to guarantee deterministic
         simulation
11    output1.initialize(0);
12    output1.initialize(0);
13
14    // This process will not be initialized
15    SC_THREAD(generate_task);
16    sensitive << clk.pos();
17    dont_initialize();
18
19    // Notice that this process will not be initialized
20    SC_METHOD(get_result);
21    sensitive << result;
22    dont_initialize();
23  }
24
25
26  /**
27   * Generates a new problem for the connected arithmetic unit
28   */
29  void
```

```
30 driver::generate_task() {
31   DP->dprint(4, "generate_task(): started execution\n");
32
33   for (int i = 0; i < initial_number; i++) {
34
35     int to_output1 = running_total;
36     int to_output2 = initial_number;
37
38     DP->dprint(6, "generate_task(): writing %d to port \"output1
          \"\n", to_output1);
39     output1.write(to_output1);
40
41     DP->dprint(6, "generate_task(): writing %d to port \"output2
          \"\n", to_output2);
42     output2.write(to_output2);
43
44     DP->dprint(5, "generate_task(): suspending until the next
          clock tick\n");
45     wait();
46
47     DP->dprint(5, "generate_task(): resuming after return from \"
          wait()\"\n");
48   }
49
50
51   DP->dprint(4, "generate_task(): finished execution\n");
52
53   DP->dprint(2, "generate_task(): %d ^ 2 = %d\n", initial_number
       , running_total);
54   sc_stop();
55
56 }
57
58
59 /**
60  * Checks the result of the operation
61  */
62 void
63 driver::get_result() {
64   DP->dprint(4, "check_result(): started execution\n");
65
66   DP->dprint(5, "check_result(): reading from port
67      \"result\"\n");
68
69   running_total = result.read();
```

```
70
71    DP->dprint(5, "check_result(): the value of the port is %d\n",
          running_total);
72
73    DP->dprint(4, "check_result(): finished execution\n");
74  }
```

Listing A.4: Source code of `driver.cc`

```
1  #ifndef GLOBAL_H
2  #define GLOBAL_H
3
4  #include "debug_printer.h"
5  #include <systemc.h>
6
7  #endif
```

Listing A.5: Source code of `global.h`

Bibliography

[ABG+00] Y. Abarbanel, I. Beer, L. Gluhovsky, S. Keidar, and Y. Wolfsthal. Focs: Automatic generation of simulation checkers from formal specifications. In *CAV'00: Proc. of the 12th International Conference on Computer Aided Verification*, pages 538–542, 2000.

[AFF+02] R. Armoni, L. Fix, A. Flaisher, R. Gerth, B. Ginsburg, T. Kanza, A. Landver, S. Mador-Haim, E. Singerman, A. Tiemeyer, M.Y. Vardi, and Y. Zbar. The ForSpec temporal logic: A new temporal property-specification logic. In *Proc. 8th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, volume 2280 of *LNCS*, pages 296–211, Grenoble, France, April 2002. Springer-Verlag.

[AKT+06] R. Armoni, D. Korchemny, A. Tiemeyer, M.Y. Vardi, and Y. Zbar. Deterministic dynamic monitors for linear-time assertions. In *Proc. Workshop on Formal Approaches to Testing and Runtime Verification*, volume 4262 of *Lecture Notes in Computer Science*. Springer, 2006.

[BBL98] I. Beer, S. Ben-David, and A. Landver. On-the-fly model checking of RCTL formulas. In *Computer Aided Verification, Proc. 10th International Conference*, volume 1427 of *LNCS*, pages 184–194. Springer-Verlag, 1998.

[BCC+05] L. Burdy, Y. Cheon, D. Cok, M. D. Ernst, J. Kiniry, G. T. Leavens, K. R. M. Leino, and E. Poll. An overview of JML tools and applications. *Software Tools for Technology Transfer*, 7(3):212–232, June

2005.

[BCH+04] D. Beyer, A. J. Chlipala, T. A. Henzinger, R. Jhala, and R. Majumdar. The BLAST query language for software verification. In *SAS'04: Static Analysis, 11th International Symposium*, pages 2–18, 2004.

[BD05] D. C. Black and J. Donovan. *SystemC: From the Ground Up.* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005.

[Ber03] J. Bergeron. *Writing Testbenches: Functional Verification of HDL Models, Second Edition.* Springer, February 2003.

[BF03] B. F. Bass and H. D. Foster. System and method for specifying hardware description language assertions targeting a diverse set of verification tools. Patent, 07 2003. US 6591403.

[BFH05] D. Bustan, D. Fisman, and J. Havlicek. Automata construction for PSL. Technical report, The Weizmann Institute of Science, 2005.

[BGM04] A. Bunker, G. Gopalakrishnan, and S. A. McKee. Formal Hardware Specification Languages for Protocol Compliance Verification. *ACM Transactions on Design Autom. of Elec. Sys.*, 9(1):1–32, January 2004.

[BGR02] A. Braun, J. Gerlach, and W. Rosenstiel. Checking temporal properties in SystemC specifications. *High-Level Design Validation and Test Workshop, 2002. 7th IEEE International*, pages 23–27, Oct. 2002.

[BK10] N. Blanc and D. Kroening. Race analysis for systemc using model checking. *ACM Trans. Des. Autom. Electron. Syst.*, 15(3):1–32, 2010.

[BLS06] A. Bauer, M. Leucker, and C. Schallhart. Monitoring of real-time properties. In *FSTTCS'06: Foundations of Software Technology and Theoretical Computer Science, 26th International Conference, volume 4337 of LNCS*, pages 260–272. Springer, 2006.

[Bod05] E. Bodden. Efficient and expressive runtime verification for Java. In *Grand Finals of the ACM Student Research Competition 2005*, 2005. Winner paper of the Grand Finals.

[BR02] T. Ball and S. Rajamani. SLIC: A specification language for interface checking. Technical report, Microsoft Research, January 2002.

[Bry86] R.E. Bryant. Graph-based algorithms for Boolean-function manipulation. *IEEE Trans. on Computers*, C-35(8), 1986.

[Bry92] R.E. Bryant. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys*, 24(3):293–318, 1992.

[Büc62] J.R. Büchi. On a decision method in restricted second order arithmetic. In *Proc. Internat. Congr. Logic, Method. and Philos. Sci. 1960*, pages 1–12, Stanford, 1962. Stanford University Press.

[BZ08] M. Boulé and Z. Zilic. *Generating Hardware Assertion Checkers.* Springer Publishing Company, Incorporated, 2008.

[CA02] L. Charest and E. M. Aboulhamid. A VHDL/SystemC comparison in handling design reuse. In *Proceedings of 2002 International Workshop on System-on-Chip for Real-Time Applications*, pages 79–85, Banff, Canada, July 2002.

[CCH+99] H. Chang, L. Cooke, M. Hunt, G. Martin, A. J. McNelly, and L. Todd. *Surviving the SOC revolution: a guide to platform-based design.* Kluwer Academic Publishers, Norwell, MA, USA, 1999.

[CDHR02] James C. Corbett, Matthew B. Dwyer, John Hatcliff, and Robby. Expressing checkable properties of dynamic systems: the Bandera Specification Language. *International Journal on Software Tools for Technology Transfer (STTT)*, 4(1):34–56, 2002.

[CE81] E.M. Clarke and E.A. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Proc. Workshop on Logic of Programs*, volume 131 of *LNCS*, pages 52–71. Springer-Verlag, 1981.

[CH07] H. B. Carter and S. G. Hemmady. *Metric Driven Design Verification.* Springer Publishing Company, Incorporated, 2007.

[Chu57] Alonzo Church. Application of recursive arithmetic to the problem of circuit synthesis,. In *Summaries of talks presented at the Summer Institute for Symbolic Logic, Cornell University 1957*, pages 3–50, Princeton, 1957. Institute for Defense Analyses.

[CR07] F. Chen and G. Roşu. MOP: an efficient and generic runtime verification framework. In *OOPSLA '07: Object-Oriented Programming, Systems, Languages and Applications*, pages 569–588, New York, NY, USA, 2007. ACM.

[CT04] K. D. Cooper and L. Torczon. *Engineering a Compiler.* Morgan Kaufmann, 2004.

[CVWY92] C. Courcoubetis, M.Y. Vardi, P. Wolper, and M. Yannakakis. Memory efficient algorithms for the verification of temporal properties. *Formal Methods in System Design*, 1:275–288, 1992.

[DG02] R. Drechsler and D. Große. Reachability analysis for formal verification of SystemC. In *Euromicro Symposium on Digital Systems Design*, pages 337–340, 2002.

[DGV99] M. Daniele, F. Giunchiglia, and Moshe Y. Vardi. Improved automata generation for linear temporal logic. In *CAV '99: Proc. 11th Int. Conf. on Computer Aided Verification*, pages 249–260, London, UK, 1999. Springer-Verlag.

[dH05] M. d'Amorim and K. Havelund. Event-based runtime verification of java programs. In *WODA '05: Proceedings of the third international workshop on Dynamic analysis*, pages 1–7, New York, NY, USA, 2005. ACM.

[DLN05] N. Dhanwada, I. Lin, and V. Narayanan. A power estimation methodology for systemc transaction level models. In *CODES+ISSS '05: Proceedings of the 3rd IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pages 142–147, New York, NY, USA, 2005. ACM.

[DLP04] A. Duret-Lutz and D. Poitrenaud. SPOT: An extensible model checking library using transition-based generalized Büchi automata. *Modeling, Analysis, and Simulation of Computer Systems*, 0:76–83, 2004.

[DM06] D. Déharbe and S. Medeiros. Aspect-oriented design in SystemC: implementation and applications. In *SBCCI '06: Proceedings of the 19th annual symposium on Integrated circuits and systems design*, pages 119–124, New York, NY, USA, 2006. ACM.

[dMB08] L. Mendonça de Moura and N. Bjørner. Z3: An efficient SMT solver. In *TACAS'08: Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference*, pages 337–340, 2008.

[dR05] M. d'Amorim and G. Roşu. Efficient monitoring of ω-languages. In *Proc. 17th International Conference on Computer Aided Verification*, pages 364–378, 2005.

[EES+05] W Ecker, V Esen, T Steininger, M Velten, and J Smit. Implementation of a SystemC assertion library, 2005.

[EES+06] W. Ecker, V. Esen, T. Steininger, M. Velten, and M. Hull. Specification

language for Transaction Level Assertions. *HLDVT'06: IEEE International High-Level Design, Validation, and Test Workshop*, pages 77–84, 2006.

[EF06] C. Eisner and D. Fisman. *A Practical Introduction to PSL.* Springer, New York, Inc., Secaucus, NJ, USA, 2006.

[EFB01] T. Elrad, R. E. Filman, and A. Bader. Aspect-oriented programming: Introduction. *Commun. ACM*, 44(10):29–32, 2001.

[EIIK08] Y. Endoh, T. Imai, M. Iwamasa, and Y. Kataoka. A pointcut-based assertion for high-level hardware design. In *ACP4IS '08: Proc. AOSD workshop on Aspects, components, and patterns for infrastructure software*, pages 1–6, New York, NY, USA, 2008. ACM.

[FKL03] H. Foster, A. Krolnik, and D. Lacey. *Assertion-Based Design.* Kluwer Academic Publishers, Norwell, MA, USA, 2003.

[Fos08] H. Foster. Assertion-based verification: Industry myths to realities (invited tutorial). In *CAV '08: Proceedings of the 20th international conference on Computer Aided Verification*, pages 5–10, Berlin, Heidelberg, 2008. Springer-Verlag.

[FS04] B. Finkbeiner and H. Sipma. Checking finite traces using alternating automata. *Form. Methods Syst. Des.*, 24(2):101–127, 2004.

[GBA+99] D. Geist, G. Biran, T. Arons, M. Slavkin, Y. Nustov, M. Farkas, K. Holtz, A. Long, D. King, and S. Barret. A methodology for the verification of a "system on chip". In *DAC '99, Proc. 36th Design Automation Conference*, pages 574–579, New York, NY, 1999. ACM.

[GBVE08] C. Grimm, M. Barnasconi, A. Vachoux, and K. Einwich. An introduction to modeling embedded analog/mixed-signal systems using systemc

ams extensions. White Paper, 2008. Available online.

[GD03] D. Große and R. Drechsler. Formal verification of LTL formulas for SystemC designs. In *ISCAS (5)*, pages 245–248, 2003.

[Gei01] M. Geilen. On the construction of monitors for temporal logic properties. *Electr. Notes Theor. Comput. Sci.*, 55(2), 2001.

[GH01] D. Giannakopoulou and K. Havelund. Automata-based verification of temporal properties on running programs. In *Int. conf. on Automated Software Engineering*, page 412, Washington, DC, USA, 2001. IEEE.

[GH06] J. Geldenhuys and H. Hansen. Larger automata and less work for LTL model checking. In *In Model Checking Software, 13th Int. SPIN Workshop, volume 3925 of LNCS*, pages 53–70. Springer, 2006.

[Ghe06] F. Ghenassia. *Transaction-Level Modeling with Systemc: TLM Concepts and Applications for Embedded Systems.* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[GHT04] A. Gawanmeh, A. Habibi, and S. Tahar. Enabling SystemC verification using Abstract State Machines. In *Proc. Languages for Formal Specification and Verification, Forum on Specification and Design Languages*, 2004.

[GLMS02] T. Grotker, S. Liao, G. Martin, and S. Swan. *System Design with SystemC.* Kluwer Academic Publishers, Norwell, MA, USA, 2002.

[Goe05] R. Goering. A call to action for the EDA industry. EETimes, June 2005. Available online.

[GPVW95] R. Gerth, D. Peled, M.Y. Vardi, and P. Wolper. Simple on-the-fly automatic verification of Linear Temporal Logic. In P. Dembiski and

M. Sredniawa, editors, *Protocol Specification, Testing, and Verification*, pages 3–18. Chapman & Hall, August 1995.

[GSPS01] A. Gal, W. Schrder-Preikschat, and O. Spinczyk. AspectC++: Language proposal and prototype implementation. In *OOPSLA'01: Object-Oriented Programming, Systems, Languages and Applications*, 2001.

[HGT04] A. Habibi, A. Gawanmeh, and S. Tahar. Assertion based verification of PSL for SystemC designs. In *International Symp. on System-on-Chip*, pages 177–180, 2004.

[HJ08] K. W. Hamlen and M. Jones. Aspect-oriented in-lined reference monitors. In *PLAS '08: Proceedings of the third ACM SIGPLAN workshop on Programming languages and analysis for security*, pages 11–20, New York, NY, USA, 2008. ACM.

[HJMS03] T. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Software verification with BLAST. In *Tenth International Workshop on Model Checking of Software (SPIN)*, volume LNCS 2648, 2003.

[HMMCM06] C. Helmstetter, F. Maraninchi, L. Maillet-Contoz, and M. Moy. Automatic generation of schedulings for improving the test coverage of Systems-on-a-Chip. In *FMCAD '06: Proceedings of the Formal Methods in Computer Aided Design*, pages 171–178, Washington, DC, USA, 2006. IEEE Computer Society.

[Hoo08] H. H. Hoos. Computer-aided design of high-performance algorithms. Technical report, University of British Columbia, 2008.

[HT04] A. Habibi and S. Tahar. On the extension of SystemC by SystemVerilog assertions. *Electrical and Computer Engineering, 2004. Canadian Conf. on*, 4:1869–1872 Vol.4, 2-5 May 2004.

[HU79] J.E. Hopcroft and J.D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley, 1979.

[IEE06] *IEEE Std 1666 - 2005 IEEE Standard SystemC Language Reference Manual*, 2006.

[IS03] C. Ip and S. Swan. A tutorial introduction on the new SystemC verification standard. White paper, 2003. Available online (6 pages).

[KEP06] D. Karlsson, P. Eles, and Z. Peng. Formal verification of SystemC designs using a Petri-net based representation. In *DATE '06: Proceedings of the conf. on Design, automation and test in Europe*, pages 1228–1233, 3001 Leuven, Belgium, Belgium, 2006. European Design and Automation Association.

[KHT04] A. Kasuya, E. Hawk, and T. Tesfaye. Verification applications of aspect-oriented-programming (AOP). In *DvCON'04: Design and Verification Conference*, 2004.

[KIL+97] G. Kiczales, J. Irwin, J. Lamping, J. Loingtier, C. V. Lopes, C. Maeda, and A. Mendhekar. Aspect-oriented programming. In *ECOOP'97: European Conference on Object-Oriented Programming*, pages 220–242, 1997.

[KL06] O. Kupferman and R. Lampert. On the construction of fine automata for safety properties. In *ATVA'06: Proc. of the International Symposium on Automated Technology for Verification and Analysis*, pages 110–124, 2006.

[KS05] D. Kroening and N. Sharygina. Formal verification of SystemC by automatic hardware/software partitioning. In *MEMOCODE'05: 3rd ACM/IEEE International Conference on Formal Methods and Models for Codesign*, pages 101–110, 2005.

[KT07] A. Kasuya and T. Tesfaye. Verification methodologies in a TLM-to-RTL design flow. In *DAC'07: Proc. 44th Design Automation Conference*, pages 199–204, 2007.

[KTZ06] A. Kasuya, T. Tesfaye, and E. Zhang. Native SystemC Assertion mechanism with transaction and temporal assertion support. In *EDA Tech Forum (Available Online)*, September 2006.

[KV01] O. Kupferman and M.Y. Vardi. Model checking of safety properties. *Formal methods in System Design*, 19(3):291–314, November 2001.

[Lam05] W. K. Lam. *Hardware Design Verification: Simulation and Formal Method-Based Approaches (Prentice Hall Modern Semiconductor Design Series)*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2005.

[LS09] M. Leucker and C. Schallhart. A brief account of runtime verification. *Journal of Logic and Algebraic Programming*, 78(5):293 – 303, 2009. The 1st Workshop on Formal Languages and Analysis of Contract-Oriented Software (FLACOS'07).

[LTSA10] F. Liu, Q. Tan, X. Song, and N. Abbasi. Aop-based high-level power estimation in systemc. In *GLSVLSI '10: Proceedings of the 20th symposium on Great lakes symposium on VLSI*, pages 353–356, New York, NY, USA, 2010. ACM.

[MAB06] K. Morin-Allory and D. Borrione. Proven correct monitors from PSL specifications. In *DATE'06: Proc. Conf. on Design, automation and test in Europe*, pages 1246–1251. European Design and Automation Association, 2006.

[MMMC05] M. Moy, F. Maraninchi, and L. Maillet-Contoz. LusSy: A toolbox for the analysis of systems-on-a-chip at the transactional level. In *Inter-*

national Conf. on Application of Concurrency to System Design, June 2005.

[MMMC06] M. Moy, F. Maraninchi, and L. Maillet-Contoz. LusSy: an open tool for the analysis of systems-on-a-chip at the transaction level. *Design Automation for Embedded Systems*, 2006.

[MMS90] L. E. Moser and P. M. Melliar-Smith. Formal verification of safety-critical systems. *Softw. Pract. Exper.*, 20(9):799–811, 1990.

[Mø04] A. Møller. dk.brics.automaton. http://www.brics.dk/automaton/, 2004.

[Moo65] G. E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8):114–117, April 1965.

[Moo75] G. E. Moore. Progress in digital electronics. In *Technical Digest of the Internlational Electron Devices Meeting.* IEEE Press, 1975.

[Moo95] G. E. Moore. Lithography and the future of Moore's Law. In *Proc. Society of Photo-Optical Instrumentation Engineers*, volume 25, 1995.

[Moy05] Matthieu Moy. *Techniques and Tools for the Verification of Systems-on-a-Chip at the Transaction Level.* PhD thesis, INPG, Grenoble, France, December 2005.

[NdPF+03] J. A. Nacif, F. M. de Paula, H. D. Foster, C. J. N. Coelho Jr., F. C. Sica, D. C. da Silva Jr., and A. O. Fernandes. An assertion library for on-chip white-box verification at run-time. In *Proceedings of the 4th IEEE Latin-American Test Workshop (LATW'03)*, Natal, RN, Brazil, February 2003.

[NH06] B. Niemann and C. Haubelt. Assertion-based verification of transaction level models. In *ITG/GI/GMM Workshop*, pages 232–236, 2006.

[OH95] P. Øhrstrøm and P.F.V. Hasle. Temporal logic: from ancient times to artificial intelligence. *Studies in Linguistics and Philosophy*, 57, 1995.

[Pet81] James L. Peterson. *Petri Net Theory and the Modeling of Systems.* Prentice Hall PTR, Upper Saddle River, NJ, USA, 1981.

[PF08] L. Pierre and L. Ferro. A tractable and fast method for monitoring SystemC TLM specifications. *IEEE Transactions on Computers*, 57:1346–1356, 2008.

[PF10] L. Pierre and L. Ferro. Enhancing the assertion-based verification of TLM designs with reentrancy. In *Proc. 8th Int'l Conf. on Formal Methods and Models for Codesign.* IEEE, July 2010.

[Piz07] A. Piziali. *Functional Verification Coverage Measurement and Analysis.* Springer Publishing Company, Incorporated, 2007.

[Pnu77] A. Pnueli. The temporal logic of programs. In *Proc. 18th IEEE Symp. on Foundation of Computer Science*, pages 46–57, 1977.

[Pri57] A. Prior. *Time and Modality.* Oxford University Press, 1957.

[PSL07] Standard for property specification language (PSL). *IEC 62531:2007 (E)*, pages 1–156, 2007.

[RV07] Kristin Y. Rozier and Moshe Y. Vardi. LTL satisfiability checking. In *Proc. 14th Int. SPIN conference on Model checking software*, pages 149–167, Berlin, Heidelberg, 2007. Springer.

[SB06] Volker Stolz and Eric Bodden. Temporal assertions using AspectJ. *Electron. Notes Theor. Comput. Sci.*, 144(4):109–124, 2006.

[SGSP02] O. Spinczyk, A. Gal, and W. Schröder-Preikschat. AspectC++: an aspect-oriented extension to the C++ programming language. In *CRPIT '02: Proceedings of the Fortieth International Conference on Tools*

Pacific, pages 53–60, Darlinghurst, Australia, Australia, 2002. Australian Computer Society, Inc.

[SOA08] A. Sen, V. Ogale, and M. S. Abadir. Predictive runtime verification of multi-processor socs in systemc. In *DAC '08: Proceedings of the 45th annual Design Automation Conference*, pages 948–953, New York, NY, USA, 2008. ACM.

[Syn02] Inc. Synopsis. Assertion-based verification. White Paper, May 2002. Available online.

[TCMM07] C. Traulsen, J. Cornet, M. Moy, and F. Maraninchi. A SystemC/TLM semantics in Promela and its possible applications. In *14th Workshop on Model Checking Software SPIN*, July 2007.

[TLSS10] R. Tartler, D. Lohmann, F. Scheler, and O. Spinczyk. AspectC++: An integrated approach for static and dynamic adaptation of system software. *Knowledge-Based Systems*, 2010(in press), 2010.

[TV05] D. Tabakov and M. Y. Vardi. Experimental evaluation of classical automata constructions. In *LPAR'05, 12th Int. Conf. on Logic for Programming, Artificial Intelligence, and Reasoning*, pages 396–411, 2005.

[TV10a] D. Tabakov and M.Y. Vardi. Monitoring temporal SystemC properties. In *Proc. 8th Int'l Conf. on Formal Methods and Models for Codesign*, pages 123–132. IEEE, July 2010.

[TV10b] D. Tabakov and M.Y. Vardi. Optimized temporal monitors for SystemC. In *Runtime Verification*, 2010.

[TVKS08] D. Tabakov, M.Y. Vardi, G. Kamhi, and E. Singerman. A temporal language for SystemC. In *FMCAD '08: Proc. Int. Conf. on Formal*

Methods in Computer-Aided Design, pages 1–9. IEEE Press, 2008.

[Var07] M. Y. Vardi. Formal techniques for SystemC verification. In *DAC '07: Proceedings of the 44th annual conf. on Design automation*, pages 188–192, New York, NY, USA, 2007. ACM.

[Var09] M. Y. Vardi. From philosophical to industrial logics. In *ICLA '09: Proceedings of the 3rd Indian Conference on Logic and Its Applications*, pages 89–115, Berlin, Heidelberg, 2009. Springer-Verlag.

[Vel05] Todd L. Veldhuizen. C++ templates are Turing complete, 2005.

[VR05] S. Vijayaraghavan and M. Ramanathan. *A Practical Guide for SystemVerilog Assertions.* Springer, New York, NY, USA, 2005.

[VW94] M.Y. Vardi and P. Wolper. Reasoning about infinite computations. *Information and Computation*, 115(1):1–37, November 1994.

[Win93] G. Winskel. *The formal semantics of programming languages: an introduction.* MIT Press, Cambridge, MA, USA, 1993.

[WVS83] P. Wolper, M.Y. Vardi, and A.P. Sistla. Reasoning about infinite computation paths. In *Proc. 24th IEEE Symp. on Foundations of Computer Science*, pages 185–194, Tucson, 1983.

Index