DISS. ETH No. 18851

Static Analysis for SystemC with Scoot: From Verification to Simulation

A dissertation submitted to

ETH ZÜRICH

for the degree of

DOCTOR OF SCIENCES

presented by

NICOLAS BLANC

Master of Science, EPFL

born on the 24th of June 1979
in Sion / VS

accepted on the recommendation of

Prof. Dr. Gustavo Alonso,
Dr. Daniel Kröning, and
Dr. Luke Ong

2010

Contents

Summary

SystemC is a description language for computer systems that is based on C++. The language is used for modeling electronic devices at arbitrary levels of abstraction. In particular, SystemC can describe models with both hardware and software aspects. As today's electronic designs are incredibly large and complex, validation of new products remains time consuming as ever; hence the need to keep improving state-of-the-art verification techniques.

We describe verification solutions for SystemC to improve reliability of computer devices by combining formal reasoning and simulation at high-level of abstraction. In particular, our research indicates that formal methods have applications beyond property checking, namely for code optimizations, simulation, and testing. We show that verification engines can be integreted into compilation flows to compute information valuable at runtime, statically.

More specifically, our results demonstrate that an application of formal engines at high abstraction level is practical provided that:

1. the system is partitioned into "small" and independent verification tasks, as monolithic approaches are more subject to scalability issues,

2. the procedure takes advantage of today's parallel computers to distribute those tasks among many cores and

3. authorizes trading precision for time: It is often preferable to provide partial results rapidly instead of complete results after a long period of time.

We investigate solutions for the analysis of systems with hardware and software components, exploring how formal methods, classic static analysis techniques, and simulation can be combined to improve state-of-the-art verification. We have implemented these techniques in a compiler prototype for the analysis of SystemC designs called SCOOT. Given a SystemC model written in C++, SCOOT statically extracts the module hierarchy and generates a model in an intermediate representation that is suitable for formal verification and simulation. SCOOT is the first compiler for SystemC that uses formal methods to improve simulation performance and coverage. In particular, experimental evaluation indicates that SCOOT can significantly speedup the execution of models relevant for industry.

Résumé

SystemC est un langage de modélisation de systèmes informatiques basé sur C++ pouvant être utilisé pour des spécifications de circuits logiques à haut et bas niveaux d'abstraction. En particulier, SystemC est souvent employé pour modéliser des systèmes qui comportent à la fois des aspects matériels et logiciels.

Comme la taille et la complexité des nouvelles générations de systèmes informatiques embarqués continuent de croître, les processus de validation ralentissent toujours plus les cycles de développement de ces nouveaux produits; d'où la nécessité de poursuivre des recherches afin d'améliorer les techniques de vérification actuelles.

Dans cette thèse, nous décrivons des solutions nouvelles de validation de modèles pour SystemC qui combinent à la fois des techniques classiques de simulation et des méthodes récentes de raisonnement formel à haut niveau d'abstraction. En particulier, nos recherches montrent que le domaine d'application des méthodes formelles va au-delà de la simple vérification et que ces méthodes peuvent aussi fournir des résultats utiles pour l'optimisation de code et la simulation.

De manière plus spécifique, nos travaux démontrent qu'une utilisation pragmatique des méthodes formelles à haut niveau d'abstraction est possible dès lors que:

1. le système est partitionné en de petites tâches indépendantes de vérification, car les approches de vérification monolithiques sont fortement sujettes au phénomène d'explosion exponentielle de l'espace de recherche

2. ces tâches sont exécutées en parallèle sur plusieurs machines

3. le degré de précision de l'analyse peut être ajusté: il est souvent préférable de procéder rapidement avec des résultats partiels que d'être bloqué en attente de résultats complets.

Nous présentons des solutions pour l'analyse des systèmes informatiques qui comportent des aspects logiciels et matériels, en explorant comment les développements récents des méthodes formelles peuvent être combinés aux techniques d'analyse statique et dynamique traditionnelles afin d'améliorer les procédures de vérification actuelles. Nous avons construit un prototype de compilateur pour SystemC nommé SCOOT pour évaluer ces solutions. Etant donné un modèle SystemC écrit en C++, SCOOT peut extraire de manière statique des informations concernant la structure du système et construire un modèle dans une représentation intermédiaire adéquate pour la vérification formelle et la simulation. SCOOT est le premier compilateur pour SystemC qui utilise des méthodes formelles pour améliorer les performances de simulation.

1

Introduction

VALIDATION accounts for most of the development time of today's electronic devices. The Electronic Design Automation industry (EDA) is moving toward simpler and less error-prone design methodologies to maintain high design productivity. Bugs frequently arise due to poor language interoperability that fragments design processes. With software models written in C/C++ and hardware descriptions written in VERILOG [Verilog], engineers often lack solutions to ensure convergence of development. Hence the emergence of new system modeling languages, such as SystemC [SystemC] and SYSTEM VERILOG [SystemVerilog], that span the entire design flow: from high-level software prototypes down to hardware descriptions. The *Open SystemC Initiative*'s group (OSCI) ambitions to establish the SystemC language as a standard for hardware and software development. The organisation has support form leading EDA companies. SystemC extends the C++ language to provide support for hardware modeling by means of an external library. SystemC models are thus plain C++ code that can be compiled with standard C++ compilers. The complete language specification and the simulator are released free of charge from `www.systemc.org`. Due to the complexity of C++, the development of static analysis tools for SystemC is extremely difficult. At the moment, SystemC desperately needs solutions for static analysis. So verification relies entirely on simulation and testing. In this thesis, we present techniques to extract information from SystemC models statically. We show that this information not only provides key insight of the model statically but can be exploited at runtime to speedup simulation, significantly.

Increasing design complexity keeps validation time consuming as ever. Properties of interest usually involve large parts of the designs and require very deep exploration of the systems. Currently, system designers can only rely on simulation to verify the correctness of their implementation. Extensive simulation alone can stimulate large parts of a design and catch errors rapidly but usually fails to cover corner cases. In contrast, Formal Methods exhaustively explore the behaviors of a system using symbolic techniques. Chip manufacturers recognize the importance of formal verification for proving correctness of low-level circuit transformations that are difficult to validate using testing techniques: the prestigious Turing Award, which is sponsored by leading computer companies, was

granted in 2007 to Edmund M. Clarke, E. Allen Emerson, and Joseph Sifakis for their work on Model Checking, acknowledging thus the major impact that their research had on designers and manufacturers of semiconductor chips.

Raising the application domain of Formal Methods to software verification has proven tremendously challenging, however. Software models of industrial importance are extremely large, often rely on implicit architecture assumptions, and frequently read data from inputs. In addition, those models contain hard to analysis constructs such as: dynamic data structures, calls to external libraries and dynamic functions, and unsafe pointer conversions. Most powerful simulation-based validation techniques already integrate basic formal-method components for solving constraints at runtime to guide the exploration and to improve coverage. Additionally, recent developments of automated abstraction-refinement techniques have simplified the usage of Formal Methods. Companies such as Microsoft are now promoting model-checking tools for the verification of device drivers. Recent successes with hardware and software model-checking techniques suggest to extend their application domain toward systems that integrate software and hardware aspects.

1.1 Thesis

In this thesis, we provide solutions to improve reliability of computer devices by combining formal reasoning and simulation at high-level of abstractions. *In particular, our research shows that Formal Methods have applications beyond property checking, namely for code optimizations, simulation, and testing.* We claim that applications of formal engines at high abstraction levels are practical provided that:

1. the system under test is partitioned into "small" and independent verification tasks, as monolithic approaches are more subject to scalability issues,

2. the procedure takes advantage of today's parallel computers to distribute those tasks among many cores and

3. authorizes trading precision for time: It is often preferable to provide partial results rapidly instead of complete results in long amount of time.

We provide evidence to support this statement by showing that state-of-the-art model-checking techniques can be integrated into a compiler to take advantage of the domain-specific semantics of SystemC. The compiler derives precise predicates over fragments of SystemC models using formal engines and can control the time spent for analysis: Our technique iteratively weakens safe under-approximations. The computation of the predicates can thus be stopped early before reaching a fixed-point.

1.2 Summary of the Contributions of this Dissertation

Hardware/software co-verification creates new verification challenges but is crucial for the development of reliable systems. In this thesis, we investigate solutions for the analysis of systems that combine software and hardware aspects, exploring how Formal Methods, classic static analysis techniques, and simulation can be combined to improve state-of-the-art verification techniques. We have implemented those techniques in a compiler prototype called SCOOT for the analysis of SystemC designs. Given a SystemC model written in C++, SCOOT statically extracts of an intermediate representation that is suitable for formal verification and simulation. SCOOT is the first research compiler for SystemC that uses Formal Methods to improve simulation performance and coverage. We summarize our contributions in the next paragraphs.

Research Compiler for C++ We extend model-checking techniques to C++. The C++ language is one of the most widely adopted language and is used for applications such databases, games, embedded systems. We have built a research compiler for C++ that supports a broad range of the language's constructs such as inheritance, method overloading, virtual functions, and template programming. The compiler is tightly integrated in a formal verification framework that offers decision procedures for bounded model checking and predicate abstraction among others. We use this frontend for the verification of applications relevant to industry such as the popular SAT solver MiniSAT and SystemC cryptographic models. In contrast, all previous efforts to model check C++ code are based on dynamic execution.

Model-checking Techniques for C++/STL The C++ standard also defines the Standard Template Library (STL)[Stepanov and Lee, 1994] that offers generic components such as vectors, lists, and maps, and iterators to manipulate them. In general, considerable effort are necessary to abstract data structures operations, which are not a strong suit for formal verification engines. The STL provides clear separation between high-level manipulations concepts and their implementations. We develop an axiomatic semantics to capture the concepts of iterator and container, providing a formal basis for reasoning about STL usage. From our axiomatic semantics, we show how to derive an operational model to verify the correctness of C++/STL programs.

Static Analysis of SystemC Models Technically, SystemC is a C++ library that provides essential components for modeling synchronous systems such as modules, ports, and signals. SystemC models are plain C++ programs that can be compiled and linked to the SystemC library. In contrast to other hardware description languages such as SYSTEM VERILOG, the module hierarchy of SystemC

models is built dynamically at runtime using complex design patterns unsuitable for static analysis. We have developed SCOOT, a research compiler for SystemC. Our approach is to build a static analyser with built-in knowledge of the SystemC semantics and to replace the original SystemC library with simpler declarations for the only purpose of typechecking. Our SystemC analysis relies on precise data-flow information and is able to discover module hierarchy, communication channels, and processes at compile time. The models extracted by SCOOT can serve several purposes ranging from formal verification to simulation.

Static Scheduling Techniques for SystemC SCOOT uses its built-in knowledge of SystemC to perform high-level code transformations. In particular, the original SystemC scheduler contains several sources of inefficiencies: processes are triggered using function pointers and the scheduling algorithm stores information using dynamic data structures. Among other optimization, SCOOT can use its static knowledge to replace function pointers with their actual targets. The model generated by SCOOT can then be compiled to produce significantly faster simulators. Experimental evidence shows that our tool can be used to improve performance of dynamic execution up to five times.

Race-Analysis for SystemC SCOOT is integrated in a formal verification framework. We have developed the first SystemC compiler that uses formal analysis to improve simulation performances. SystemC models are inherently concurrent and thus can exhibit nondeterministic behaviors. The SystemC language offers race-free communication channel such as signals to describe synchronous circuits. However the elimination of races is not always desirable: in practice, system designers often model nondeterministic choices implicitly using constructs that yield races. Due to the combinatorial explosions of process interleavings, testing methods for concurrent programs alone are unlikely to discover errors that depend on subtle interleavings. For each pair of processes, SCOOT uses an original formal analysis technique to compute a process dependency predicate. This information acquired at compile time is not only useful to prove or refute process dependencies statically but can also be used to improve simulation coverage. SCOOT offers to build a simulator for the exhaustive coverage of the schedules that can prune the exploration at runtime, using information from our formal analysis, to avoid visiting redundant schedules.

1.3 Organisation of the Thesis

In Chapter 2, we motivate the introduction of new design methodologies based on SystemC, and we provide the necessary background on Model-Checking.

In Chapter 3, we provide an overview of our C++ frontend, and we describe our support for key C++ mechanisms, e.g, inheritance, function overloading, and templates. We also present the internal representation of our compiler in detail.

In Chapter 4, we extend Model-checking to the verification of C++/STL programs using SATABS. We specify operations on iterators and containers using pre- and post-conditions and we show how to check the validity of iterators using Model-Checking.

In Chapter 5, we describe SystemC and its execution model at an informal level. Subsequently, we provide a fixed-point semantics for SystemC in Chapter 6 that captures the key aspects of the scheduling algorithm and abstract away secondary details.

In Chapter 7, we present our static extraction technique for SystemC models. We show that those models are useful to speedup simulation performances.

In chapter 8, we describe an application of Model-checking for race analysis. The result of this formal analysis is not only useful to diagnose race statically, but can also be used to improve exhaustive simulation.

In the appendixes, we provide additional material that is related to our research. Our experience with the verification of Linux device drivers is described in Appendix A. We present in Appendix B an automated technique that we developed to encode test benches written in C/C++ into a set of SYSTEM VERILOG properties suitable for formal verification.

2

Background

2.1 Overview of System-Level Modeling Languages

TIME-TO-MARKET requirements have rushed the electronic design and automation (EDA) industry towards design paradigms that require a very high level of abstraction. This high level of abstraction can shorten the design time by enabling the creation of fast executable verification models. This way, bugs in the design can be discovered early in the design process.

As part of this paradigm, an abundance of system design languages such as SPECC [Fujita and Nakamura, 2001], SYSTEM VERILOG [SystemVerilog], and SystemC have emerged. A key feature is joint modeling of both the hardware and software component of a system using a language that is well-known to engineers. SPECC from the university of California, Irvine, extends the syntax of ANSI-C to allow the description of hybrid systems. The language offers support for modular composition, primitive hardware channels, ports, and threads. In contrast, SYSTEM VERILOG is a system-level modeling language that extends the syntax of VERILOG to support software programming – SYSTEM VERILOG offers object classes, polymorphism, pointers, and dynamic allocation of memory, and comes with a powerful assertion language for verification.

In 1999, Synopsys (together with other industrial partners) funded the Open SystemC Initiative (OSCI) to develop a standard system description language based on C++. The first version of the language was released one year later in 2000. In 2005, IEEE approved the IEEE 1665 standard for SystemC. The open nature of the project and support from key industrial actors contributed to the world-wide acceptance of SystemC in industry. The design of SystemC is unique: in contrast to SPECC/SYSTEM VERILOG, which extend the syntax of C/VERILOG, SystemC is implemented as a C++ library. SystemC modules are, therefore, plain C++ classes, which are compiled and then linked to a runtime scheduler. This provides a simple and efficient way to simulate the behavior of the system. SystemC continues to evolve: the OSCI group is extending the application domain of SystemC above RTL by adding new libraries specialized for transaction-level modeling (TLM). At the time of writing, SystemC v2.2 is the latest version of the language and was released in 2007. In this document, we concentrate our discussion on SystemC, noting that similar observations can be drawn for other

system-level modeling languages.

2.1.1 Classic Design Methodologies

Figure 2.1 The diagram depicts a traditional design methodology. A fork occurs in the development cycle, and the hardware and software parts are refined independently.

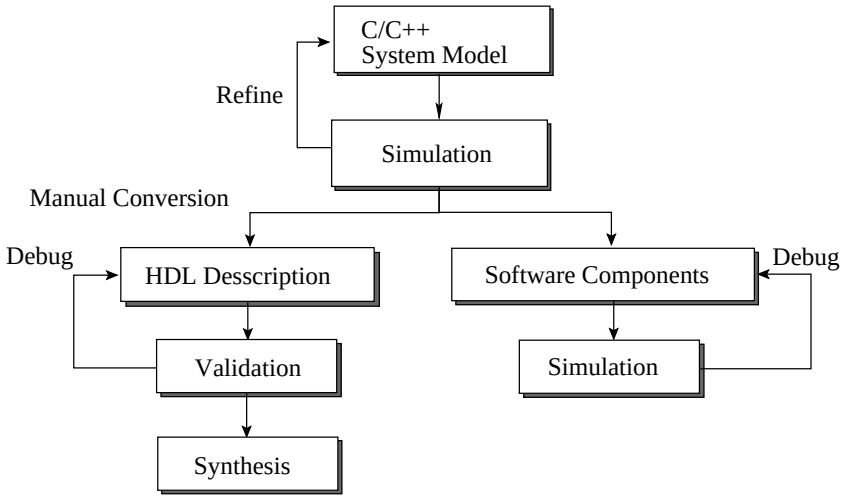
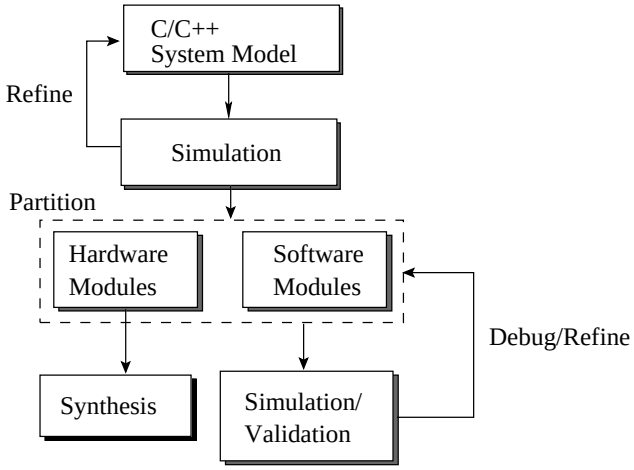


Figure 2.1 depicts a design methodology that starts with a high-level model of the system written in C or C++. The behavior of the system is then refined up to the point where the partitioning between the hardware and software components is decided. Typically, hardware components are written in VERILOG. Subsequently, a branch occurs in the development process, and the verification for the software and hardware parts is performed independently. As a result, the branches may unexpectedly diverge, corrupting the correctness of the global system. Additionally, the lack of uniformity causes productivity losses as test benches and code from previous phases cannot be reused easily. Finally, designers must juggle with different languages and different environments, which require significant learning efforts and can be source of errors.

2.1.2 Design Methodologies using SystemC

In contrast, figure 2.2 depicts a design methodology that uses SystemC at every stage of the development. The process starts with a system-level model, which is refined until the partitioning is performed. Since SystemC can be used to model circuits, both hardware and software components are described using the same language, and thus, can be refined and verified conjointly. The consistency of the global system can be tested at any time. Typically, methodologies based on SystemC are simpler and less error prone. Test benches and modules

Figure 2.2 The diagram depicts a design methodology that uses SystemC for the description of both hardware and software components.



can be reused and exchanged more easily, which improves productivity and accelerates the design cycle.

2.2 Model Checking

Simulation-based approaches and testing techniques only cover a subset of executions. In contrast, Formal Methods (FM) designates mathematic techniques for reasoning about computer systems that ensure an exhaustive coverage of the behaviors. The literature distinguishes FM based on Theorem Proving and Model-Checking. Theorem Provers are semi-automated reasoning tools. Verification using Theorem Prover is flexible but requires expertise and significant efforts to craft proofs. Model Checkers implement fully automated decisions procedures and are, thus, accessible to a larger audience of designers. Automated verification techniques for hardware and software have been developed since the seventies [King, 1976, Cousot and Cousot, 1977, Clarke et al., 1983] and are now promoted by major EDA companies. In the rest of this section, we provide an overview the Model-Checking techniques that are integrated in our framework.

A Model Checker verifies that the system under test fulfills its specification. In case the specification does not hold a Model Checker returns a counterexample exposing the bad behavior, which provides useful information to the programmer for debugging. In general, checking whether a specification holds is undecidable. Model-Checking is sound meaning that the absence of counterexample is asserted correctly. Consequently, the verification process may not converge: a Model Checker may fail to report an answer within a certain limit of time, in which case, no conclusion can be drawn. Many systems of interest are actually finite, e.g, circuits. For those systems, Model-Checking is also complete: Eventually a Model Checker produces an answer; though in practice, the execution may

time out or exhaust memory resources.

The main limitation of Model Checkers is the so called state explosion issue induced by storage elements: a state vector of n bits describes a set of 2^n different states, and in worst-case scenario, the Model Checker needs to enumerate all of them for reachability. The theory of abstract interpretation [Cousot and Cousot, 1977], developed during the seventies and eighties, is now well understood and provides mathematical solutions to tackle this explosion issue [Graf and Saïdi, 1997].

Combined with modern SAT solvers such, e.g, Chaff [Moskewicz et al., 2001] and *Minisat* [Eén and Sörensson, 2003], and BDD techniques [Bryant, 1986, McMillan, 1993], the performance of Model Checkers have greatly improved during the last decade. This enabled researchers to focus on solutions to verify concurrent software, which poses greater challenges due to the additional explosion of process interleavings. An abundance of techniques has been developed to overcome this issue using *Partial order reduction* [Peled, 1994, 1993, Godefroid, 1996] and *Symmetry reduction* [Emerson and Trefler, 1999, Miller et al., 2006]. The former techniques can detect redundant schedules that can be skipped during exploration, whereas the latter techniques exploit symmetry among processes to collapse equivalent states and to reduce the size of the system.

As verification techniques for software and hardware are making progresses, new applications for Model-Checking are emerging to verify concurrent models with both hardware and software components [Kroening et al., 2003].

2.2.1 Bounded-Model Checking

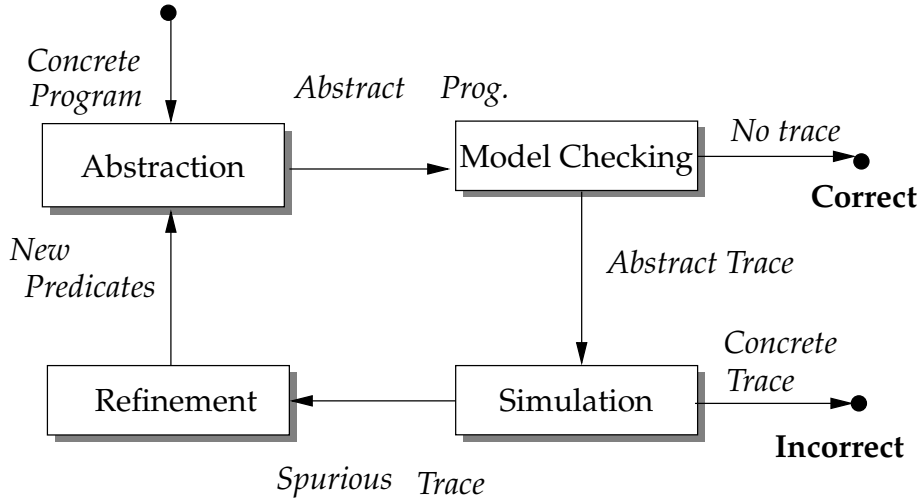
A large number of verification tasks are successfully performed using decision procedures based on SAT [Sheeran et al., 2000, McMillan, 2005, Biere et al., 2003] thanks to the development of fast SAT engines.

Bounded-Model Checking (BMC) [Biere et al., 2003, Kroening et al., 2004] is one of them: the Model Checker jointly unwinds a transition relation T and a specification ϕ up to a certain bound i, starting from a set of initial states I. The Model Checker builds then a Boolean formula that is satisfiable if and only if the model mismatches its specification within i steps:

$$I(s_0) \wedge T(s_0, s_1) \wedge T(s_1, s_2) \wedge ... \wedge T(s_{i-1}, s_i) \wedge \neg\phi(s_0, s_1, ..., s_i)$$

This formula is then passed to a SAT solver. In case the formula is satisfiable the Model Checker reconstructs a counterexample in the original model from the satisfiable assignment. The method is complete only if i exceeds the completeness threshold [Kroening and Strichman, 2003]. In practice, BMC is used to assert safety properties up to a specific bound. To overcome the limitations of BMC, designers can use other techniques based on SAT such as inductive reasoning [Sheeran et al., 2000] and interpolation-based reachability analysis [McMillan, 2005].

Figure 2.3 CEGAR Loop



BMC has been successfully applied for the verification of software and hardware systems and has been adapted for HW/SW co-verification in [Kroening et al., 2003] – Appendix B describes a similar technique in detail.

2.2.2 Predicate Abstraction

Automated software verification has taken a huge leap forward with the introduction of predicate abstraction [Graf and Saïdi, 1997] and counterexample-guided abstraction refinement (CEGAR) [Clarke et al., 2000]. Pioneered by the model checking tools SLAM [Ball and Rajamani, 2002] and BLAST [Henzinger et al., 2002], the technique has been successfully applied to analyze device drivers with more than 10,000 lines of code [Ball et al., 2006, Henzinger et al., 2003a]. The CEGAR approach comprises four phases, namely abstraction, model checking, simulation, and refinement. These four phases are executed repeatedly, until either a counterexample is found, or the program under test is proved correct. The CEGAR process is fully automated: except for providing the property to be verified, no user interaction is required. Figures 2.3 illustrates the procedure:

1. *Abstraction*: In the first phase, predicate abstraction is used to generate a finite state abstraction $\hat{T}$ of the original program T. The variables of $\hat{T}$ correspond to a finite set of predicates over the variables of T. Each predicate φ_i describes an observable "fact" about the program, and the valuation of a variable b_i in an abstract state determines whether the corresponding fact holds or not. For each instruction of T, the corresponding abstract transition is constructed independently of the surrounding instructions. The abstraction step preserves the control flow structure of the original program T. The abstract program $\hat{T}$ contains all execution traces of T, and potentially more.

2. *Verification*: In phase two, the abstract model $\hat{T}$ is examined by a model

checking tool. The abstract model contains only Boolean variables. For this purpose, Model Checkers rely on verification engines for abstract programs such as BOPPO [Cook et al., 2005] and Cadence SMV [K.L. McMillan, 1992]. Several other efficient symbolic model checking algorithms based on summarization and saturation can also be used (e.g., [Ball and Rajamani, 2000b, Schwoon, 2002, Basler et al., 2007]). In the presence of threads and recursion, the reachability problem is undecidable. BOPPO deals with this problem by over-approximating the abstract transition relation [Cook et al., 2005]. If the model checker finds a witness for for the reachability of an error location in $\hat{T}$, it is reported as an *abstract counterexample*. If no error traces are found, we can conclude that the original program is correct.

3. *Simulation*: If a counterexample is found in the abstraction, the third phase is entered. In that phase, symbolic simulation is used to determine whether the counterexample can be replayed in the original program. The existence of an abstract counterexample in $\hat{T}$ does not necessarily imply that the error state is reachable in the original program T. Since there is a one-to-one correspondence between the locations of the abstract program and the original program, it is sufficient to check whether there exists a feasible sequence of transitions in T that traverses the same locations as the abstract counterexample.

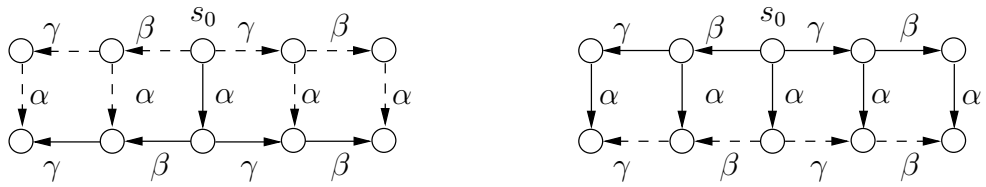
4. *Refinement*: In phase four, the abstract model is refined to block the abstract trace (and potentially others) in a way that preserve soundness of the verification. This refinement is achieved by adding new predicates to the abstract model that increase the accuracy of the transition relation and render the current abstract trace infeasible.

2.2.3 Partial-Order Reduction

Model Checking is an algorithmic technique for exhaustive exploration of transition systems [Edmund M. Clarke et al., 1999]. However, Model Checking when applied naïvely scales poorly on models with asynchronous concurrent components, as the number of possible interleavings rapidly explodes. *Partial order reduction* is a technique to explore the state space of concurrent systems in a way that preserves the soundness of the verification result [Peled, 1994, 1993, Godefroid, 1996]. The key idea is to exploit commutativity of transitions to obtain a subset of all possible interleavings from a state such that the reduced-state graph retains a representative behavior for each behavior that is removed. We briefly survey partial order reduction using persistent sets and sleep sets in this section [Godefroid, 1996]. We describe an implementation of these techniques for SystemC in Chapter 8.

Persistent and Sleep Techniques

Figure 2.4 Example of partial order reduction using persistent sets (1) and sleep sets (2). The reduced state graph contains only the transitions depicted with solid lines.



(1) Exploration using persistent sets *(2) Exploration using sleep sets*

The persistent-set and sleep-set techniques both preserve deadlock states,i.e, states without outgoing transition. The persistent-set and sleep-set techniques are orthogonal and achieve better results when combined. Both techniques compute a subset of the runnable transitions for each visited state and restrict future exploration to transitions in this set.

Let $(S, S_0, \rightarrow)$ denote a transition system with a set of states S, initial states $S_0 \subset S$, and a set of transitions $\rightarrow$. A transition $\alpha \in \rightarrow$ is a relation on S. For $\alpha \in \rightarrow$, we write $s \xrightarrow{\alpha} t$ if $\langle s, t\rangle \in \alpha$. A transition α is *runnable* in a state s if there exists a state t such that $s \xrightarrow{\alpha} t$, and we write $\alpha \in Runnable(s)$ to denote this fact – $Runnable$ is a function from S to $2^{\rightarrow}$. Otherwise, α is *sleeping* in s.

Definition 2.2.1. *[Wang et al., 2008] Two transitions α and β are* guarded independent *with respect to a guard $\phi \subseteq S$ if and only if for all $s \in \phi$ the following hold:*

1. $\alpha \in Runnable(s) \wedge s \xrightarrow{\alpha} t \Rightarrow$
   $\beta \in Runnable(s) \Leftrightarrow \beta \in Runnable(t)$
2. $\beta \in Runnable(s) \wedge s \xrightarrow{\beta} t \Rightarrow$
   $\alpha \in Runnable(s) \Leftrightarrow \alpha \in Runnable(t)$
3. $\alpha, \beta \in Runnable(s) \Rightarrow$
   $\langle s, t\rangle \in \alpha \circ \beta \Leftrightarrow \langle s, t\rangle \in \beta \circ \alpha$

The first two conditions guarantee that α and β cannot disable nor enable each other in s, while the third condition requires α and β to be commutative in s. Transitions α and β are *independent in s* if and only if α, β are guarded independent with respect to the guard $\{s\}$ [Godefroid, 1996].

Definition 2.2.2. *[Godefroid, 1996] Let $D \subseteq \rightarrow \times \rightarrow$ be a symmetric and reflexive relation over the transitions of the system. The relation D is a* valid dependency relation *for $\rightarrow$ if and only if $(\alpha, \beta) \notin D$ implies that α, β are independent in all reachable states.*

Definition 2.2.3. *[Godefroid, 1996] Let $(S, S_0, \rightarrow)$ be a transition system, and $s_0 \in S$ denote one of its states. A set of transitions $T \subset Runnable(s_0)$ is* persistent *in s_0 if and only if for all $\beta \in T$ and all sub-traces $s_0 \xrightarrow{\alpha_0} s_1 \xrightarrow{\alpha_1} s_2 ... s_n \xrightarrow{\alpha_n} s_{n+1}$ obtained from transitions $\alpha_i \notin T$, β, and α_i are independent in s_i.*

The Definition 2.2.3 is, thus, concerned about what can happen in the *future*. The persistent-set technique computes a persistent set of runnable transitions in each visited state and restricts the exploration to transitions in this set only. Model Checkers typically compute persistent sets using information from a preliminary static analysis.

Figure 2.4.1 illustrates the effects of the persistent-set technique. In state s_0, the exploration uses the persistent set $T = \{\alpha\}$ to avoid visiting some of the states. In contrast, the sleep-set technique maintains a set of runnable transitions that can be skipped during the exploration (the sleep set). The method is concerned with branching information from the *past*. Figure 2.4.2 shows a typical exploration using sleep sets. Unlike the previous approach, the sleep-set technique only reduces the number of explored transitions and has no effect on the number of explored states. The exploration backtracks early when the sleep set contains all runnable transitions.

3

A Research Compiler for C++

3.1 Introduction

In this chapter, we describe the research compiler for C++ that we have developed as part of our project to build a static analyser for SystemC. The frontend of our compiler was successfully employed for the analysis of large SystemC models from industrial partners. We provide an overview of the compilation process to clarify how key C++ mechanisms are implemented. This information is valuable when analysing C++ models and provides insight into performance and verification issues when programming.

The C++ language is one of the most effective programming languages for fast execution results and is generally regarded as the most-efficient object-oriented language. Indeed, most computationally-intensive applications such as compilers, virtual machines, 3D engines, and SAT solvers are generally written in C++. The language was first developed at Bell Labs as an extension to C and was originally named *"C with Classes"*. Minor details apart, C++ is a superset of C, i.e, most C programs can be compiled using C++ compilers[1]

C++ offers support for traditional object-oriented mechanisms such as *encapsulation*, *inheritance*, and *polymorphism*. As with traditional object-oriented languages, C++ classes are used to group data and functions together within a same programming structure. The programmer of a class can then hide low-level routines and choose to expose only the functionalities that are relevant to the end-user. In addition, C++ offers to combine classes using multiple inheritance, which is a mechanism that is extensively used in many libraries including SystemC. Finally, polymorphism can be used in C++ to adjust the behavior of an object at runtime using function overriding and virtual methods.

In software engineering, encapsulation, inheritance, and polymorphism are the key concepts that enable the development standard and reusable solutions to frequent programming tasks, e.g., container iterators and the visitor design pattern [Gamma et al., 1993]. In addition to object-oriented paradigms, C++ implements a powerful template programming language for writing generic code. C++ templates provide clear benefits in term of code compactness and execution

[1]Program 3.1 illustrates an incompatibility issue between C and C++.

Program 3.1 In general, C++ is viewed as an extension of C. The example below illustrates one of the few discrepancies between these two languages. The lazy declaration of the function *func* is legal in C but illegal in C++.

```
int main(int argc, char* argv[]){
  // The declaration of 'func' comes later.
  return func();
}

// Declaration of 'func'
int func(){
 return 0;
}
```

performances over C. Among other characteristics, C++ has support for template functions, template classes, template specialization, and automatic inference of template parameters. As reported by Todd [1996], C++ templates are expressive enough for meta programming: programmers can use templates to compute complex tasks at compile time.

On the down side, C++ is a language hard to master with numerous syntactic and semantic issues: it is an error prone language for programmers. The C++ standard describes the language using informal (and often vague) English text that is subject to interpretation. Moreover, some aspects of the semantics of C++ are left to the implementation, others aspects are unspecified or undefined as confirmed by the IEEE standard:

> "Certain aspects and operations of the abstract machine are described in this International Standard as **implementation-defined** (for example, sizeof(int)). These constitute the parameters of the abstract machine. Each implementation shall include documentation describing its characteristics and behavior in these respects. Such documentation shall define the instance of the abstract machine that corresponds to that implementation (referred to as the "corresponding instance" below).
>
> Certain other aspects and operations of the abstract machine are described in this International Standard as **unspecified** (for example, order of evaluation of arguments to a function). Where possible, this International Standard defines a set of allowable behaviors. These define the nondeterministic aspects of the abstract machine. An instance of the abstract machine can thus have more than one possible execution sequence for a given program and a given input.
>
> Certain other operations are described in this International Standard as **undefined** (for example, the effect of dereferencing the null pointer)."
>
> The ISO 14822 C++ Standard, page 5.

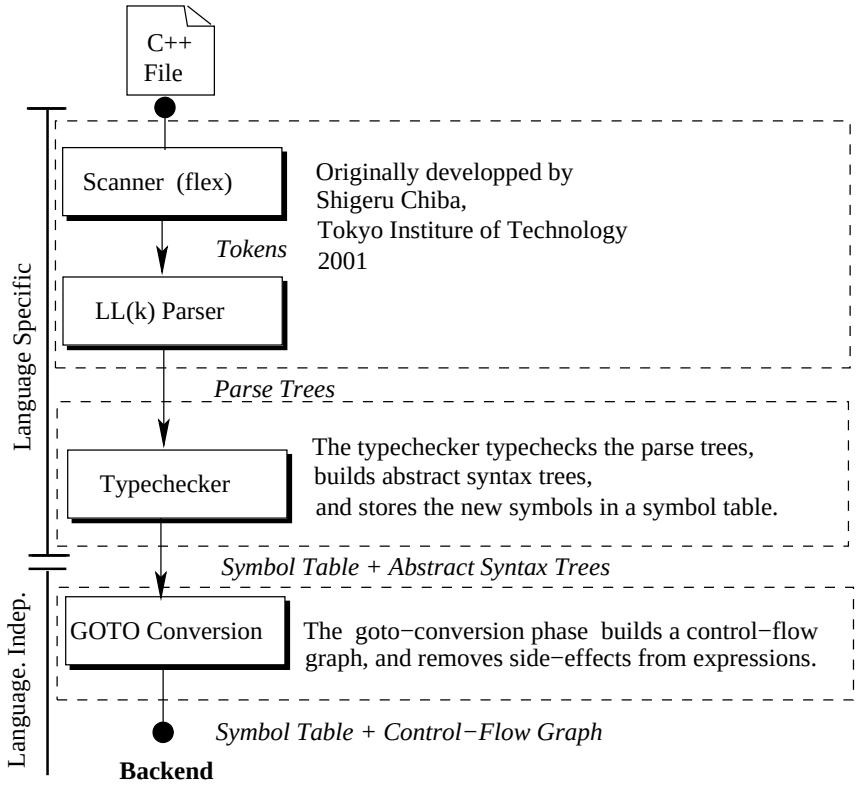
Needless to mention that compilers provide incomplete information about their implementation choices, at best. Consequently, C++ programs may be subject to portability issues. Even more problematic, unspecified behaviors, e.g., the evaluation order of expressions, may cause execution discrepancies among binaries compiled with a same compiler but using different optimization modes. This lack of formal model renders verification of C++ programs particularly tedious. We propose to overcome this semantic issue by considering that the compiler provides the operational definition of the language: The frontend converts C++ programs into an intermediate representation that has a clear semantics. This model can then be verified statically using formal methods and compiled to produce an executable binary. By construction, our compilation flow reduces the risk of mismatches between simulation and verification – previous model-checking attempts for C++ were based exclusively on explicit-state-exploration techniques to avoid mismatches [Musuvathi et al., 2002].

The development of a frontend for C++ is vast enterprise due to the complexity of the syntax and the semantics – the official standard describes the languages over more than 700 pages. As of today, C++ is rightfully regarded as one of the most difficult language for both humans and compilers, which explains the current lack of analysis tools that support C++ compared to more recent languages such as Java or C#. Our C++ parser and the typechecker alone contain 30'000 lines of codes. In order to facilitate the development, we chose to reuse a scanner and a parser that were originally developed by Shigeru Chiba from the Tokyo Institute of Technology and to adapt them to our needs. As C++ derives from C, we also decided to build our frontend for C++ on top of a C frontend available in our framework.

3.2 Overview of the Frontend

The frontend is regarded as the most important, and complex, part of a C++ compiler as it performs syntactic and semantics analysis of the source files. Figure 3.1 provides an overview of the frontend of our compiler. First, the scanner reads characters from the C++ file, filtering white space and comments, and converts the stream of character to a sequence of tokens, e.g., identifiers, keywords, constants, punctuation. Subsequently, the parser reads the stream of tokens and verifies the syntax of the input program in a top-down style. During the recursive descent, the parser enforces syntactic conventions such as operator precedence and associativity and builds *parse trees* to capture the syntactic structure of the C++ program. Parse trees contain constructs specific to the C++ language that must be rewritten in more general form. So the typechecker performs semantical analysis and converts parse trees into *abstract syntax trees* (AST), which are easier to analyse and language independent. Beside verifying the type of expressions, the typechecker instantiates templates, resolves overloaded function calls, and performs implicit conversions. Additionally, the typechecker associates a unique

Figure 3.1 Overview of the C++ frontend.



symbol to each declaration of variable, type, function, and template. Those symbols and their related informations are then stored in a symbol table. This process is repeated for each C++ file given as input. After typechecking, the symbol tables of each file are merged together. At this stage, the program has been fully converted to a language independent-tree representation, which is subsequently converted into a control-flow graph.

3.3 The Internal Representation

In this section, we provide an informal overview of our language-independent representation, focusing on the original constructs used to facilitate symbolic reasoning. We use this intermediate language as a portable representation that can be interpreted by mean of a simulator or compiled into machine code for fast execution performances.

3.3.1 Types

Figure 3.2 shows the syntax of types. In addition to the fundamental types for booleans, integers, floating-point numbers, and fixed-point numbers, our intermediate representation offers types for pointers, arrays, functions, and com-

Figure 3.2 Types.

$Type$	::=	`bool` | `empty` | *alias*$_{id}$ | *IntbvType* | *FloatbvType* | *FixedbvType* | *PtrType* | *ArrayType* | *FuncType* | *FuncType*
$IntbvType$	::=	`signedbv` <*width*$_{\mathbb{N}}$> | `unsignedbv` <*width*$_{\mathbb{N}}$>
$FloatbvType$	::=	`floatbv` < *width*$_{\mathbb{N}}$, *mantissa*$_{\mathbb{N}}$>
$FixedbvType$	::=	`fixedbv` < *width*$_{\mathbb{N}}$, *integer*$_{\mathbb{N}}$>
$PtrType$	::=	Sub_{Type} *****
$ArrayType$	::=	Sub_{Type} **[**$size_{\mathbb{N}}$**]**
$StructType$	::=	`struct` **{** {*field*$_{id}$ *Type*} **}**
$FuncType$	::=	$Return_{Type}$ **(** { Arg_{Type} } **)**

pound data structures. In the usual way, we write `bool` to denote a Boolean type. Many languages provide an empty type to indicate that a function returns no value or that a pointer is pointing to an object of unknown type and cannot, thus, be dereferenced. We write `empty` to denote this empty type. Given a type t_{sub} and a strictly positive integer `s`, we write t_{sub} $[s]$ to denote an array of `s` elements of type t_{sub}. In a similar way, $t_{sub} *$ denotes a pointer type to objects of type t_{sub}. Note that C++ offers reference types. Technically, references are plain pointers. This distinction between references and pointers is relevant only during typechecking: internally, the typechecker converts references to pointers. Function types define inputs and outputs of functions. Given a return type t_{ret} and two argument types t_1 and t_2, we write t_{ret} (t_1, t_2) to denote the type of the functions that take as input two arguments of types t_1 and t_2 and returns a value of type t_{ret}. Our intermediate representation also supports type aliases. They are used for breaking cycles that may appear in declaration of compound types. In the next paragraphs, we describe bit-vector types and compound types.

Bit-Vector Types

When modeling circuits, system designers need to represent bit vectors of arbitrary length. Languages such as VERILOG or VHDL have built-in support for bit-vector arithmetic. In contrast, software programming languages such as C or C++ only provide similar functionalities through external libraries that are difficult to analyse statically. Our intermediate representation handles bit-vector types natively to facilitate formal reasoning.

Given a strictly positive integer `w`, we write `signedbv<w>` and `unsignedbv<w>` to denote signed and unsigned bit-vector types of width `w`, respectively – note

that `unsignedbv<1>` and `signedbv<1>` denote types different from `bool`. On 32-bit machines type `int` is mapped to type `signedbv<32>`.

Additionally, our intermediate representation supports arbitrary floating-point and fixed-point types: `floatbv<w,m>` denotes a floating-point type for bit-vectors of width `w`. The second type parameter `m` indicates the size of the mantissa. In a similar way, `fixedbv<w,i>` denotes a fixed-point type for bit-vectors of width `w`. The second type parameter `i` indicates the size of the integer part.

Compound Types

Structures enable to aggregate heterogeneous data together and to access them randomly using labels. For instance, we write

$$\texttt{struct\{unsignedbv<32>}\, x\, \texttt{, unsignedbv<32>}\, y\, \texttt{\}}$$

to denote a structure with two fields named `x` and `y`, respectively. Both fields are 32-bits unsigned integers. Note that we restrict our discussion to non-recursive data-types, that is, a structure `S` cannot contain values of same type as `S`. Our internal representation of classes and structures is compatible with C: members are laid out in memory in their declaration order.

Methods The C++ language offers to encapsulate data and methods together, providing a clear way of organizing code. In addition to methods, a C++ class may declare several constructors to initialize the state of a newly created object and a destructor to release the resources owned by an object before it is destroyed. Encapsulation is a high-level programming concept. Internally, the frontend flattens encapsulation, i.e., Methods, constructors, and destructors are converted to plain C functions. The next lines shows the declaration of a structure `B` with a constructor, a destructor, and two overloaded methods:

```
struct B {
  B();                          // Constructor
  ~B();                         // Destructor
  void set(int v);              // Overloaded Method
  void set(bool v);             // Overloaded Method
};
```

The previous declaration yields the following function symbols after type-checking:

Symbol Table	
Identifier	*Type*
`B_ctorB`	`empty(B*)`
`B_dtorB`	`empty(B*)`
`B_set_int`	`empty(B*, unsignedbv<32>)`
`B_set_bool`	`empty(B*, bool)`

The first parameter corresponds to the `this` pointer – the object associated with the call – and is added implicitly by the typechecker. As for data members, methods are renamed using a complex renaming scheme that refer to the identifier of the class that declare them, e.g., constuctor `B` is renamed *B_ctorB* [2]. Additionally to permit overloaded declarations, the identifier of a method reflects its signature: in previous example, the two methods *set(int)* and *set(bool)* are renamed to *B_set_int* and *B_set_bool*. Note that the type of the `this` parameter is left apart.

Figure 3.3 Syntax of Expressions.

$$
\begin{aligned}
\mathit{Expr} ::= \ & \texttt{true} \mid \texttt{false} \mid \texttt{null} \mid \mathit{variable}_{id} \mid \mathit{string\_of\_bits} \mid \mathit{string\_of\_chars} \\
& \mid \mathit{Expr}(\ \texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{/} \mid \texttt{mod}\ )\ \mathit{Expr} \\
& \mid \mathit{Expr}(\ \texttt{shl} \mid \texttt{lshr} \mid \texttt{ashr}\ )\ \mathit{Expr} \\
& \mid \mathit{Expr}(\ \texttt{bitand} \mid \texttt{bitor} \mid \texttt{bitxor}\ )\ \mathit{Expr} \\
& \mid \texttt{bitnot}\ \mathit{Expr} \\
& \mid \mathit{Expr}(\ \texttt{=} \mid \texttt{<} \mid \texttt{>} \mid \texttt{<=} \mid \texttt{>=}\ )\ \mathit{Expr} \\
& \mid \mathit{Expr}(\ \texttt{and} \mid \texttt{or} \mid \texttt{xor}\ )\ \mathit{Expr} \\
& \mid \texttt{not}\ \mathit{Expr} \\
& \mid \mathit{Expr}\ \texttt{.}\ \mathit{member\_name}_{id} \\
& \mid \mathit{Expr}\ \texttt{[}\mathit{Expr}\texttt{]} \\
& \mid \texttt{array\_of(}\mathit{size}_{\mathbb{N}}, \mathit{Expr}\texttt{array\_of)} \\
& \mid \texttt{\&}\ \mathit{Expr} \\
& \mid \texttt{*}\ \mathit{Expr} \\
& \mid \texttt{typecast} < \mathit{Type} > \texttt{(}\mathit{Expr}\texttt{)} \\
& \mid \texttt{array\_with(}\mathit{Expr}, \mathit{Expr}, \mathit{Expr}\texttt{)} \\
& \mid \texttt{struct\_with(}\mathit{Expr}, \mathit{member}_{id}, \mathit{Expr}\texttt{)} \\
& \mid \texttt{concat(}\mathit{Expr}, \mathit{Expr}\texttt{)} \\
& \mid \texttt{extractbit(}\mathit{Expr}, \mathit{Expr}\texttt{)} \\
& \mid \texttt{extractbits(}\mathit{Expr}, \mathit{Expr}, \mathit{Expr}\texttt{)} \\
& \mid \mathit{Expr}\texttt{?}\mathit{Expr}\texttt{:}\mathit{Expr} \\
& \mid \texttt{nondet}\mathit{Type} \\
& \mid \texttt{new}\mathit{Type}
\end{aligned}
$$

3.3.2 Expressions

Figure 3.3 shows the prefix-notation of the internal representation for expressions. Expressions represent values, not actions, and thus, expressions have no

[2] For the sake of clarity and conciseness, we use renaming scheme in the examples that is simpler than the actual one.

side-effects. The language offers traditional operators for integers, boolean values, pointers, arrays, and structures. In addition to the classic operators, our intermediate representation offers operators for bit-extraction, concatenation, and member substitutions. Most of the constructs are standard expressions and we skip their presentations to focus on the more unusual operations.

L-Value Expressions

The next grammar is used to distinguish l-value expressions, i.e, expressions that can appear on the left-hand side of an assignment, from other expressions:

$$\begin{aligned} \textit{LValue} \quad ::= \quad & variable_{id} \\ & \mid \texttt{*}\ Expr \\ & \mid \textit{LValue}\,\texttt{.}\,\textit{member\_name}_{id}\ \textit{LValue} \\ & \mid \textit{LValue}\texttt{[}Expr\texttt{]} \end{aligned}$$

At its core, an l-value is either a variable, a dereference expression, or an array element. The grammar extends the set of l-values to member expressions and index expressions built from l-values.

According the syntax of l-value expressions, variables of type array yield l-value expressions: Our compiler supports copy of arrays to facilitate formal reasoning. In contrast, C and C++ do not support array assignments – C++ programmers must use vectors to achieve similar effects.

AddressOf Expressions

$$AddressOfExpr \quad ::= \quad \texttt{\&}\ Expr_0$$

The address-of operator & returns the address of an l-value expression.

Dereference Expressions

$$DerefExpr \quad ::= \quad \texttt{*}\ Expr_0$$

In a standard way, the dereference operator * returns the value at the memory location indicated by the operand.

ArrayOf Expressions

$$ArrayOfExpr \quad ::= \quad \texttt{array\_of(}size_{\mathbb{N}}, Expr_0\texttt{)}$$

An expression array_of(s, e_0) creates an array of s elements that are equal to e_0.

Concatenation Expressions

$$\textit{ConcatExpr} \;::=\; \texttt{concat(}Expr_1, Expr_0\texttt{)}$$

The expression `concat(`e_1, e_0`)` concatenates the two unsigned bit-vector e_1 and e_0 such that e_1 and e_0 represent the upper part and the lower part of the concatenation, respectively.

Bit-Extraction Expressions

$$\textit{ExtractBitExpr} \;::=\; \texttt{extractbit(}Expr_{src}, Expr_{index}\texttt{)}$$

The bit-extraction operator `extractbit` takes as parameter two bitvector expressions e_{src} and e_{index} and extracts the bit at the position indicated by e_{index} from e_{src}, e.g., expression `extractbit` e 0 extracts the first bit of the bit-vector expression `e`.

Bits-Extraction Expressions

$$ExtractBitsExpr \;::=\; \texttt{extractbits(}Expr_{src}, Expr_1, Expr_0\texttt{)}$$

The ternary bits-extraction operator is used to extract a subset of bits from a bit-vector expression. The first operand is a bit-vector. The second and the third operands are constant bit-vector expressions that specify the left- and right-hand index positions, respectively: `extractbits(`$e_{src}, 3, 1$`)` returns a bit-vector of width 3 that holds the value of the third, second, and first bits of e_{src}.

Member Expressions

$$\textit{MemberExpr} \;::=\; Expr_0\texttt{.}\ member_{id}$$

In the usual way, a member expression e_0 `.`*field* takes as parameter an object e_0 and extracts its member named *field*. The expression is ill-formed if *field* is not a member of e_0.

New Expressions

$$\textit{NewExpr} \;::=\; \texttt{new}\ \textit{Type}$$

The operator `new` is used for dynamic memory allocation. Expression *new* t returns a new object of type `t`.

Non-Deterministic Expressions

$$\textit{NonDetExpr} \;::=\; \texttt{nondet}\ \textit{Type}$$

Operator `nondet` returns a random values upon evaluation. Non-deterministic expressions are used to model external inputs such as files or user commands.

ArrayWith Expressions

$$\textit{ArrayWithExpr} \ ::= \ \texttt{array\_with(}\textit{Expr}, \textit{Expr}, \textit{Expr}\texttt{)}$$

ArrayWith expressions are used for model-checking purposes to express predicates over arrays in compact form. Let e_0 denote an expression array of type $t_0[s]$, where t_0 indicates the subtype of the array and `s` the size of the array. Additionally, let e_1 denote an integer expression and e_2, a value of type t_0. The expression `array_with(`e_0, e_1, e_2`)` represents the same value as array e_0 except that the value of the element at position e_1 is equal to e_2.

StructWith Expressions

$$\textit{StructWithExpr} \ ::= \ \texttt{struct\_with(}\textit{Expr}, \textit{member}_{id}, \textit{Expr}\texttt{)}$$

Let e_0 denote an expression of type structure with a member named *field* of data type t_1, and let e_1 denote of value of type t_1.

The expression `struct_with(`e_0,*field*,e_1`)` represents the same value as expression e_0 except that the value of *field* is equal to e_1.

3.3.3 Statements

$$\begin{aligned}\textit{Command} \ ::= \ & \textit{Expr} \ \texttt{:=} \ \textit{Expr} \\ & \mid \textit{Expr} \ \texttt{(} \ \{ \ \textit{Expr} \ \} \ \texttt{)} \\ & \mid \textit{Expr} \ := \ \textit{Expr} \ \texttt{(} \ \{ \ \textit{Expr} \ \} \ \texttt{)} \\ & \mid \texttt{assert} \ \textit{Expr} \\ & \mid \texttt{assume} \ \textit{Expr} \\ & \mid \texttt{goto} \ \textit{Expr} \ \textit{location}_{id}\end{aligned}$$

Our intermediate representation recognizes the traditional commands: assignments, function calls, assert statements, assume statements, and control statements that form the basic nodes of control-flow graphs.

Assignments

$$\textit{Assignment} \ ::= \ \textit{Expr} \ \texttt{:=} \ \textit{Expr}$$

In the standard way, the left-hand side of an assignment must be a l-value expression. As mentioned before, our intermediate representation supports copy of arrays.

Function Calls

$$\begin{aligned}\textit{FuncCall} \ ::= \ & \textit{Expr} \ \texttt{(} \ \{ \ \textit{Expr} \ \} \ \texttt{)} \\ & \mid \textit{Expr} \ := \ \textit{Expr} \ \texttt{(} \ \{ \ \textit{Expr} \ \} \ \texttt{)}\end{aligned}$$

Our intermediate representation considers expressions without side-effects. Function calls are thus expressed using function-call statements. The return value of a function can be directly assigned to a l-value expression.

Assert Statements

$$\textit{Assert} \;::=\; \texttt{assert}\ \textit{Expr}$$

An assert statement defines a safety property that must hold when the command is executed. The execution is stopped and an error is reported if the property is violated.

Assume Statements

$$\textit{Assume} \;::=\; \texttt{assume}\ \textit{Expr}$$

An assume statement defines a safety properties that must hold when the command is executed. In contrast to assert statements, the execution is stopped silently if the property is unsatisfied. Assume statements are exclusively used for formal verification purposes.

Control Statements

$$\textit{Goto} \;::=\; \texttt{goto}\ \textit{Expr}\ \textit{location}_{id}$$

Control statements are used to implement control flow. A goto instruction takes as parameter an boolean expression that represents the guard of the statement and a destination identifier that indicates the location where the execution shall jump if the guard evaluates to `true` – otherwise the execution shall continue with the next instruction.

3.4 Inheritance

Type inheritance is a high-level programming concept. Internally, the frontend considers structural equivalence, that is, a structure `A` *derives* from a structure `B` if the structure of `B` subsumes the structure of `A` – any field of `B` is also field of `A`. It is the duty of the typechecker to verify the concordance of types and to perform implicit conversions if necessary. Single inheritance provides a limited way of expressing relation among classes. In many cases, however, we wish to combine behaviors of different classes. C++ offers multiple inheritance for that purpose, and indeed, the implementation of SystemC is based on this mechanism. Our compiler handles multiple inheritance in a classic way. The next lines shows the declaration of three C++ structures `A1`, `A2`, and `B`:

```
struct A1 { bool a; };
struct A2 { int a; };
struct B: A1, A2 {bool b;};
```

These structures yield the following types in the symbol table after conversion:

Symbol Table	
Identifier	*Type*
A1	`struct{ bool` *A1_a* `}`
A2	`struct{ unsignedbv<32>` *A2_a* `}`
B	`struct{ bool` *A1_a*`, unsignedbv<32>` *A2_a*`, bool` *B2_b* `}`

Structure `B` inherits from `A1` and `A2`, and therefore, `B` contains a copy of the members of each of its base classes. The compiler automatically renames members using the identifier of the enclosing class to avoid name clashes, e.g., *"A1_"*. From the table above, `B` derives from both `A1` and `A2` as the fields *A1_a* and *A2_a* of `A1` and `A2` are both fields of `B`.

Virtual Inheritance In general, a new class inherits a copy of all the members of its parents. As shown in the previous example, the frontend renames members to avoid name clashes using class identifiers. This schema is applicable only if classes do not inherit multiple times from a same parent. In order to avoid duplicated copies of parents, the programmer can specify virtual bases that can be inherited several times – these virtual bases are then shared among other base types. We illustrate this mechanism with an example. The next lines shows the declaration of four C++ structures `A`, `B`, `C` and `D`:

```
struct A { bool a; };
struct B: virtual A { int b; };
struct C: virtual A {char c;}
struct D:B,C{bool d;};
```

Classes `B` and `C` both inherit from the virtual base class `A`, meaning that `A` can be shared between `B` and `C`. Subsequently, class `D` inherits from `B` and `C`. These structures yields the following types after conversion:

Symbol Table	
Identifier	*Type*
A	`struct{ bool A_a }`
B	`struct{ bool A_a, unsignedbv<32>` *B_b* `}`
C	`struct{ bool A_a, signedbv<8>` *C_c* `}`
D	`struct{ bool A_a, unsignedbv<32>` *B_b*`,signedbv<8> C_c, bool` *D_d* `}`

Note that member *A_a* appears only once in the definition of `D`. Therefore, *A_a* is well defined in `D`.

Program 3.2 Example with virtual functions

```
struct A {
  virtual int f(){return 0;};  // virtual function
};

struct B: A {
  void f(){return 1;}          // f() is virtual implicitly
  virtual void g(){return 2;} // virtual function
};

int main()
{
  A a;
  a.f();        // returns 0
  B b;
  b.f()         // returns 1;
  ((A&)b).f()  // returns 1;
};
```

3.5 Virtual Functions

A method is virtual if it is declared with the *virtual* keyword or if it overrides a virtual method from a base class, in which case the method is said to be virtual implicitly. C++ requires compilers to handle call to virtual methods in a dynamic way: The actual target of the call is chosen at runtime according to the final type of the object associated with the method. To achieve this goal, the compiler implicitly creates a virtual table for holding pointers to methods. Those tables are used to resolve function addresses at runtime. Each virtual function in a class has a corresponding entry in the virtual table for that class. Virtual tables are filled statically by the compiler. When necessary, the compiler adds references to virtual tables inside the body of a class. Those references are then set dynamically during the construction of the objects.

We use Program 3.2 as a running example to illustrate the mechanisms underneath virtual functions. The program defines two classes `A` and `B`. Class `A` declares a virtual function `f()`. Class `B` inherits from `A`, declares a virtual function `g()`, and overloads function `f()` – observe that the declaration of `f()` in `B` is implicitly virtual as `f()` is virtual in `A`. Subsequently, the main function instantiates two variables `a` and `b` of respective types `A` and `B` and then calls `a.f()`, `b.f()`, and `((A&)b).f()`. Note that the call `a.f()` returns zero, which means that the function `f()` from `A` is executed. In contrast, both expressions `b.f()` and `((A&)b).f()` return one, which means that the function `f()` from `B` is chosen, as `f()` is virtual and overloaded in `B`.

The next table shows the structures after conversion. Upon semantics analy-

sis, the compiler creates the virtual-table types *A_vt* and *B_vt* for holding references to the virtual methods. The compiler then adds to the body of the classes `A` and `B` the two pointers *A_vtptr* and *B_vtptr* for holding references to virtual tables.

Symbol Table	
Identifier	*Type*
A	`struct{` *A_vt** *A_vtptr* `}`
B	`struct{` *A_vt** *A_vtptr*, *B_vt** *B_vtptr* `}`
A_vt	`struct{ signedbv<32>` (*A**)* *A_vt_f* `}`
B_vt	`struct{ signedbv<32>` (*B**)* *B_vt_f*,
	`signedbv<32>` (*B**)* *B_vt_g* `}`

In addition, the next table shows the functions that are created during conversion. Symbol *A_f* denotes the method `f()` in `A`. Similarly, *B_f* and *B_g* denote the methods `f()` and `g()` in `B`, respectively. In addition to those methods, the compiler creates a function *B_f_A* implicitly. This function is identical to function *B_f* except that its first parameter is a pointer to an object of type `A` (instead of pointer to an object of type `B`). The first argument corresponds to the *this* parameter of the method and is converted at runtime to a pointer to elements of type `B`. The purpose of function *B_f_A* is to override the behavior *A_f* in `A`.

Symbol Table	
Identifier	*Type*
`A_f`	`unsignedbv<32>` (A*)
`B_f`	`unsignedbv<32>` (B*)
`B_g`	`unsignedbv<32>` (B*)
`B_f_A`	`unsignedbv<32>` (A*)

The compiler generates virtual tables statically. Those virtual tables are then shared among all object instances. In the previous example, the compiler creates three global virtual tables *vtAA*, *vtAB*, and *vtBB* with static storage duration:

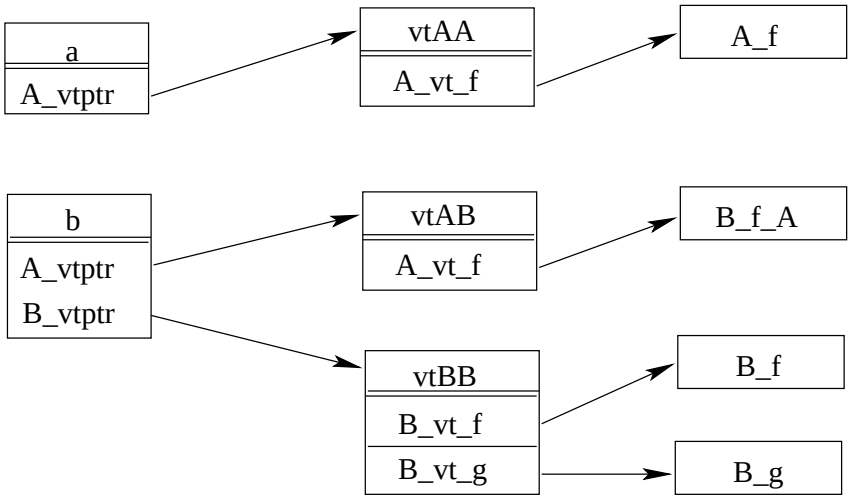
Symbol Table	
Identifier	*Type*
`vtAA`	*A_vt*
`vtAB`	*A_vt*
`vtBB`	*B_vt*

Virtual table `vtAA` is used for calls made in the context of class `A` if the final type of the object is `A`. Otherwise, virtual table `vtAB` is used if the call occurs in the context of `A` and if the final type of the object is `B`. Finally, virtual table `vtBB` is used if the call is made in the context of `B` and the final type of the object is `B`. Our compiler initializes those variables in the following way:

$$vtAA.A\_vt\_f := \&A\_f$$
$$vtAB.A\_vt\_f := \&A\_f\_B$$
$$vtBB.B\_vt\_f := \&B\_f$$
$$vtBB.B\_vt\_g := \&B\_g$$

Figure 3.4 summarizes the situation. The virtual table of `a` is set to `vtAA`. Variable `b` has two virtual tables. The first one is set to `vtAB` and is used to override the behavior of the base type of `b`. The second one is set to `vtBB`.

Figure 3.4 Layout of virtual tables for Program 3.2



Our compiler provides a complete support for virtual functions, multiple inheritance, and virtual bases. Additionally, we have a full implementation of function overloading, including operator overloading and virtual operators.

3.6 Templates

Generic programming is a key concept for writing efficient and reusable programs. In C++, a template is essentially a pattern for functions and classes that is parameterized and can be instantiated to produce ordinary functions and classes. The parameters of a pattern can be types or constant expressions whose values can be determined at compile time. As it was discovered after its creation, C++ templates provide a complete meta-programming language [Todd, 1996]. The syntax of C++ templates is generally regarded as awkward and difficult to read

and is poorly described in the standard. The implementation of templates is certainly the trickiest part of C++ compilers, which explains why for a long time compilers only had limited support for templates. Nevertheless, C++ templates are extremely powerful and widely popular. They are key to the implementation of many libraries, including SystemC and the Standard Template Library (STL) that provides standard generic containers and algorithms. In this section, we provide an overview of our implementation of templates noting that the frontend already supports a broad subset of templates that is large enough to analyse SystemC models and many programs that contain STL containers.

3.6.1 Template Classes

We begin the review with template classes, which are fully supported by our compiler. The next example defines a generic class `array` for arrays of fixed size. The template has the two parameters `T` and `C` for the subtype and size of the array, respectively. Subsequently, the code instantiate the template and creates the arrays a_int10 and a_bool4 for holding ten integers and four boolean values, respectively:

```
// template class
template <class T, int C>
class array {
  public:
   T& operator [] (int i) { return t[i] };
  private:
   T[C] t;
};

array<int,10> a_int10;  // template instantiation
array<bool,4> a_bool4;  // template instantiation
```

3.6.2 Template Functions

The next example defines a generic function `max` that takes as parameter two arguments and returns a reference to the greatest one. The unique template parameter `T` denotes the type of the arguments. Note that any template instantiation is ill-formed if no suitable operator exists to compare the arguments. The typechecker must then report an error.

```
// template function
template <class T>
T& max(T& t1, T& t2) { return t1 < t2 ? t1 : t2; }

max<int>(i,j);     // template instantiation
```

Our compiler supports template functions. In additions to template functions, our frontend can also handle template methods, which are illustrated in the following example:

```
class A
{
  public:
  // template method
  template <class T>
    void func(T t){v = t; };
  bool v;
};

A a;
a.set<int>(0); // template instantiation
```

In the example above, class `A` declares the method template *func*. An error is generated at compile time if the template parameter `T` cannot be implicitly converted to `bool`.

3.6.3 Template Specialization

The idea of template specialization is to override the default definition of a template to handle a particular type in a different way. For instance, the next example shows a possible generic type for arrays of fixed size:

```
template <class T, int C>
class array {
  public:
  T& operator [] (int i) { return t[i] };
  private:
   T[C] t;
};
```

When dealing with boolean values, a programmer might decide to pack boolean values more efficiently using integers. The next example illustrate such a case:

```
// Specialization for boolean arrays of width 32
template <>
class array<bool,32>
{
  public:
  bool get(int i) const {...}
  bool set(int i, bool b) const {...};
  private:
  unsigned t;
};
```

Our frontend currently supports template specialization for classes and functions. The implementation of SystemC relies on this mechanism in several places,e.g., to specialize the behavior of boolean signals and ports.

3.6.4 Future Work

As part of our work to build a full C++ compiler, we are working to extend our support for partial specialization of templates and automatic inference of template parameters:

```
// template function
template <class T>
T& max(T& t1, T& t2) { return t1 < t2 ? t1 : t2; }

// instantiation using automatic inference of template parameters
int i,j;
max(i,j);


template<class T>
class {
  T a; T& get(){ return a;}
};

// (partial) specialization
template<class T*>
class {
  T* pa; T& get(){ return *pa;}
};
```

At the current time, we overcome compilation issues related to partial specialization by adding specialized versions of the templates. In a similar way, we can overcome compilation issues related to automatic inference of template parameters by providing those parameters explicitly.

3.7 Code Resynthesis

After typechecking and conversion to a control-flow graph, our framework offers the possibility to re-synthesize a flat C++ model of our intermediate representation that can be compiled to native code using an external an compiler such as *g++*. This provides a flexible, and portable, way of generating binaries. Additionally, the C++ models that we generate contains the original file-location informations to simplify debugging. Similar compilation flows are used for languages such as Haskell [Jones et al., 1992] and Verilog.

4

Verification of C++/STL Programs

4.1 Introduction

C++ offers many useful features not provided by C, including support for object-oriented programming and *generic programming*, where general purpose algorithms or data structures can be applied to many types of data, with proper type-checking. Software model checking for C programs is widely recognized as providing real benefits for suitable programs, and is implemented by a number of tools [Ball and Rajamani, 2000a, Henzinger et al., 2002, Chaki et al., 2004, Clarke et al., 2004]. All previous efforts to model check C++ code are based on explicit-state exploration and execution of the program; we propose to extend the popular *predicate abstraction* framework [Graf and Saïdi, 1997, Ball and Rajamani, 2000a] to the verification of C++ programs using abstract data types.

We concentrate our efforts on uses of the Standard Template Library (STL) [Stepanov and Lee, 1994], which provides a clear example of an advantage over verifying C code. Use of interesting data structures in C typically involves direct pointer manipulation and "hand-crafted" approaches to even common structures such as lists. Considerable effort must be spent in directly abstracting pointer behavior, not a strong suit of typical predicate abstraction engines. In contrast, code using the STL makes the operations explicit at the level of the data structure — the STL has made the most difficult part of the abstraction trivial, e.g., by replacing a `for`-loop stepping through next pointers of a `struct` with a `for`-loop incrementing an STL iterator into a list variable. Liskov and Zilles noted that abstract data types (such as those provided by STL) allow *programmers* to abstract away from the implementation details of commonly used structures and concentrate on the task at hand [Liskov and Zilles, 1974]. We observe that abstract data types (ADTs) provide the same facility in abstraction for *verification tools*.

We verify the *use* of STL calls rather than behavior for any particular STL implementation. STL implementations are precisely the kind of pointer-manipulation intensive, optimized-for-efficiency code that is difficult to abstract. Choosing a particular implementation to verify would also be difficult. Additionally, the STL implementations are typically well-tested, and even subtle bugs are likely to be revealed given the large amount of code depending on correct behavior. Discovering errors in a pattern of STL calls is therefore more useful to C++ programmers

than verification of STL implementations. Ignoring the implementation details is simply following the underlying principle of using abstract data types. It is precisely the implementation details that make testing of STL code difficult: an incorrect use of the STL may, in fact, work in a particular implementation of the STL. However, this "correct" behavior will be both non-portable and likely to break if changes are made to the code, such as the ordering of structures in memory. This difficult-to-test, difficult-to-reproduce behavior, typical of pointer and memory errors, makes a strong case for verification that code *relies only on behavior guaranteed by the Standard Template Library's definition in the C++ language standard [ISO/IEC, 2003]*, which is precisely what we provide.

We therefore assume the correctness of STL *implementations*, and pursue the more fruitful (and more likely to catch errors) path of checking the correctness of STL use according to the guarantees of the C++ language definition [ISO/IEC, 2003]. This is analogous to checking the correctness of code using standard "hand-defined" data structures in C, a difficult task for many current software model checkers: we leverage the power of abstract data types to avoid the usual difficulties.

Our approach is to produce an operational model of the behavior guaranteed by the STL standard and apply predicate abstraction to a modified C++ program in which STL calls have been replaced by the operationally equivalent model. In particular, our verification tool, SATABS [Clarke et al., 2004], is a predicate abstraction-based model checker that handles a large subset of the C++ language, and our operational model is written in (a variation of) C++. The operational model is not an implementation of the Standard Template Library, as it makes use of non-executable features such as infinite arrays — supported by the logic of our model checker, but not realizable in compiled code. The C++ model checker handles STL code, once it has been rewritten using the operational model, with the same standard abstraction-refinement loop as is used for the rest of the program. We show that it suffices to verify correctness using the operational model by proving that the preconditions on operations in the model imply the preconditions guaranteed by the language definition for those operations, and the post-conditions given by the standard imply the strongest post-conditions for the operational model.

The contribution of this work is to extend the powerful predicate abstraction technique for software model checking to apply to C++ programs, and in particular to use an operational model and the principles of abstract data types to efficiently verify usage of the C++ Standard Template Library [Stepanov and Lee, 1994, ISO/IEC, 2003] in an implementation-independent manner.

An Example

Program 4.1 shows a toy program using the STL vector container. This program declares an STL vector, `a`. On line 9, the program pushes a value into `a` (`push_back` expands into a call to the vector's `insert` method). Line 10 de-

Program 4.1 example.cpp

```
4  int main () {
5    int x;
6    vector<int> a;
7
8    //a.reserve(5);
9    a.push_back(10);
10   vector<int>::const_iterator i = a.begin();
11   a.push_back(6);
12   x = (*i);
13 }
```

clares an STL iterator `i`, initially pointing to the first element of `a`. Line 11 pushes another value into `a`, and line 12 dereferences the iterator from line 10. We wish to consider the question: is this program memory-safe? Is the dereference on line 12 guaranteed to succeed?

The answer is no. Our tool produces a counterexample demonstrating that the iterator `i` may be invalid at line 12, after 4 iterations, using 17 predicates (Prog. 4.1).

Figure 4.1 example.cpp counterexample

```
$\ldots$
State 57 file example.cpp line 11 thread 0
-------------------------------------------
  c::main::1::a={ .size=1, .capacity=1,
  .version=2, .data={  } }

Violated property:
  file example.cpp line 12
  dereferencing of invalidated iterator
  a.version == i.version
```

The standard makes no guarantee about the initial *capacity* of a vector. When an insertion exceeds the current capacity, the vector must be expanded, which may *invalidate all iterators* into that vector. Uncommenting line 8 produces a memory-safe program: the standard guarantees that after a reserve call, the capacity of `a` will be at least the value reserved. Until vector size exceeds this limit, no expansions (and thus invalidations) will occur. The model checker produces an abstraction proving the memory-safety of the fixed `example.cpp` in 9 iterations, using 45 predicates.

The model checker produces the counterexample and proof via standard predicate-abstraction techniques, after replacing the STL operations on lines 8, 9, 10, and 11

with our operational semantics. For example, the call to `reserve` becomes:

```
if(s>capacity) {
  capacity=s;
  version++;
}
```

where `s` is the argument to reserve (5), and `capacity` and `version` are fields in our model of the vector container. The field `version` is critical: Iterators also have `version` fields, and we construct the operational model such that an iterator is guaranteed valid iff the `version` field of the iterator matches the `version` field of the container (as discussed in Section 4.3).

The `push_back` call expands, in part, into:

```
reserve(size+1);
size++;
```

with the `reserve` call expanded as shown above.

The 45 predicates used to prove correctness include $a.capacity >= 5$, $1 + a.size <= 5$ and $a.version == i.version$. Program 4.1 is incorrect because the condition $s > capacity$ in the `reserve` produced by the `push_back` cannot be shown to be false, resulting in a possible change of `version`. In the fixed program, the new size is provably less than or equal to the capacity of the vector. The erroneous program executes successfully on many platforms, demonstrating the difficulty of testing such implementation-dependent errors.

4.2 Axiomatic Semantics

Table 4.1 Axiomatization of Iterators

$\dfrac{}{c.begin() \overset{0}{\rightarrowtail} c}$	(it-begin)	$\dfrac{}{c.end() \overset{c.size}{\rightarrowtail} c}$	(it-end)
$\dfrac{it_1 \overset{i}{\rightarrowtail} c \wedge it_2 \overset{i}{\rightarrowtail} c}{it_1 = it_2}$	(it-eq)	$\dfrac{it_1 \overset{i}{\rightarrowtail} c \wedge it_2 \overset{j}{\rightarrowtail} c \wedge i \neq j}{it_1 \neq it_2}$	(it-neq)
$\dfrac{it \overset{i}{\rightarrowtail} c \wedge i < c.size}{it + 1 \overset{i+1}{\rightarrowtail} c}$	(it-inc)	$\dfrac{it \overset{i}{\rightarrowtail} c \wedge 0 < i}{it - 1 \overset{i-1}{\rightarrowtail} c}$	(it-dec)

The C++ standard defines the semantics of the STL informally using pre- and post-conditions — in potentially ambiguous English text that is not machine-readable. We axiomatically formalize the semantics of the standard sequential containers `list`, `vector`, and `deque`, providing a basis for correctness of an abstract implementation. The semantics of associative containers such as `map`,

Table 4.2 Rules for *iterator*

$\{$	$P \wedge it \overset{i}{\rightarrowtail} c \wedge 0 \leq i + j \leq c.size \}$ it+=j	(it-mut-inc)
$\{$	$P[it/it'] \wedge it \overset{i+j}{\rightarrowtail} c \}$	$c \in \mathcal{A}$
$\{$	$P \wedge it \overset{i}{\rightarrowtail} c \wedge i < c.size\}$ $* it := t$	(it-deref1)
$\{$	$P[c/c'] \wedge c_i = t \wedge$	$c \in \mathcal{A}$
	$\forall j \neq i \,.\, c_j = c'_j \wedge$	
	$\forall it_2, j \,.\, it_2 \overset{j}{\rightarrowtail} c' \Rightarrow it_2 \overset{j}{\rightarrowtail} c\}$	

`multimap`, `set` and `multiset` are defined in a similar manner (we omit their presentation). We define Hoare triples in the "forward"-style for the methods of the container classes. "Backward"-style axioms for the purpose of generating verification conditions can be derived using the consequence rule. Hoare-style axiomatizations of languages that permit aliasing are problematic [Hoare and Wirth, 1973, Bornat, 2000, Reynolds, 2002, Ishtiaq and O'Hearn, 2001]; we reduce the aliasing problem between iterators to aliasing between elements of an array.

As the constructors of the containers have trivial semantics (either creating an empty container, or copying an existing container), we omit their axiomatizations. Methods such as $push\_back()$ and $pop\_front()$ are syntactic sugar for $insert()$ and $erase()$. We therefore limit the presentation to $insert()$ and $erase()$. Furthermore, the standard defines several forms of $insert()$ and $erase()$ methods. As some of these variations can be implemented in terms of other versions, we present only one of each category — a minimal basis for formalization.

4.2.1 The Assertion Language

We distinguish three types of variables: we define the set of container variables $\mathcal{C}$, the set of integer variables $\mathcal{N}$, and the iterator variables $\mathcal{I}$. The set of variables is denoted by $\mathcal{V} = \mathcal{C} \dot{\cup} \mathcal{I} \dot{\cup} \mathcal{N}$. By convention, we assume $\{c, l, v\} \subset \mathcal{C}$, $\{i, j, n\} \subset \mathcal{N}$, and $\{it, it_1, it_2\} \subset \mathcal{I}$. We assume that the containers contain elements of some type `T`. We denote the set of variables of this type by $\mathcal{T}$, and by convention, $t \in \mathcal{T}$.

We distinguish two different kind of container variables: *active* and *inactive* containers. By convention, we denote active containers with unprimed variables, e.g., c, v, l, and inactive containers by primed variables, e.g., c', v', l'. Inactive container variables are used in post-conditions to denote the pre-state of containers. The set of active containers is denoted by $\mathcal{A} \subset \mathcal{C}$.

We define the syntax for integer expressions (*IntExpr*) in the usual manner:

$$\begin{aligned} \mathit{IntExpr} \;:=\; & \mathcal{N} \mid \mathbb{Z} \\ & \mid \mathcal{C}.\mathbf{size} \mid \mathcal{C}.\mathbf{capa} \\ & \mid \mathit{IntExpr}\ (+ \mid - \mid * \mid ...)\ \mathit{IntExpr} \end{aligned}$$

The expressions $c.size$ and $c.capa$ denote the size and the capacity of a container c, respectively. We define the following iterator expressions:

$$\begin{aligned} \mathit{ItExpr} \;:=\; & \mathcal{I} \mid \mathit{ItExpr}\ (+ \mid -)\ \mathit{IntExpr} \\ & \mid \mathcal{C}.\mathbf{begin} \mid \mathcal{C}.\mathbf{end} \end{aligned}$$

Note that expressions of iterator type used in the program may contain additional operators, e.g., the dereferencing operator defined below. These operators are not permitted in assertions. We define the following expressions of type `T`:

$$\mathit{TExpr} \;:=\; \mathcal{T} \mid \mathcal{C}_{\mathit{IntExpr}}$$

The expression c_i denotes the value of the i^{th} element of the container c.

Note that in order to avoid some substitution details in the following rules, we assume the expressions in commands to be variable expressions.

Assertions may relate integers, compare container elements and iterators, relate iterators to container elements, and may contain the usual Boolean connectives:

$$\begin{aligned} \mathit{Assert} \;:=\; & \mathit{IntExpr}\ (< \mid = \mid ...)\ \mathit{IntExpr} \\ & \mid \mathit{TExpr} = \mathit{TExpr} \mid \mathit{ItExpr} = \mathit{ItExpr} \\ & \mid \mathit{ItExpr} \overset{\mathit{IntExpr}}{\rightarrowtail} \mathcal{C} \\ & \mid \neg \mathit{Assert} \mid \mathit{Assert}\ (\vee \mid \wedge \mid ...)\ \mathit{Assert} \\ & \mid \forall v\ .\ \mathit{Assert} \mid \exists \mathit{var}\ .\ \mathit{Assert} \end{aligned}$$

By $it \overset{i}{\rightarrowtail} c$ we denote the fact that the iterator it points to the i^{th} element of the container c. As a special case, i may be equal to the number of elements in the container. In this case, we say that i points to the end of the container c. The operator $\overset{i}{\rightarrowtail} c$ is only defined for offsets $i \in \{0, \ldots, c.size\}$.

4.2.2 Iterators

We first formalize the concept of the *Iterator*, which is technically a pointer to an element inside of a container. Besides iterators, the C++ standard permits references to the elements inside a container. For all containers except `deque<T>`, references can be replaced trivially by iterators.

We postpone the discussion of how references to elements inside a `deque` are handled.

Table 4.1 shows the axiomatization of the semantics of the operations on iterators. Iterators are typically created using the $begin()$ and $end()$ methods of containers. This is axiomatized by the two schemata *it-begin* and *it-end*.

Table 4.3 Basic Rules for Sequential Containers

$$\{\ P \wedge it_1 \overset{i}{\rightarrowtail} c\}\ it_2 := c.insert(it_1, t) \qquad \text{(seq-ins)}$$
$$\{\ P[c/c'][it_2/it_2'] \wedge i' = i[c/c'] \wedge$$
$$it_2 \overset{i'}{\rightarrowtail} c \wedge c.size = c'.size + 1 \wedge c_{i'} = t \wedge$$
$$\forall j < i' \,.\, c_j = c'_j \wedge$$
$$\forall j \geq i' \,.\, c_{j+1} = c'_j\ \}$$

$$\{\ P \wedge it_1 \overset{i}{\rightarrowtail} c \wedge i < c.size\}\ it_2 := c.erase(it_1) \qquad \text{(seq-era)}$$
$$\{\ P[c/c'][it_2/it_2'] \wedge i' = i[c/c'] \wedge$$
$$it_2 \overset{i'}{\rightarrowtail} c \wedge c.size = c'.size - 1 \wedge$$
$$\forall j < i' \,.\, c_j = c'_j \wedge$$
$$\forall j > i' \,.\, c_{j-1} = c'_j\ \}$$

Two iterators that point to the same location are equal (schema *it-eq*). To argue that two iterators are not equal it is necessary to show that they point to two different positions inside the same container (schema *it-neq*).

All containers permit incrementing and decrementing an iterator. If *it* points to the position i inside container c, then $it + 1$ points to the position $i + 1$ (schema *it-inc*). Note that $it + 1$ may be $c.end()$. Similarly, if *it* points to the position i and i is greater than zero, then $it - 1$ points to the position $i - 1$ (schema *it-deq*).

In addition to the previous axiom schemata, we provide the semantics of the mutation of iterators and one dereferencing command in Table 4.2.

4.2.3 Sequential Containers

The sequential containers `list`, `vector` and `deque` conform to a common basic semantics described by the rules *seq-ins* and *seq-era* given in Table 4.3.

Let c denote an instance of a sequential container with elements of type `T`. The $insert()$ method takes an iterator it_1 and a reference to an object of type `T` as arguments. As a pre-condition, it_1 must point to an element in c or be equal to $c.end()$. The post-condition guarantees that it_2 points to the newly inserted element.

The $erase()$ method removes the element pointed to by the iterator it_1 from the container c. The post-condition guarantees that the iterator it_2 points to the position in the sequence that was just beyond the erased element. The post-conditions of $insert()$ and $erase()$ for the validity of iterators depend on the particular container type, and are formalized in the following.

The `list` Container

Table 4.4 shows the additional rules for lists. Let l be an active instance of

Table 4.4 Additional Rules for `list`

$\{ \quad P \wedge it_1 \overset{i}{\rightarrowtail} l\}\ it_2 := l.insert(it_1, t)$ (lst-ins)
$\{ \quad P[l/l'][it_2/it_2'] \wedge i' = i[l/l'] \wedge$
$\forall it, j < i' \,.\, it \overset{j}{\rightarrowtail} l' \Rightarrow it \overset{j}{\rightarrowtail} l \wedge$
$\forall it, j \geq i' \,.\, it \overset{j}{\rightarrowtail} l' \Rightarrow it \overset{j+1}{\rightarrowtail} l \}$

$\{ \quad P \wedge it_1 \overset{i}{\rightarrowtail} l \wedge i < l.size\}\ it_2 := l.erase(it_1)$ (lst-era)
$\{ \quad P[l/l'][it_2/it_2'] \wedge i' = i[l/l'] \wedge$
$\forall it, j < i' \,.\, it \overset{j}{\rightarrowtail} l' \Rightarrow it \overset{j}{\rightarrowtail} l \wedge$
$\forall it, j > i' \,.\, it \overset{j}{\rightarrowtail} l' \Rightarrow it \overset{j-1}{\rightarrowtail} l \}$

`list<T>`. The $insert()$ method takes an iterator it_1 and a reference to an object of type `T`. As a pre-condition, it_1 must point to an element of l or be equal to $l.end()$. The post-condition guarantees that an iterator valid in the pre-state is also valid in the post-state.

The $erase()$ method removes the element pointed to by the iterator it_1 from the list l. The post-condition provides no guarantee about the validity of the iterators that were pointing to the erased element, but any other iterators are not affected by the removal. Note that the post-condition does not guarantee that iterators pointing to the erased element are invalid; however, previous guarantees about the validity of such iterators cannot carry over from the pre-condition, as the container is renamed. Thus, no conclusions can be made about the validity or invalidity of such iterators in the post-state.

The `vector` Container

Table 4.5 shows the additional rules for vectors. Let v be an instance of `vector<T>`. A vector v has a capacity $v.capa$ that corresponds to the number of elements v can hold without having to reallocate its content. Therefore, there are two different rules for the $insert()$ method. If no reallocation occurs (schema *vec-ins1*), the post-condition guarantees that the iterators pointing before the inserted element are still valid. Otherwise (schema *vec-ins2*), no guarantee is provided in the post-state about the validity of the iterators that are pointing into v in the pre-state.

The $erase()$ method removes the element contained in the vector v and pointed to by the iterator it_1. The post-condition does not provide any guarantees about the validity of the iterators that were pointing to or beyond the erased element. The validity of other iterators is not affected by the removal.

The $reserve()$ method adjusts the capacity of the vector. After its invocation, the capacity of the vector is greater or equal to the argument n. If the capacity in the pre-state is less than n, a reallocation occurs and the capacity is increased. Otherwise, the invocation has no effect.

Table 4.5 Additional Rules for `vector`

$$\{\ P \wedge it_1 \overset{i}{\rightarrowtail} v \wedge v.size < v.capa\}\ it_2 := v.insert(it_1, t) \quad \text{(vec-ins1)}$$
$$\{\ P[v/v'][it_2/it_2'] \wedge i' = i[v/v'] \wedge v.capa = v'.capa \wedge \forall it, j < i' \,.\, it \overset{j}{\rightarrowtail} v' \Rightarrow it \overset{j}{\rightarrowtail} v\}$$

$$\{\ P \wedge it_1 \overset{i}{\rightarrowtail} v \wedge v.size = v.capa\}\ it_2 := v.insert(it_1, t) \quad \text{(vec-ins2)}$$
$$\{\ P[v/v'][it_2/it_2'] \wedge v.capa \geq v.size\}$$

$$\{\ P \wedge it_1 \overset{i}{\rightarrowtail} v \wedge i < v.size\}\ it_2 := v.erase(it_1) \quad \text{(vec-era)}$$
$$\{\ P[v/v'][it_2/it_2'] \wedge i' = i[v/v'] \wedge v.capa = v'.capa \wedge \forall it, j < i' \,.\, it \overset{j}{\rightarrowtail} v' \Rightarrow it \overset{j}{\rightarrowtail} v\}$$

$$\{\ P \wedge n \leq v.capa\}\ v.reserve(n)\ \{P\} \quad \text{(vec-res1)}$$

$$\{\ P \wedge n > v.capa\}\ v.reserve(n) \quad \text{(vec-res2)}$$
$$\{\ P[v/v'] \wedge v.size = v'.size \wedge v.capa \geq n \wedge \forall j \,.\, v_j = v_j'\ \}$$

The formal definitions provided here are adapted from the informal English text of the ISO standard. For example, schemata *vec-res1* and *vec-res2* are equivalent to "Reallocation happens at this point if and only if the current capacity is less than the argument of `reserve()`" [ISO/IEC, 2003] and *vec-era* is a formalization of "Invalidates all the iterators and references after the point of the erase" [ISO/IEC, 2003].

The `deque` Container

The deque is a container for which insert and erase operations at either end of the sequence are optimized. The deque differs from other containers. The validity of the iterators and references to the elements in the sequence do not follow the same policy. For instance, an insert at either end of a deque invalidates all the iterators but has no effect on the references. Therefore, we have to distinguish references from iterators.

The rule *dqe-ins* describes the effects of an insertion at either end of a deque. Let d be an instance of `deque<T>`. The pre-condition asserts that the iterator it_1 points either to the first element of d or to its end. The post-condition guarantees that the validity of the references *ref* do not change. It provides no guarantee about the validity of the iterators of d. An insertion in the middle of a deque invalidates both the references and the iterators. Thus, there is no specific rule for this case.

The $erase()$ method removes the element pointed to by the iterator it_1 from

Table 4.6 Additional Rules for `deque`

$$\{\ P \wedge it_1 \overset{i}{\rightarrowtail} d \wedge (i = d.size \vee i = 0)\}\ it_2 := d.insert(it_1, t) \qquad \text{(dqe-ins)}$$
$$\{\ P[d/d'][it_2/it_2'] \wedge i' = i[d/d'] \wedge$$
$$\forall ref, j < i' \,.\, ref \overset{j}{\rightarrowtail} d' \Rightarrow ref \overset{j}{\rightarrowtail} d \wedge$$
$$\forall ref, j \geq i' \,.\, ref \overset{j}{\rightarrowtail} d' \Rightarrow ref \overset{j+1}{\rightarrowtail} d\}$$

$$\{\ P \wedge it_1 \overset{i}{\rightarrowtail} d \wedge (i = d.size - 1 \vee i = 0)\}\ it_2 := d.erase(it_1) \qquad \text{(dqe-era)}$$
$$\{\ P[d/d'][it_2/it_2'] \wedge i' = i[d/d'] \wedge$$
$$\forall it, j < i' \,.\, it \overset{j}{\rightarrowtail} d' \Rightarrow it \overset{j}{\rightarrowtail} d \wedge$$
$$\forall it, j > i' \,.\, it \overset{j}{\rightarrowtail} d' \Rightarrow it \overset{j-1}{\rightarrowtail} d \wedge$$
$$\forall ref, j < i' \,.\, ref \overset{j}{\rightarrowtail} d' \Rightarrow ref \overset{j}{\rightarrowtail} d \wedge$$
$$\forall ref, j > i' \,.\, ref \overset{j}{\rightarrowtail} d' \Rightarrow ref \overset{j-1}{\rightarrowtail} d\ \}$$

the deque d. If the element is at either end of d, then the operation has no effect on the validity of the iterators and references that were not pointing to the erased element (rule *dqe-era*). A removal of an element in the middle of a deque invalidates both the references and the iterators.

The `map` Container

A `map<K,T, ≺ >` associates unique keys of type K to values of type T. The template argument $\prec$ is a predicate function that must induce a strict weak ordering relation on the elements of K. The equivalence of the keys noted $\cong$ is defined as follows:

$$k_1 \cong k_2 \Leftrightarrow \neg(k_1 \prec k_2) \wedge \neg(k_2 \prec k_1)$$

The $insert()$ method described here takes an argument $t \in K \times T$ (Table 4.7). The first component is denoted by $t.first$ and corresponds to a key. The second component is denoted by $t.second$ and corresponds to the value the key is mapped to. The tuple t is inserted into the map m if and only if no key is equivalent to $t.first$. The returned value is a pair of an iterator and a Boolean value. Its second component is true if and only if t is inserted. In this case, the iterator $p.first$ points to the newly inserted tuple t. The post-condition guarantees that the validity of the iterators is not affected by the insertion.

The $erase()$ method removes the tuple pointed to by the argument it from the map. The post-condition does not provide any guarantees about the new value of the iterators that were pointing to the erased element. The validity of other iterators is not affected by the removal.

Table 4.7 Rules for `map`

$$
\begin{array}{ll}
\{ & P \wedge \forall i \,.\, \neg m_i.first \cong t.\textit{first}\}\ p := m.insert(t) \\
\{ & P[m/m'][p/p'] \wedge \\
 & \exists i \leq m'.size \,.\, \wedge \\
 & \forall j < i \,.\, m'_j.\textit{first} \prec t.\textit{first} \wedge \\
 & \forall j \geq i \,.\, t.\textit{first} \prec m'_j.\textit{first} \wedge \\
 & p.\textit{first} \overset{m}{\rightarrowtail} i \wedge p.second = true \wedge m_i = t \wedge m.size = m'.size + 1 \wedge \\
 & \forall it, j < i \,.\, it \overset{j}{\rightarrowtail} m' \Rightarrow it \overset{j}{\rightarrowtail} m \wedge \\
 & \forall it, j \geq i \,.\, it \overset{j}{\rightarrowtail} m' \Rightarrow it \overset{j+1}{\rightarrowtail} m \wedge \\
 & \forall j < i \,.\, m_j = m'_j \wedge \\
 & \forall j \geq i \,.\, m_{j+1} = m'_j \,\} \\
 & \\
\{ & P \wedge m_i.\textit{first} \cong t.\textit{first}\}\ p := m.insert(t) \\
\{ & P[p/p'] \wedge p.second = false \,\} \\
 & \\
\{ & P \wedge it \overset{i}{\rightarrowtail} m \wedge i < m.size\}\ m.erase(it) \\
\{ & P[m/m'] \wedge i' = i[m/m'] \wedge m.size = m'.size - 1 \wedge \\
 & \forall it_2, j < i' \,.\, it_2 \overset{j}{\rightarrowtail} m' \Rightarrow it_2 \overset{j}{\rightarrowtail} m \wedge \\
 & \forall it_2, j > i' \,.\, it_2 \overset{j-1}{\rightarrowtail} m' \Rightarrow it_2 \overset{j}{\rightarrowtail} m \wedge \\
 & \forall j < i \,.\, m_j = m'_j \wedge \\
 & \forall j > i \,.\, m_{j-1} = m'_j \,\} \\
 & \\
\{ & P \wedge m_i.first \cong k\}\ it := m.find(k)\{P[it/it'] \wedge it \overset{i}{\rightarrowtail} m \,\} \\
 & \\
\{ & P \wedge \forall i \,.\, \neg m_i.first \cong k\}\ it := m.find(k)\{P[it/it'] \wedge it \overset{m.size}{\rightarrowtail} m \,\}
\end{array}
$$

The $erase()$ and $insert()$ methods of the `multimap`, `set` and `multiset` containers behave in the same way as the ones of map: the insertion of an element does not invalidate iterators; erasing of an element only invalidates the iterators pointing to that element.

The $find()$ method searches for a tuple of map m with a key equivalent to the argument k. If such a tuple exists, then the returned value is an iterator pointing to it. Otherwise, the returned iterator points to the end of the map. Since the definition of the semantics of other methods such as `lower_bound()`, `upper_bound()`, and `equal_range()` follows a similar pattern, we skip their presentation.

4.3 An Operational Model for the STL

Table 4.8 The definitions of the functions and sets of the operational model.

It	$=$	$\{V_{Cont} \cup \bot\} \times \mathbb{N}_0 \times \mathbb{N}_0$	(*vcont*, *offset*, *version*)	$\in It$
Cont	$=$	$(\mathbb{N}_0 \to T) \times (\mathbb{N}_0 \to \mathbb{N}_0)$ $\times \mathbb{N}_0 \times \mathbb{N}_0$	(*data*, *version*, *size*, *capa*)	$\in Cont$
Σ	$=$	$(V_Z \to \mathbb{Z}) \times (V_{It} \to It) \times$ $(V_{Cont} \to Cont) \times (V_T \to T)$	($\sigma_{\mathbb{Z}}$, σ_{It}, σ_{Cont}, σ_T)	$\in \Sigma$
$[\![ ]\!]_X$	:	$(X\ Expression \times \Sigma) \to X$		
$[\![ ]\!]_{cmd}$	:	$(Command \times \Sigma) \to \Sigma$		

In order to verify that a program using the STL obeys the pre-conditions of the methods of the containers and iterators as formalized above, we use an operational model. The operational model assumes that variables with an array type of infinite size can be declared, i.e., mappings from $\mathbb{N}_0$ into some arbitrary domain. Note that the operational model is therefore optimized for verification purposes, and is not actually executable.

The model is expressed using operational semantics. In the following, we use X and Y as meta types. Let It and $Cont$ denote respectively the set of iterators and container values. The set of variables of a specific type X is written V_X. The set of states is denoted by Σ. A state s is a tuple of functions from variables to values. The symbol $[\![]\!]_X$ denotes a function from states and expressions to values of type X. We write $[\![c]\!]_X s$ to represent the evaluation of the expression c in the state s. The symbol $[\![]\!]_{cmd}$ denotes a function from states and commands to states. We employ the standard notation $s[\![c]\!]_{cmd}s'$ to signify that executing the command c in the state s yields the state s'. The definitions of the previous sets and functions are shown in Table 4.8. Furthermore, note that containers have a field *capa*, which is only used if the container is a vector.

We relate the sets of the axiomatic model and the sets of the operational model in the following way: $V_{It} \subset \mathcal{I}$, $V_{Cont} = \mathcal{A}$, $V_T \subset \mathcal{T}$ and $V_{\mathbb{Z}} \subset \mathcal{N}$.

Table 4.9 The Operational Semantics of the Expressions.

$[\![x]\!]_X s$	$= s.\sigma_X(x)$	(expr-var)
$[\![\mathring{c}]\!]_{V_{Cont}} s$	$= c$	(expr-vcont)
$[\![e_0\,?\,e_1\ :\ e_2]\!]_X s$	$= [\![e_0]\!]_{bool} s\,?\,[\![e_1]\!]_X s\ :\ [\![e_2]\!]_X s$	(expr-ite)
$[\![\lambda x.\, e]\!]_{X \to Y} s$	$= \lambda x'.\, [\![e]\!]_Y s\langle x,\, x'\rangle$	(expr-λ)
$[\![e_0(e_1)]\!]_Y s$	$= [\![e_0]\!]_{X \to Y} s([\![e_1]\!]_X s)$	(expr-func)
$[\![c_e]\!]_T s$	$= [\![c.data(e)]\!]_T s$	(expr-at)

Let x denote a variable, e an expression and c a container variable. Table 4.9 shows the meaning of some of the expressions of the language. The semantics of the trivial expressions are skipped. For the sake of conciseness, the language used for the operational model has new constructs such as the ones found in *expr-ite*,

expr-vcont or *expr-func*. Note that in *expr-vcont*, $\mathring{c}$ denotes the variable c itself and not its value.

A version number is associated with each offset of the data array of a container. The *version* and *data* arrays can be seen as functions $\mathbb{N}_0$ to respectively $\mathbb{N}_0$ and T. Each iterator has a field called $version$, which is a number. The field $vcont \in V_{Cont} \dot{\cup} \{\bot\}$ identifies the container into which an iterator points, or is $\bot$ in the case of an iterator that has not yet been assigned to.

Our operational model maintains the following invariant: An iterator it points into a container c if and only if the version of the iterator matches the version of the element it points to:

$$s \vDash it \overset{i}{\rightarrowtail} c \iff s \vDash \begin{array}{l} it.vcont = \mathring{c} \wedge it.\mathit{offset} = i \wedge \\ it.version = c.version(i) \end{array}$$

(ass-ptsto)

We use $s\langle a,\ x\rangle$ to denote the state equal to s except that the value of the variable a is x. If a has a field named b, then $s\langle a.b,\ x\rangle$ denotes the state that is equal to s except that $a.b$ is x. For arrays, we use the notation $s\langle c_i,\ t\rangle$ to refer to the state equal to s except that the i^{th} element of c is equal to t. For convenience, we use $s\langle..|a_i, x_i|..\rangle$ to denote the state obtained from s by simultaneously substituting all a_i by x_i.

We translate the axiomatic semantics of the iterators into an operational model (Table 4.10). Note that **I** denotes a macro used for shortening the formulas.

To argue that incrementing an iterator it yields a state that is equal to the previous state except that both the offset and the version of it are updated, it is necessary to show that it points inside of a container c and that incrementing its offset produces a position that is still in the bounds of c (*opm-it-muc-inc*).

The Operational Semantics of Vectors

We present the operational semantics of the insertion and the removal of an element of a vector in Table 4.11. The rule *opm-vec-ins* describes the operational semantics of the command $it_2 := v.\mathit{insert}(it_1, t)$; by means of the program I_{vec} given in Table 4.1. The program I_{vec} inserts the value t into the vector c just before the position pointed to by the iterator it_1. The iterator it_2 is then set to the newly inserted element. The validity of the iterators depends on the capacity of the vector.

The rule *opm-vec-era* describes the effect of removing an element form a vector by means of the program E_{vec}, given in Alg. 4.2. Note that only the iterators that point beyond the erased element are invalidated.

Table 4.10 The Operational Semantics of Iterators

$$\frac{s \vDash it \overset{\mathbf{I}}{\rightarrowtail} c \wedge 0 \leq \mathbf{I} + j < c.size}{s[\![it+=j]\!]_{cmd} s\langle \mathbf{I},\ [\![\mathbf{I}+j]\!]_{\mathbb{N}_0} s \mid \mathbf{V},\ [\![c.version(\mathbf{I}+j)]\!]_{\mathbb{N}_0} s\rangle}$$

(opm-it-mut-inc)

$\mathbf{I} = it.offset$
$\mathbf{V} = it.version$

$$\frac{s \vDash it \overset{\mathbf{I}}{\rightarrowtail} c \wedge \mathbf{I} < c.size}{s[\![*it := t]\!]_{cmd} s\langle c_{[\![\mathbf{I}]\!]_{\mathbb{N}_0} s},\ [\![t]\!]_T s\rangle}$$

(opm-it-deref1)

$\mathbf{I} = it.offset$

$$\frac{s \vDash it \overset{\mathbf{I}}{\rightarrowtail} c \wedge \mathbf{I} < c.size}{s[\![t := *it]\!]_{cmd} s\langle t,\ [\![c_{\mathbf{I}}]\!]_T s\rangle}$$

(opm-it-deref2)

$\mathbf{I} = it.offset$

$$\frac{s \vDash it_2 \overset{it_2.offset}{\longmapsto} c \wedge it_1 \overset{it_1.offset}{\longmapsto} c}{s[\![i := it_2 - it_1]\!]_{cmd} s\langle i,\ it_2.offset - it_1.offset\rangle}$$

(opm-it-dist)

Table 4.11 The Operational Semantics of Vectors

$$\frac{s[\![I_{vec}]\!]_{cmd} s' \wedge s \vDash it_1 \overset{\mathbf{I}}{\rightarrowtail} v}{s[\![it_2 := v.insert(it_1, t)]\!]_{cmd} s'}$$

(opm-vec-ins)

$\mathbf{I} = it_1.offset$

$$\frac{s[\![E_{vec}]\!]_{cmd} s' \wedge s \vDash it_1 \overset{\mathbf{I}}{\rightarrowtail} v \wedge \mathbf{I} < v.size}{s[\![it_2 := v.erase(it_1)]\!]_{cmd} s'}$$

(opm-vec-era)

$\mathbf{I} = it_1.offset$

The Operational Semantics of a List

Alg. 4.3 shows a program that inserts a value into a list. Note that due to the universal quantifier in the post-condition of the rule *lst-ins*, every iterator variable may need to be updated. In order to overcome the issues that arise with the use of universal quantifiers we propose an over-approximation. Note that we require the over-approximation to be sound, i.e., the checker does not incorrectly report that a program is correct.

One possible over-approximation for a list consists in keeping valid only the iterators whose offsets are not affected. The checker may as a result report spurious counterexamples, but the approximation may be sufficient for proving some properties. We hope to implement a refinement procedure for ruling out some spurious counterexamples introduced by the over-approximation.

The translation of the remaining axiomatic rules for STL into our operational semantics can be carried out in the same manner. We therefore omit their presentation. The translation of the operational model into a C++ library that is used for

Algorithm 4.1 The program I_{vec} inserts into the vector v the element t.

```
1:  procedure insert_vec
2:      v.size := v.size + 1;
3:      v.data := λ i. i < it1.offset ? v.data(i) : i = it1.offset ? t : v.data(i-1) ;
4:      if v.size ≤ v.capa then
5:          v.version := λ i. i < it1.offset ? v.version(i) : v.version(i)+1;
6:      else
7:          v.version := λ i. v.version(i)+1;
8:              ▷ NonDetVal stands for a non-deterministic non-zero positive value
9:          v.capa := v.capa + NonDetVal;
10:     it2 := (v̊, it1.offset, v.version(it1.offset) );
```

Algorithm 4.2 The program E_{vec} removes from the vector v the element pointed to by it_1.

```
1:  procedure erase_vec
2:      v.size := v.size - 1 ;
3:      v.version := λ i. i < it1.offset ? v.version(i) : v.version(i)+1;
4:      v.data := λ i. i < it1.offset ? v.data(i) : v.data(i+1);
5:      it2 := (v̊, it1.offset, v.version(it1.offset));
```

model checking an application is straight-forward, though an over-approximation is necessary to handle the quantifiers.

Depending on the property being checked, it may even be sufficient to adopt a coarser over-approximation that has the benefit of making the verification more efficient. Instead of considering an array of version numbers, one can associate a single version number with an entire container. Every time the version of a container is increased, all the iterators pointing to it are invalidated.

Correctness

One can show correctness of the operational model with respect to the formal semantics given in Section 4.2. The following three claims are shown for each of the methods:

1. The invariant is maintained *(ass-ptsto)*,

2. the pre-condition of the operational model is at least as strong as the pre-condition required by the standard, and

3. the post-condition of the operational model is at most as strong as the post-condition guaranteed by the standard.

Algorithm 4.3 The program I_{lst} inserts in the list l the element t before the position pointed to by iterator it_1.

```
1: procedure insert_lst
2:     l.size := l.size + 1 ;
3:     l.data := λ i. i < it1.offset ? l.data(i) : ( i = it1.offset ? t : l.data(i-1) );
4:     for all var ∈ V_It/{it1, it2} do
5:         var.offset := var.offset < it1.offset ? var.offset : var.offset+1;
6:         var.version := l.version(var.offset);
7:     it2 := (l̊, it1.offset, l.version(it1.offset));
8:     it1.offset := it1.offset + 1;
9:     it1.version := l.version(it1.offset);
```

Algorithm 4.4 The program I_{lst2} inserts into the list l the element t. Note that this is an over-approximation

```
1: procedure insert_lst2
2:     l.size := l.size +1 ;
3:     l.data := λ i. i < it1.offset ? l.data(i) : i = it1.offset ? t : l.data(i-1) ;
4:     l.version := λ i. i < it1.offset ? l.version(i) : l.version(i)+1;
5:     it2 := (l̊, it1.offset, l.version(offset) );
```

4.4 Experimental Results

Our implementation is based on SATABS. SATABS implements counterexample-guided abstraction refinement, and, as a distinguishing feature, uses a SAT-solver to compute the abstract model [Clarke et al., 2004]. It is therefore sound with respect to the bit-vector semantics of C and C++. Besides predicate abstraction, SATABS also implements static analysis with two powerful abstract domains: a 'may' analysis, mostly used for pointers, and a 'must' analysis, mostly used for integer variables.

The operational model uses an unbounded array in order to store the container elements. We therefore extend SATABS in order to support unbounded arrays in the predicates and in the transition relation. We first reduce the formula with array operations to a formula over uninterpreted functions. This formula is then reduced to bit-vector logic by means of Ackermann's reduction. This is an eager reduction, and is therefore similar to the implementation in UCLID [Bryant et al., 2002].

Our front-end to SATABS supports a large subset of the C++ language. We currently lack support for template partial specialization and automatic inference of template parameter. We do implement template classes, overloading and virtual methods. We refer the reader to Chapter 3.6.

We use the source code of MiniSAT as a benchmark for our technique. Min-

iSAT is "a minimalistic, open-source SAT solver," recognized in the SAT 2005 competition as one of the most efficient SAT solvers available [Eén and Sörensson, 2003]. The importance of effective SAT solvers to many applications, particularly verification, is well known, and MiniSAT is a popular base for cutting-edge research in Boolean satisfiability.

A number of variants of MiniSAT are available. The standard release is written in C++. One of these variants replaces the custom made dynamic vector used in the main releases with the vector class provided by the C++ Standard Template Library. The MiniSAT code is hand-crafted for high performance, and makes use of templates, references, and operator overloading.

We obtain a total of 299 non-trivial safety properties for the MiniSAT code, out of which 272 are due to the pre-conditions of our operational version of the vector class. The benchmarks were performed on a Linux machine with a 2.8 GHz Intel Xeon processor. Within 13s (including parsing), the static analysis is able to prove 150 of the properties. The remaining ones are passed to the predicate-abstraction engine. We use a limit of 20 refinement iterations. The times are split up into the time taken by the abstraction, model checking, simulation, and refinement.

		Avg. Time (s)					
	No.	Total	Abs.	Mc.	Sim.	Ref.	It.
CE	5	324.8	18.3	276.7	5.6	23.8	6.4
Success	229	52.8	7.6	38.1	1.5	5.6	2.6
Failed	65	420.6	14.8	331.8	29.1	44.9	20.0

We were able to prove 229 properties (76%) in an average of about one minute each, and obtained counterexamples for 5 properties. The counterexamples are due to imprecise modeling of the environment. As an example, MiniSAT contains an assertion that compares an integer read from a file with a constant. For 65 properties, the iteration limit was exceeded. Those cases were mostly due to vector indexing by values taken from non-STL dynamic memory. The model checker and the operational model of STL are available to other researchers for experimentation[1].

Limitations and Future Work

Proving full memory-safety for C++ programs is, of course, no easier than for C programs. Most real code mixes STL containers amenable to our abstraction with more "C-like" allocation and memory management. As discussed in the introduction, finding a good abstraction for low-level memory management is far more difficult than abstracting usage of a well defined ADT. Program 4.2 shows a portion of the MiniSAT code mixing STL usage with a C-style allocation involving an empty declaration (for `data`), `malloc`, and a cast into a float. The ADT-based approach used to show safety for STL constructs is not applicable,

[1] `http://www.cprover.org/satabs/` and `http://www.cprover.org/stl/`.

Program 4.2 Problematic code from MiniSAT

```
class Clause {
 uint     size_learnt;
 Lit      data[0];
public:
 // NOTE: This constructor cannot be used directly
 // (doesn't allocate enough memory).
 Clause(bool learnt, const vector<Lit>& ps) {
     size\_learnt = (ps.size() << 1) | (int)learnt;
     for (int i = 0; i < (int)(ps.size()); i++) data[i] = ps[i];
     if (learnt) activity() = 0; }

 // -- use this function instead:
 friend Clause* Clause_new(bool learnt, const vector<Lit>& ps) {
     assert(sizeof(Lit)      == sizeof(uint));
     assert(sizeof(float)    == sizeof(uint));
     void*    mem = xmalloc<char>(sizeof(Clause) +
              sizeof(uint)*(ps.size() + (int)learn
     return new (mem) Clause(learnt, ps);
 }

 int     size() const { return size_learnt >> 1; }
 bool    learnt() const { return size_learnt & 1; }
 Lit     operator[] (int i) const { return data[i]; }
 Lit&    operator[] (int i)       { return data[i]; }
 float& activity() const { return *((float*)\&data[size()]);}
};
```

and so we are only able to show memory-safety for the STL usage, not the entire program. This highlights the value of ADTs for verification tools. Code directly manipulating memory is more difficult to abstract for both a programmer seeking code understanding and a model checker seeking a correctness proof than code making use of the STL.

ADTs and Invariants

A problematic pattern appearing in MiniSAT is the use of a literal taken from a clause vector as an index into another vector, as in this code fragment:

```
Lit q = c[j];
if (!seen[var(q)] && level[var(q)] > 0) {
```

where `c` originally derives from a clause database. In order to prove safety for the accesses to `seen` and `level`, SATABS must derive and manipulate a predicate involving a pair of quantifiers, e.g. $\forall$`i`.(`i` < `db.size()`) $\Rightarrow$ ($\forall$`j`.(`j` <

`db[i].size()`) $\Rightarrow$ `db[i][j] < seen.size()`) if `c[j]` derives from a vector of clauses `db`. Such a nested quantifier predicate is beyond the state-of-the-art for predicate abstraction tools at this stage. However, we note that the property we really wish to prove is that `var` of a `Lit` is always a valid index into a set of vectors. This is a class invariant of the `Lit` class. As future work in the spirit of our approach to STL, we hope to introduce annotations for such class/type invariants, enabling proofs for programs making judicious use of ADTs.

4.5 Bibliographic Notes

Our approach to model checking code calling the Standard Template Library is based on a variation of predicate abstraction [Graf and Saïdi, 1997], and inspired by the recent success of software model checkers [Ball and Rajamani, 2002, Henzinger et al., 2002, Chaki et al., 2004, Clarke et al., 2004] based on predicate abstraction and counterexample-guided abstraction refinement [Clarke et al., 2000].

We extend our SAT-based predicate abstraction [Clarke et al., 2004] to handle a large subset of the C++ language, including objects, (operator) overloading, references and templates. Previous abstraction-based model checkers neither handle C++ programs nor provide an operational semantics supporting implementation-independent verification of code using the STL.

Wang and Musser's present a dynamic approach for verifying template code (using `gdb`, which provides correctness proofs only if loop invariants are provided by a programmer [Wang and Musser, 1997].

The CMC model checker [Musuvathi et al., 2002] can, in theory, verify C++ code compiled with templates and STL constructs, but checks implementation-dependent behavior as it performs explicit-state exploration, actually executing the code (with the attendant scaling and completeness problems). JPF 2 [Visser et al., 2003] has been applied to Java code that makes use of standard Java containers, but also relies on explicit exploration. At the other end of the spectrum, SAVCBS 2006 presented iterator specification as a challenge problem, resulting in a number of approaches, focusing mainly on logical specification rather than practical verification methods [Jacobs et al., 2006, Weide, 2006, Bierhoff, 2006, Krishnaswami, 2006]. Cok shows how to use ESC/Java 2 and JML to verify usage in some cases, but notes the serious limitations of such an approach[Cok, 2006].

Gregor and Schupp [Gregor and Schupp, 2003, 2006] describe STLlint, a static analysis tool for checking properties of the STL. Their goals are quite similar to ours: checking the implementation-independent properties of STL usage in source code. STLlint relies on symbolic execution of an executable specification, similar in spirit to our approach, but without a formalization to establish the link between the operational semantics and the STL definition, or the power of model checking to produce *counterexample traces* for errors. The latter is critical both for determining which errors are spurious and which are genuine, and for correcting real errors. Our approach additionally provides correctness proofs in cases

Program 4.3 example2.cpp

```
1 int main(int argc, char* argv)
2 {
3   int x;        // not initialized
4   std::vector<int> vec;
5   vec.reserve(x);
6
7   vec.push_back(0);
8   std::vector<int>::iterator it =
9       vec.begin();
10   vec.push_back(1);
11
12   if(x > 1) *it=1;
13 }
```

where STLlint produces a false positive (due to a lack of disjunctions in invariants, for example), including in `example.cpp` with the addition of a single loop and boolean variable.

Program 4.3 is correct because if `x` is greater than one, then vector `v` can hold at least two elements without reallocating its content. In such cases, STLlint produces an error while more powerful model checking techniques such as predicate abstraction confirm the correctness of the code.

More importantly, STLlint *misses errors* related to container lifetime easily detectable by SATABS, as in this code, where the assignment to `*it` references a destroyed vector `v`.

```
vector<int>::iterator it;
{
   vector<int> v;
   v.push_back(0);
   it = v.begin();
}
*it = 10;
```

Our approach is also able to detect STL errors related to container lifetime undetectable by STLlint. Program 4.4 declares a function `bad_func` that returns an iterator to the local container `v` — a container which is destroyed when the function returns. After the first call of `bad_func` on line 16, iterator `it_bad` is not valid but is used as parameter in the second call of `bad_func` on line 19.

4.6 Summary

We have shown how an operational semantics for the defined behavior of the C++ Standard Template Library may be used to verify programs with STL data

Program 4.4 example3.cpp

```
1  vector<int>::iterator
2    bad_func(vector<int>::iterator it)
3  {
4    vector<int> v;
5    v.push_back(0);
6    *it = 10;
7    return v.begin();
8  }
9
10 int main()
11 {
12   vector<int> v;
13   v.push_back(0);
14
15   vector<int>::iterator it_bad =
16         bad_func(v.begin());
17
18   // cause assertion failure on line 7
19   bad_func(it_bad);
20 }
```

structures in an implementation-independent manner, leveraging the high-level nature of abstract data types to aid predicate abstraction. The success of this effort demonstrates that ADTs may be as useful in assisting automated reasoning tools in "understanding" code as they are in assisting programmers in organizing code: theorem provers and abstraction engines find ADTs easier to reason about than low-level pointer manipulations, as the implicit relationships in structures are made explicit, in an implementation-independent way. This approach relies on the first reported symbolic model checker for complex C++ code, implemented in the SATABS model checker.

5

The SystemC Language

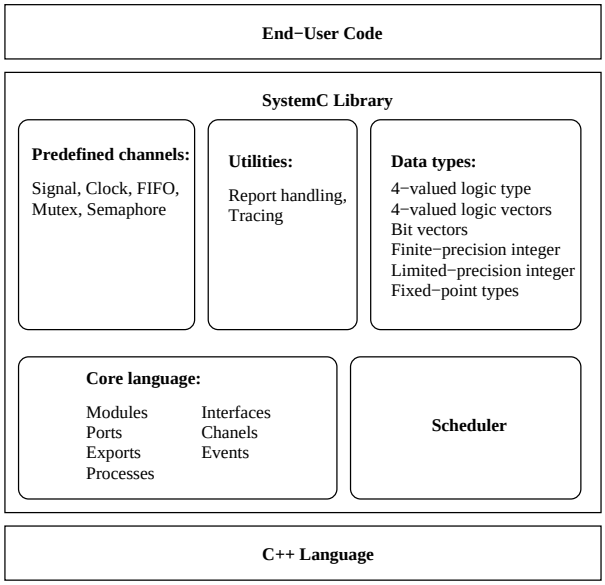
5.1 Introduction

SYSTEMC is a system-level modeling language implemented as a C++ library. It offers support for concurrency and arbitrary-width bit-vector arithmetic. Along with an event-driven simulation environment, the library provides a notion of timing, which is well-suited for modeling circuits. SystemC permits describing a system at several levels of abstraction, starting at a high-level functional description, down to synthesizable gate-level. A SystemC program consists of a set of *modules*. Modules may declare processes, ports, internal data, channels and instances of other modules. Processes implement the functionality of the module, and are sensitive to events. As in Verilog or VHDL, ports are objects through which the module communicates with other modules. Although variables are shared between processes, classic interprocess communication is achieved through predefined channels such as signals and FIFOs. The architecture of the SystemC language is shown in Figure 5.1. *SystemC* is essentially a *C++* library with concurrency support. The library offers a set of data types useful for hardware modeling and certain types of software programming. These include 2-valued and 4-valued bit-vectors of arbitrary width, and fixed-point representations. Additionally, the library also includes a set of built-in primitive channels such as signals and FIFOs.

5.2 Method Processes and Threads

Processes are plain C++ functions that are registered at runtime to the SystemC kernel using low-level routines. In general, processes are created during the elaboration of the module hierarchy before the simulation starts – technically, the language offers the possibility to create processes dynamically during simulation but this feature is rarely used. The scheduler maintains a list of *runnable* processes to execute. Those processes are run sequentially one by one in arbitrary order. Upon execution, a process is removed form the set of runnable processes. Additionally, the scheduler keeps track of a sensitivity list of events for each process. Sensitivity lists are used to notify processes waiting for events. Upon notification

Figure 5.1 The *SystemC* language architecture



of events, the scheduler inserts all the processes waiting for these events in the set of runnable processes. The SystemC language distinguishes three classes of processes: *threads*, *clocked threads*, and *method processes* [1], which we are reviewing next.

5.2.1 Threads and Clocked Threads

The execution of both threads and clocked threads is driven by events, the main distinction between those two classes being that clocked threads are sensitive only to one clock signal – the scheduler can then uses this distinction to improve simulation performances. In comparison, the SystemC standard imposes no restriction on the sensitivity list of threads. So, we employ the term 'thread' to designate both classes indiscriminately.

Typically, threads are meant for modeling non-hardware components such as test-benches. Threads can suspend their execution by calling the function *wait*. Otherwise, threads run without interruption. Upon context switches, the scheduler must preserve the program counter and the stack of the thread that is returning control. Subsequently, the scheduler continues with the evaluation of the remaining processes. To resume the execution of a thread, the scheduler restores the corresponding context, and the execution proceeds as if the most recent call to the function *wait* is returning. The execution of a thread terminates definitively if the execution reaches the end of the function.

[1] As a convention, we employ the term 'process' without the adjective 'method' to designate a (clocked) thread or a method process indiscriminately.

5.2.2 Method Processes

The implementations of threads and method processes differ significantly. Method processes are designed for fast execution of hardware logic. Like threads, method processes are sensitive to events. But unlike threads, the SystemC standard forbids method processes to call the function *wait*. Method processes runs without interruption – the scheduler gains control back only when the execution of the method process terminates. In this way, the scheduler needs to preserve no context information for method processes.

From a technical point of view, method processes can be implemented using threads. Program 5.1 illustrates such a reduction. Module *mod* declares one thread and one method processes called *thread* and *method_process*, respectively. Both processes are sensitive to the same signal *b*, and they perform equivalent computations except that the solution with a method process is faster at runtime.

Program 5.1 Example of implementation of a method process with a thread.

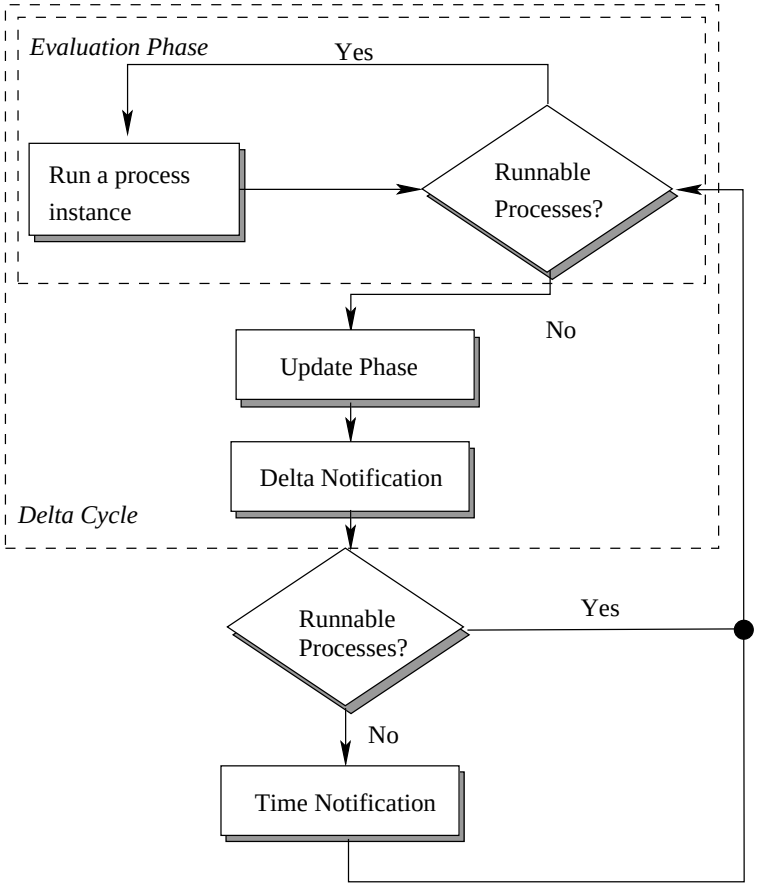
```
SC_MODULE(mod)
{
  bool a;
  sc_signal<bool>  b;

  SC_COTR(mod){
    SC_THREAD(thread);          sensitive << b;
    SC_METHOD(method_process); sensitive << b;
  }

  void thread() {
    while(true)
      { a = ~b; wait(); }
  }

  void method_process()
  { a = ~b; }
};
```

5.3 The Concurrency Model of SystemC

The scheduler of SystemC follows a *co-operative multitasking* policy, meaning that the execution of processes is serialized by explicit calls to a `wait()` method and that threads are not preempted.

The scheduler tracks simulation time and *delta cycles*. The simulation time is a positive integer value (the clock). Delta cycles are used to stabilize the state of

Figure 5.2 The *SystemC* scheduler



the system. A delta cycle consists of three phases: *evaluate*, *update*, and *notify*:

1. The evaluation phase selects a process from the set of runnable processes and triggers or resumes its execution. The process runs immediately up to the point where it returns or invokes the function *wait*. The evaluation phase is iterated until the set of runnable processes is empty. The SystemC standard allows simulators to choose any runnable process, as long as the policy is consistent between runs.

2. In order to simulate synchronous executions, processes can delay change-of-state effects by scheduling *update requests*. After the evaluation phase terminates, the kernel executes any pending update request. This is called the *update phase*. Signal assignments are typically implemented using the update mechanism. Therefore, signals keep their value for an entire evaluation phase.

3. Finally, during the *delta-notification phase*, the scheduler determines which processes are sensitive to events that have occurred, and adds all such processes to the set of runnable processes.

The scheduler executes delta cycles until the set of runnable processes is empty at the beginning of the evaluation phase. Subsequently, it updates the simulation time and notifies processes waiting for the time event as shown in Figure 5.2.

6

A Formal View of the SystemC Scheduler

6.1 Introduction

THE SystemC standard defines the semantics of the concurrency model using English text that is not machine readable. We formalize the relevant aspects of the concurrency model of SystemC. Different approaches have been proposed to formalize SystemC: Ruf et al. [2001] described the semantics of SystemC using an operational semantics, Salem [2003] using a denotational semantics, Kroening and Sharygina [2005] using Kripke structures, Savoiu et al. [2005] using petri-nets. In contrast to this related work, we express the concurrency model of SystemC using a fixed-point semantics – a choice motivated by the iterative nature of scheduling algorithm.

We refer the reader to Slonneger and Kurtz [1995] for an introduction to fixed-point semantics. We first recall standard definitions from the literature:

Theorem 6.1.1. *If D is a finite set, D with $\subseteq$ forms a complete partial order, and $f : D \to D$ is monotone (and total), then f is also continuous.*

Theorem 6.1.2 (Kleene fixed-point theorem)**.** *Let D with $\subseteq$ be a complete partial order, and let $f : D \to D$ be any continuous (and therefore monotone) function. Then the least fixed point of f is the least upper bound of the ascending chain $\bot \leq f(\bot) \leq (f \circ f)(\bot) \subseteq ... \subseteq f^n(\bot) \subseteq ...$*

We formalize the behavior of SystemC programs by means of *transition systems*.

Definition 6.1.3 (Transition system)**.** *A* transition system *is a triple (S, S_0, θ) with a set of states S, initial states $S_0 \subseteq S$, and a set of transitions $\theta \subseteq \mathscr{P}(S \times S)$. A transition $\alpha \in \theta$ is a relation on S.*

Note that the state comprises not only of the data of the processes, but also of the data required for the scheduler (process queue, event notifications). A process $\alpha \in \theta$ is a relation between states, that is, a process can exhibit both nondeterministic and non-terminating behavior. Nondeterminism is typically caused

by external inputs. The execution of the process may not terminate due to an unbounded loop, or may simply abort with an error. We assume then that the execution enters a special error state. For $\alpha \in \theta$, we write $s \xrightarrow{\alpha} t$ if $\langle s, t\rangle \in \alpha$. A transition α is *enabled* in a state s if there exists a state t such that $s \xrightarrow{\alpha} t$, and we write $\alpha \in \mathit{Enabled}(s)$ to denote this fact – $\mathit{Enabled}$ is a mapping from S to $\mathscr{P}(\theta)$. Otherwise, α is *sleeping* in s. In the context of SystemC, an enabled process is usually called *runnable*.

In the subsequent definitions, we formalize the semantics of the evaluation and delta phases in terms of functions from $\mathscr{P}(S)$ to $\mathscr{P}(S)$. Both phases are the least solution f of a fixed-point equation of the from $F(f) = f$, where $F : \big(\mathscr{P}(S) \to \mathscr{P}(S)\big) \longrightarrow \big(\mathscr{P}(S) \to \mathscr{P}(S)\big)$ is a higher-order function. We apply the usual ordering for two functions $f_i, f_j : \mathscr{P}(S) \to \mathscr{P}(S)$:

$$f_i \subseteq f_j \quad \Rightarrow \quad \forall X \in \mathscr{P}(S).\ f_i(X) \subseteq f_j(X)\ .$$

The set of all functions from $\mathscr{P}(S)$ to $\mathscr{P}(S)$ with this ordering forms a complete partial order. Let $\bot : \mathscr{P}(S) \to \mathscr{P}(S)$ denote the minimum, i.e., $\bot(X) = \emptyset$. Using the Theorems 6.1.2 and 6.1.1 we make the usual observations: A least fixed point exists if F is continuous – the application of F preserves the least upper bound. To show that F is continuous, it is sufficient to prove that F is monotone and that the set of all functions from $\mathscr{P}(S)$ to $\mathscr{P}(S)$ is finite. Provided that F is continuous, the least solution of $F(f) = f$ can be computed iteratively by beginning with $f_0 = \bot$, and applying the recurrence $f_{n+1} = F(f_n)$ until saturation. If the set of all functions from $\mathscr{P}(S)$ to $\mathscr{P}(S)$ is infinite, then continuity has to be proven in a different way.

6.2 The Evaluation Phase

We start with the formalization of the evaluation phase. Definition 6.2.1 models the evaluation phase $\epsilon : \mathscr{P}(S) \to \mathscr{P}(S)$ as the least function that satisfies the equation $E(\epsilon) = \epsilon$ where E is the higher-order function given in the definition.

Definition 6.2.1. *Let* $E : \big(\mathscr{P}(S) \to \mathscr{P}(S)\big) \longrightarrow \big(\mathscr{P}(S) \to \mathscr{P}(S)\big)$ *denote the following function:*

$$E(f) \triangleq \lambda X.\Big(\big\{s \in S \big| \mathit{Enabled}(s) = \emptyset\big\} \cup f\Big(\bigcup_{s \in X} \bigcup_{\rho \in \mathit{Enabled}(s)} \rho(s)\Big)\Big)\ .$$

The evaluation phase $\epsilon : \mathscr{P}(S) \to \mathscr{P}(S)$ *is the least solution of the fixed-point equation* $E(\epsilon) = \epsilon$.

Note that this definition of ϵ models the choice of ordering of processes the scheduler can make.

Given two functions $\epsilon_{i+1}, \epsilon_i : \mathscr{P}(S) \to \mathscr{P}(S)$ such that $\epsilon_{i+1} = E(\epsilon_i)$, the function ϵ_{i+1} contains more information than ϵ_i in the sense that for any set of states

X in $\mathscr{P}(S)$, $\epsilon_i(X)$ is an under-approximation of $\epsilon_{i+1}(X)$. Informally, ϵ_i describes all computations up to bound i, whereas ϵ_{i+1} describes all computations up to bound $i+1$. Definition 6.2.1 gives rise to Lemma 6.2.2, which establishes the monotonicity of E.

Lemma 6.2.2. *Function E (Def. 6.2.1) is monotone: $E(f)$ describes a function smaller than $E(g)$ if f describes a function smaller than g.*

Proof. Let us we write $\mathcal{A}$ and $\mathcal{B}$ for "$\{_{s\in X|Enabled(s)=\emptyset}\}$" and "$\bigcup_{s\in X}\ \bigcup_{\rho\in Enabled(s)}\rho(s)$", respectively. If $f \subseteq g$, then $E(f)(X) = \big(\mathcal{A} \cup f(\mathcal{B})\big) \subseteq \big(\mathcal{A} \cup g(\mathcal{B})\big) = E(g)(X)$. Thus, we conclude that $E(f)$ is smaller than $E(g)$. □

From Lemma 6.2.2 and Theorems 6.1.1, 6.1.2 we conclude that the evaluation phase is well-defined if S is finite:

Theorem 6.2.3. *The evaluation phase ϵ is well-defined for models with bounded memory.*

Proof. We write $\mathscr{P}(S) \to \mathscr{P}(S)$ to denote the set of all (total) functions from $\mathscr{P}(S)$ to $\mathscr{P}(S)$. As mention before, the set $\mathscr{P}(S) \to \mathscr{P}(S)$ together with the standard $\subseteq$ operator forms a complete partial order. Since E is monotone (Lemma 6.2.2), it is sufficient to show that $\mathscr{P}(S) \to \mathscr{P}(S)$ is finite to prove that E is continuous (Theorem 6.1.1). Note that if S is finite then $\mathscr{P}(S) \to \mathscr{P}(S)$ is also finite – there exists only a finite number of functions over $\mathscr{P}(S)$. Hence, E is continuous if S is bounded. Finally, if E is continuous then Theorem 6.1.2 guarantees the existence of a least fixed point and provides an iterative method to compute it. □

Additionally, we introduce Lemma 6.2.4, which establishes that ϵ is additive, e.g, $\epsilon(X) = \bigcup_{s\in X} \epsilon(\{s\})$. The proof is by induction over the ascending chain of the successive approximations of ϵ given by $\epsilon_{i+1} = E(\epsilon_i)$ and $\epsilon_0 = \bot$.

Lemma 6.2.4. *The evaluation phase ϵ is additive: $\epsilon(X \cup Y) = \epsilon(X) \cup \epsilon(Y)$.*

Proof. We demonstrate by induction that ϵ_n is additive for all functions in the chain defined defined by $\epsilon_{n+1} = E(\epsilon_n)$ and $\epsilon_0 = \bot$. The property holds for $n = 0$ as $\epsilon_0(A \cup B) = \bot(A \cup B) = \bot(A) \cup \bot(B) = \epsilon_0(A) \cup \epsilon_0(B)$. The case for $n+1$ follows directly from the definition of E (Def. 6.2.1) and the induction hypothesis $\epsilon_n(A \cup B) = \epsilon_n(A) \cup \epsilon_n(B)$, so we conclude that the property holds for any ϵ_n. In particular, this remains true when the computation of ϵ_{n+1} reaches the fixed point. □

6.3 The Delta Phase

We continue with the formalization of the delta cycle in this style. Definition 6.3.1 expresses the delta cycle as the least function $\delta : \mathscr{P}(S) \to \mathscr{P}(S)$ that satisfies the equation $D(\delta) = \delta$ where D is the higher-order function given in the definition.

Definition 6.3.1. *Let* $Up : \mathscr{P}(S) \to \mathscr{P}(S)$ *denote the function that updates the state and notifies the processes as described by the standard. The function* $D : \big(\mathscr{P}(S) \to \mathscr{P}(S)\big) \longrightarrow \big(\mathscr{P}(S) \to \mathscr{P}(S)\big)$ *is the following higher-order function:*

$$D(f) \triangleq \lambda X.\big(\{_{s\in X}|_{Enabled(s)=\emptyset}\} \cup (f \circ Up \circ \epsilon)(X)\big) \ .$$

The delta cycle $\delta : \mathscr{P}(S) \to \mathscr{P}(S)$ *is the least solution of the fixed-point equation* $D(\delta) = \delta$.

Definition 6.3.1 gives rise to Lemma 6.3.2, which guarantees the monotonicity of D. Subsequently, Theorem 6.3.3 establishes that the delta phase is well defined if S is finite.

Lemma 6.3.2. *Function* D *(Def 6.3.1) is monotone.*

Proof. Let us we write $\mathcal{A}$ and $\mathcal{B}$ for "$\{_{s\in X}|_{Enabled(s)=\emptyset}\}$" and "$(Up \circ \epsilon)(X)$", respectively. If $\delta_i \subseteq \delta_j$, then $E(\delta_i)(X) = \big(\mathcal{A} \cup \delta_i(\mathcal{B})\big) \subseteq \big(\mathcal{A} \cup \delta_j(\mathcal{B})\big) = D(\delta_j)(X)$. Thus, we conclude that $D(\delta_i)$ is smaller than $D(\delta_j)$. □

Theorem 6.3.3. *The least fixed point* δ *is well-defined for models with finite memory.*

Proof. Same argument as for Theorem 6.2.3. □

6.4 The Simulation Time

We conclude the formalization of the concurrency model of SystemC with Definition 6.4.1, which captures the simulation semantics of SystemC.

Definition 6.4.1. *Let* $\mathbb{N}$ *denote the set of positive integers, let* $S_0 \subseteq S$ *denote the set of initial states, and let* $Up_{time} : \mathscr{P}(S) \to \mathscr{P}(S)$ *denote the function that updates the simulation time and notifies the processes waiting for this event. The execution semantics of SystemC is given by the function* $Sim : \mathbb{N} \to \mathscr{P}(S)$*, which defines the states of the system as a function of the simulation time:*

$$Sim(0) = S_0, \qquad Sim(t+1) = (\delta \circ Up_{time} \circ Sim)(t) \ .$$

6.5 Correctness of Partial Order Reduction for SystemC

Partial order reduction restricts the analysis of the behaviors of a concurrent system to a set of representative traces. For a specific class of properties, the reduction is sound: if the property of interest holds in the reduced model, it also holds in the original one. In the context of this paper, we assume that the property can

be defined as a state predicate, which is evaluated at the end of the evaluation phase.[1] We formalize a sufficient condition for the soundness of partial order reduction as follows.

Definition 6.5.1. *Let $\hat{\epsilon} : \mathscr{P}(S) \to \mathscr{P}(S)$ denote a function that models the evaluation phase when applying a reduction technique, e.g., by restricting the ordering of evaluation. We say that $\hat{\epsilon}$ is* sound *for reachability if $\epsilon \subseteq \hat{\epsilon}$.*

Definition 6.5.1 and Lemma 6.2.4 yield the following theorem that provides a method to show correctness of a partial order reduction $\hat{\epsilon}$:

Theorem 6.5.2. *Let $\hat{\epsilon} : \mathscr{P}(S) \to \mathscr{P}(S)$ denote a function that models the evaluation phase. $\hat{\epsilon}$ is* sound *for reachability if:*

1. $\hat{\epsilon}(A \cup B) = \hat{\epsilon}(A) \cup \hat{\epsilon}(B)$,
2. *and for all s in S, $\epsilon(\{s\}) \subseteq \hat{\epsilon}(\{s\})$.*

Typically, $\hat{\epsilon}$ is defined as the least solution of a fixed-point equation, and therefore, additivity is usually demonstrated by induction over the ascending chain of $\hat{\epsilon}_i$.

Note that Def. 6.5.1 permits over-approximation. We restrict the discussion to precise reductions $\epsilon = \hat{\epsilon}$ in order to prevent spurious counterexamples. In the following, we illustrate our correctness criterion by means of definitions of the persistent-set and sleep-set techniques in terms of a fixed-point semantics. Both approaches preserve deadlock states, i.e., states without enabled (runnable) transitions Godefroid [1996].

Definition 6.5.3. *Let $Persistent : S \to \mathscr{P}(\theta)$ denote some function that returns a set of persistent processes (Def. 2.2.3). Additionally, we require that $Persistent(s)$ is empty only if no process is runnable in s. We define $\hat{E}_P : \big(\mathscr{P}(S) \to \mathscr{P}(S)\big) \longrightarrow \big(\mathscr{P}(S) \to \mathscr{P}(S)\big)$ as the following higher-order function:*

$$\hat{E}_P(f) \triangleq \lambda X.\Big(\{s \in X \,|\, Enabled(s) = \emptyset\} \cup f\big(\bigcup_{s \in X} \bigcup_{\rho \in Persistent(s)} \rho(s)\big)\Big) .$$

The persistent-set technique is the least solution $\hat{\epsilon}_P : \mathscr{P}(S) \to \mathscr{P}(S)$ of the fixed-point equation: $\hat{E}_P(\hat{\epsilon}_P) = \hat{\epsilon}_P$.

During exploration, techniques based on sleep sets maintain a set of enabled transitions that can be skipped. Thus, we extend the states in S to carry sleep sets, and we write $Sleep(s)$ to denote the set of enabled processes in s that can be skipped; that is, $Sleep : S \to \mathscr{P}(\theta)$. Additionally, let $NextSleep : (S \times \theta) \to S$ denote a function that takes as argument a state s and a process $\alpha \in \theta$ and returns a state equal to s except that $(Sleep \circ NextSleep)(s, \alpha)$ describes the next sleep set, i.e., the sleep set for the states in $\alpha(s)$. Definitions 6.5.4 and 6.5.5 formalize this technique.

[1] Assertions within an atomic block are verified by considering their post-image.

Definition 6.5.4. *$Sleep \circ NextSleep(s, \alpha)$ is a sleep set if and only if $\alpha \in Enabled(s)$ and $\beta \in (Sleep \circ NextSleep)(s, \alpha)$ implies that the following holds:*

1. *$\beta \in Enabled(s)$,*
2. *α and β are independent in s, and*
3. *$\alpha \in (Sleep \circ NextSleep)(s, \beta) \Rightarrow \beta \in Sleep(s)$.*

Definition 6.5.5. *Let $NextSleep(s, \rho)$ denote a function that computes sleep sets (Def. 6.5.4). We define $\hat{E}_S : \big(\mathscr{P}(S) \to \mathscr{P}(S)\big) \longrightarrow \big(\mathscr{P}(S) \to \mathscr{P}(S)\big)$ as the following function:*

$$\hat{E}_S(f) \triangleq \lambda X.\Big(\{_{s \in X | Enabled(s) = \emptyset}\} \cup f\big(\bigcup_{s \in X} \bigcup_{\rho \in Enabled(s) \setminus Sleep(s)} (\rho \circ NextSleep)(s, \rho)\big)\Big) .$$

The sleep-set technique is the least solution $\hat{\epsilon}_S : \mathscr{P}(S) \to \mathscr{P}(S)$ of the fixed-point equation: $\hat{E}_S(\hat{\epsilon}_S) = \hat{\epsilon}_S$.

7

Static Analysis for SystemC with Scoot

7.1 Introduction

DUE to the complexity of C++, existing static analyzers for SystemC consider only small fragments of the language, essentially searching for specific key-words [Berner and Talpin, 2005, Kostaras and Vergos, 2005]. In this chapter, we present SCOOT, a model extractor for SystemC. The tool supports a wide range of language constructs, as it based on our C++ front-end. The models generated by SCOOT can serve several purposes, ranging from verification and simulation to synthesis. The tool is tightly integrated with verification back-ends for Bounded Model Checking (CBMC) [Kroening et al., 2004] and SAT-based predicate abstraction (SATABS) [Clarke et al., 2005]. Results on applying model checking to models generated by SCOOT have been reported before [Kroening and Sharygina, 2005]. As an example of the utility of SCOOT beyond formal verification, we report results indicating that our tool can be used to improve the performance of dynamic execution up to five times.

7.2 Overview of Scoot

SCOOT uses the C++ front-end of Chapter 3 to translate the SystemC source files into a control flow graph. The nodes of the graph are annotated with assignments and guards (implemented in the typechecking and CFG-conversion phases in Figure 7.1). Subsequently, static analysis techniques are used to determine the following information, which is specific to SystemC:

- The module hierarchy,
- the sensitivity list of the processes, and
- the port bindings.

The SystemC library makes heavy use of virtual functions and dynamic data structures, which are not easily analyzed by static analysis techniques. SCOOT

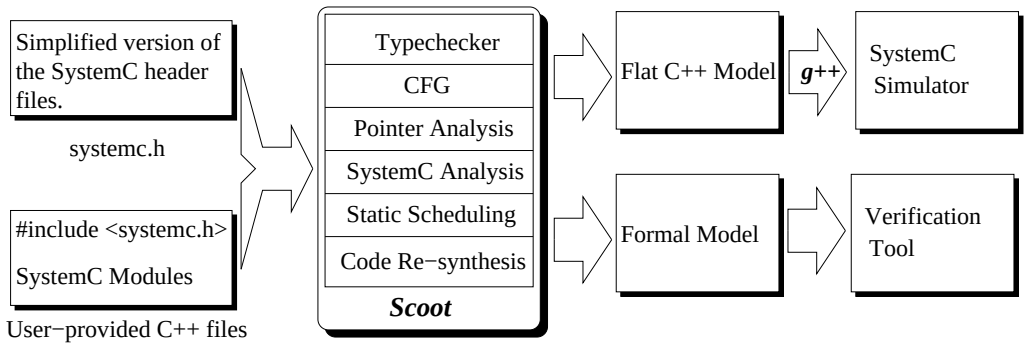


Figure 7.1: Overview of SCOOT

abstracts implementation details of the library by using simplified header files that declare only relevant aspects of the API and omit the actual implementation. Systems described using SystemC shall provide a *sc_main* procedure for building the module hierarchy. The systemc-analysis phase sequentially processes the body of this function, and reports any model-construction error.

7.3 Static Analysis of SystemC

Table 7.1 Principal SystemC types for RTL modeling.

`sc_event`	Objects of class `sc_event` are used for driving simulation. This class offers notification methods to trigger the execution of processes.
`sc_module`	Basic building block of the system.
`sc_signal<T>`	Communication channel intended to model the behavior of a digital signal.
`sc_clock`	Communication channel intended to model the behavior of a clock signal in hardware.
`sc_fifo<T>`	Communication channel intended to model the behavior of a FIFO.
`sc_mutex`	Communication channel intended to model the behavior of a mutex.
`sc_in<T>`	Input port. Must be bound to a signal before simulation starts.
`sc_out<T>`	Output port. Must be bound to a signal before simulation starts.
`sc_inout<T>`	Inout port. Must be bound to a signal before simulation starts.

In this section, we present the static analysis technique that we use to extract model hierarchy. This information is obtained via precise alias analysis. SystemC models can be compiled and linked to the SystemC library using `g++` providing thus a cost-efficient way of simulating designs. In general, better execution performances can be achieved using commercial simulators. Simulators such as Synopsys's VCS and Cadence's NCVerilog employ special compilers for VERILOG that can perform high-level optimizations. SCOOT is a research compiler for SystemC with similar aims. To facilitate the analysis of SystemC models, SCOOT considers a subset of the language interface that is relevant to the user. We define this subset by means of a special version of the header files of SystemC. These header files declare SystemC types, such as the classes for signals, modules, and ports. Table 7.1 lists the principal SystemC components currently supported. Our header files cover the standard requirements for RTL development and can be easily extended to include additional communication channels. Finally, SCOOT has built-in support for bit-vector arithmetic and handles the bit-vector types `sc_uint` and `sc_int` natively.

7.3.1 The Supported Subset

The purpose of SystemC is to provide system designer with a standard C++ library for modelling complex hardware and software systems. The SystemC standard provides complete description of that library so that designer can safely refer to this standard, and companies can develop tools for the analysis of SystemC models. As for VERILOG and SYSTEM VERILOG, vendors are expected to provide support for SystemC for only a relevant subset of the language:

> "It is anticipated that tool vendors will create implementations that support only a subset of this standard or that impose further constraints on the use of this standard. Such implementations are not fully compliant with this standard but may nevertheless claim partial compliance with this standard and may use the name SystemC."
>
> IEEE Standard SystemC Language Reference Manual

From a simulation perspective, specific implementation may constrain the usage of SystemC to improve simulation performances dramatically, and from verification perspective, a static analyser can abstract C++ details of the implementation to simplify reasoning. SCOOT's support for SystemC targets a broad subset of the language meaningful for RTL modeling and TLM channels such as fifos.

7.3.2 Implementation of Modules

A SystemC model is composed hierarchically in term of modules. As in VERILOG, a module encapsulates the description of an execution unit. In SystemC, a

module is a C++ class that derives from class `sc_module`. Prog. 7.1 is SCOOT's declaration of the class `sc_module`.

Program 7.1 SCOOT's implementation of class `sc_module`. The creation of a new module is marked by a call to the special function `scoot_module_decl`.

```
class sc_module
{
    public:

   // Constructors
   sc_module()
   { scoot_module_decl(this,null); }

   sc_module(char* name)
   { scoot_module_decl(this,name); }

   private:
   sc_module(const sc_module&);                // Disabled
   sc_module& operator=(const sc_module&);     // Disabled
};
```

SCOOT exploits the inheritance and initialization mechanisms of C++ to construct the module hierarchy. Our implementation of class `sc_module` is primary used for typechecking SystemC models and performs very few computation. The public interface of the class declares two constructors: the first one is the default constructor and takes no parameter, whereas the second one takes as parameter an identifier for the module. When an object deriving from type `sc_module` is created, the compiler calls one of the constructors of that class. Upon invocation, the constructors of `sc_module` call `scoot_module_decl` to mark the creation of a new module. This function has a special meaning for SCOOT and is not implemented in C++. The function takes two parameters: a pointer to the module that is being declared, and a pointer the name of the module. SCOOT uses a flow-sensitive alias analysis to determine the value of those pointers statically. The creation of other SystemC components such as processes, signals, and ports follows a similar pattern.

The class `FullAdder` shown in Prog. 7.2 describes a full-adder module, and inherits thus from `sc_module`. The interface of the module comprises the three 1-bit-wide input ports `A`, `B`, and `Cin` and the two 1-bit-wide output ports `S` and `Cout`.

Upon instantiation of an object of type `FullAdder`, the constructor shown in Prog 7.2 is called to initialize the state of the object. The C++ language guarantees the following sequence of initialization:

1. The default constructor of class `sc_module` is called first,

Program 7.2 1-Bit Full Adder

```
class FullAdder: sc_module {
  sc_in<bool> A,B,Cin;
  sc_out<bool> S,Cout;
  void add();
  HAS_PROCESS(FullAdder);
  FullAdder()
  {
    SC_METHOD(add);
     sensitive << A, B, Cin;
  }
};
```

2. the constructors of `A`, `B`, `Cin`, `S`, and `Cout` are called next, and finally,

3. the body of the constructor is executed.

Beside data members, a module declares processes for running computations. Those processes are defined in the body of the constructor. SystemC provides the C++ macros `SC_METHOD(method)` and `SC_METHOD(method)` to declare method processes and threads. In the previous example, method `add` is a method process that is made sensitive to the inputs ports `A`, `B`, and `Cin`.

7.3.3 Implementation of Signals

Signals are essential components for modelling circuits. The standard SystemC library implements signals using complex mechanisms based on multiple-inheritance and virtual functions, which makes the static analysis of SystemC models difficult. We implement the concept of a signal in a way more adequate for formal reasoning.

Prog 7.3 is SCOOT's implementation of signals. In 7.3, the constructor of the class `sc_signal<T>` calls the function `scoot_channel_decl` to mark the creation of a new signal. This function takes two parameters: a pointer to the module that is being declared and a pointer the name of the module. The method `read` is the implementation of the read operation and retrieves the value of the field `current_value`. The method `write` is the implementation of the write operation and sets the value of the field `new_value`. During the update phase, the scheduler executes the method `update` to replace the value of `current_value` with the value of `new_value` and to notify the processes.

7.3.4 Implementation of Ports

Ports are used to model interconnection between modules. In SystemC, ports derive from the template type `sc_port<IF>` where `IF` stands for the interface of

Program 7.3 SCOOT's implementation of signals.

```
template <class T>
  class sc_signal
{
  public:
    sc_signal():
      current_value((T)0), next_value((T)0)
      { scoot_channel_decl(this);}

    sc_signal(const char* name):
      current_value((T)0), next_value((T)0)
      { scoot_channel_decl(this,name);}

    T read() const
    { return current_value; }

    void write(T t)
    { next_value =  t; }

  private:
    void update() {
      if(current_value != next_value)
        value_changed_event.notify();
      current_value = next_value;
    }

    T current_value, next_value;
    sc_event _value_changed_event;
};
```

the underlying communication channel. Conceptually, a port is pointer to some external object. In addition to `sc_port<IF>`, the SystemC language provides the three types of specialized ports `sc_inout<T>`, `sc_in<T>`, and `sc_out` for signals. Program 7.4 shows a simplified version of SCOOT's implementation of ports. The class closely matches the standard API. SCOOT use the functions *scoot_port_decl*, *scoot_port2port*, and *scoot_port2channel* as marker functions to detect the instantiation of a new port and to set connections. Those functions have no implementation in C++.

7.3.5 Discovering Module Hierarchy

In contrast to VERILOG, the module hierarchy of SystemC model is discovered at runtime before the simulation starts during the *elaboration phase*. SystemC model shall declare a function *sc_main* for building the module hierarchy and for

Program 7.4 SCOOT's implementation of ports.

```
template <class IF>
class sc_port
{
public:
// Constructors
sc_port(){};
explicit sc_port(const char* name):
{ scoot_port_decl(this,name); }

// Port-to-channel binding
void operator()(IF& _if)
{
  const scoot_channel* channel = & _if.get_channel();
  scoot_port2channel(this,channel);
}

// Port-to-port binding
void operator()(sc_port<IF>& port)
{ scoot_port2port(this, &port); }

// Return the channel bound to the port
IF* operator->()
{ return  channel; }

const IF* operator->() const
{ return channel; }

private:
IF* channel; // The channel bound to the port
};
```

starting the simulation. After parsing, typechecking, and converting SystemC code to control-flow graph, SCOOT extracts the module hierarchy statically. Internally, SCOOT proceeds in the following way:

1. First, SCOOT recursively inlines function *sc_main* including calls to constructors.

2. Subsequently SCOOT unrolls loops, propagate constants, and

3. compute point-to-sets for each program location.

4. Finally, SCOOT analyses calls to the functions shown in Table 7.2 using the points-to information from previous step to build the module hierarchy.

Note that the module hierarchy may depend on information unavailable at compile-time such as command-line arguments. Additionally, the computation of the module hierarchy is an undecidable problem.

We focus on SystemC models with module hierarchies that can be extracted statically.

Upon call to function *sc_start*, the SystemC kernel verify that ports are bound correctly.

7.4 Static Scheduling

Technically, SystemC modules are plain C++ classes that can be compiled and linked to a runtime scheduler, providing thus a way to simulate the behavior of the system. The model hierarchy is discovered at run-time only and therefore, the compiler is missing opportunities to take advantage of this knowledge. To illustrate the utility of the model generated by SCOOT, we re-synthesize more efficient C++ code from the model.

SystemC has a *co-operative multitasking* semantics, meaning that the execution of processes is serialized by explicit calls to a `wait()` method and that threads are not preempted. The scheduler tracks simulation time and *delta cycles*. The simulation time corresponds to a positive integer value (the clock), while delta cycles are used to stabilize the state of the system. A delta cycle consists of three phases: *evaluate*, *update*, and *notify*, which were described in Section 5.3.

The standard SystemC scheduler contains several sources of inefficiency: first, the scheduler stores data in containers that allocate memory at run-time, and second, it triggers processes using function pointers. SCOOT generates a completely static scheduler by fixing the evaluation order of the processes and resolving dynamic calls. Finally, the execution of threads is sequentialized to simplify context switches. We describe this technique in Section 7.4.1.

7.4.1 Conversion of Threads

SystemC distinguishes between *method processes* and *threads*. Technically, a method process is a C++ method that is executed up to completion and is forbidden to call synchronization routines. In contrast, a thread can suspend its execution using wait statements. The scheduler must then preserve the local state and the current program location of the running thread. Saving and restoring contexts cause overhead at runtime. To avoid this issue, SCOOT convert threads to suppress stacks and implements context switches with goto statements.

For each thread, our conversion technique proceeds as follow:

1. *Static Memory Allocation*: SCOOT substitutes the local variables of the thread by fresh ones with static storage duration. During this process, SCOOT recursively expands functions containing wait statements. These statements are converted in the next phase.

2. *Conversion of Wait/Return Statements*: SCOOT implements context switches with goto statements. It first creates a program counter to hold the location that caused the context switch. Subsequently, each wait/return statement is converted into an assignment followed by a goto statement. The assignment saves the current program location, while the goto statement returns control back to the scheduler.

3. *Branching Code*: Finally, SCOOT inserts branching instructions at the beginning of the thread to control where the execution shall resume.

Program 7.5.1 and Program 7.5.2 show the code of a thread before and after conversion, respectively. Program 7.5.1 declares a unique local variable i. After conversion, the variable is declared with static storage duration. Additionally, Program 7.5.2 introduces a program counter pc and sets its initial value to zero to indicate that the process is triggered for the first time. On lines 4 to 6, the program counter is used to decide where the execution shall resume. The wait statements in Program 7.5.1 corresponds to the assignments followed by a goto statement in Program 7.5.2 (lines 11 and 14).

Technically, this conversion approach is applicable only if wait statements are not executed within a recursive function. In practice however, SystemC threads rarely execute recursive calls that contain synchronization routines, and this technique allows SCOOT to handle threads as easily as method processes.

7.4.2 Code Re-synthesis

The intermediate representation used by SCOOT was originally designed for model checking, and uses bit-vector arithmetic expressions. After static scheduling, SCOOT translates the intermediate representation back to a flat C++ program that does not rely on the SystemC library anymore. The generated model is subsequently passed to `g++`, which results in a faster simulator.

We compare in Figure 7.1 the compilation time using SCOOT and the standard compilation flow on a 3 GHz Intel pentium 4 processor with `g++` 4.2.4. Our collection of benchmarks contains designs of industrial significance such as an updated version of the RISC-CPU that is shipped with the SystemC library and classic modules such as AES, DES, FFT, and FIR modules. The results indicate that static extraction of the module hierarchy has no negative impact on the compilation time. We compare in Figure 7.2 the runtime performance using the simulator generated by SCOOT and the original approach. The performances demonstrate that our technique can speedup the simulation up to five times on applications relevant to industry.

7.5 Bibliographic Notes

Due to the complexity of the C++ language, the development of any tool for SystemC is a difficult task. Hardware synthesis tools for SystemC only consider a small subset of the C++ syntax [Berner and Talpin, 2005, Kostaras and Vergos, 2005]. SCOOT is the first public static analyser for SystemC that is based on a C++ frontend.

VERILATOR [Synder] is a tool for converting Verilog specifications to SystemC or C++ programs. The goal is to achieve higher simulation performance.

Savoiu et al. [2005] propose to use Petri-net reductions for SystemC, and report a speedup of 1.5 for an AES core. Pérez et al. [2004] describe a static-scheduling technique restricted to method processes. Our sequentialization technique extends the benefits of static scheduling to general threads by eliminating the overhead caused by context switches.

We provide a tool that extracts formal models from SystemC code. The tool supports a broad subset of the language, as it is built on top of our C++-frontend. The main applications are formal analysis, e.g., by model checking, and synthesis. Exemplarily, we show that formal models have value even in dynamic verification: we show a significant improvement in simulation performance by using a statically scheduled model.

We are continuing to improve the SystemC support of our tool. It currently handles the most commonly used features of the SystemC API. We are also investigating additional formal techniques to further enhance static scheduling. We describe in the rest of this section existing public frontends for SystemC.

7.5.1 The SystemCXML Frontend

SYSTEMCXML [Berner and Talpin, 2005] is a parser for SystemC that can extract information about SystemC modules. The analysis is based on Doxygen, which is a tool for code documentation, and is therefore limited to the detection of simple syntactic constructs – semantic aspects are ignored.

7.5.2 The ParSyC Frontend

PARSYC [Kostaras and Vergos, 2005] is a frontend for SystemC from the University of Bremen. PARSYC only supports a small subset of SystemC, e.g, synthesizeable constructs, whereas SCOOT is based on C++ frontend that accepts much more constructs. In particular, SCOOT can be used for the analysis of SystemC/TLM models, which requires support multiple inheritance and virtual functions.

7.5.3 The Quiny Frontend

QUINY [Schubert and Nebel, 2006] is a library replacement for SystemC from the University of Oldenburg that can be used to translate SystemC code to VHDL.

This special library is used to output at runtime a VHDL file of the module hierarchy and the process statements using an approach inspired form code-reflection techniques. Therefore, QUINY requires user intervention to rewrite the program in order to overcome the lack of code-reflection support in C++.

7.5.4 The Pinapa Frontend

Moy et al. [2005] describe PINAPA, which is a tool for the extraction of SystemC models. PINAPA operates on the original SystemC library and analyses SystemC models in a two-step style: first, PINAPA compiles and runs the model to discover the module hierarchy using `g++`. Subsequently, PINAPA builds an abstract syntax tree representation of the model using the same compiler and uses information from the dynamic execution to build an internal representation of the model. In contrast, SCOOT is a C++ compiler that has a built-in knowledge of the API of SystemC. Our tool relies on data-flow analysis to detect the module hierarchy statically.

Table 7.2 SCOOT's special marker functions to create the module hierarchy.

`void scoot_module_decl(pmodule,pname);`
Function to declare a module. The first argument is a pointer to the module to declare. The second argument is the name of the module.

`void scoot_channel_decl(pchannel,pname);`
Function to declare a communication channel. The first argument is a pointer to the channel to declare. The second argument is the name of the channel.

`void scoot_port_decl(pport,pname);`
Function to declare a port. The first argument is a pointer to the port to declare. The second argument is the name of the port

`void scoot_port2port(pport1,pport2);`
Function to declare that the port that is referred by the first argument and that is referred by the second argument are bound to a same channel. Both the ports must have been declared before calling this function.

`void scoot_port2channel(pport,pchannel);`
Function to declare that the port pointed by the first argument is bound the channel pointed by the second argument. Both the port and the channel must have been declared before calling this function.

`void scoot_thread(pmodule,pcthread);`
Function to declare a thread. The first argument represents the module that is associated with the thread. The second argument is a function pointer to the thread.

`void scoot_cthread(pmodule,pcthread);`
Function to declare a clocked thread. The first argument represents the module that is associated with the thread. The second argument is a function pointer to the clocked thread.

`void scoot_method(pmodule,pmethod);`
Function to declare a method process. The first argument represents the module that is associated with the process. The second argument is a function pointer to the process.

Program 7.5 Example of conversion of a SystemC thread. Program 7.5.1 shows the original code of the thread. Program 7.5.2 shows the code after conversion.

```
1  void run()
2  {
3      int i = 0;
4      while(true){
5          if(i < 10){
6              wait();
7              i = i+1;
8          } else {
9              wait();
10             i = 0;
11         }
12     }
13     return;
14 }
```

(1) Original thread

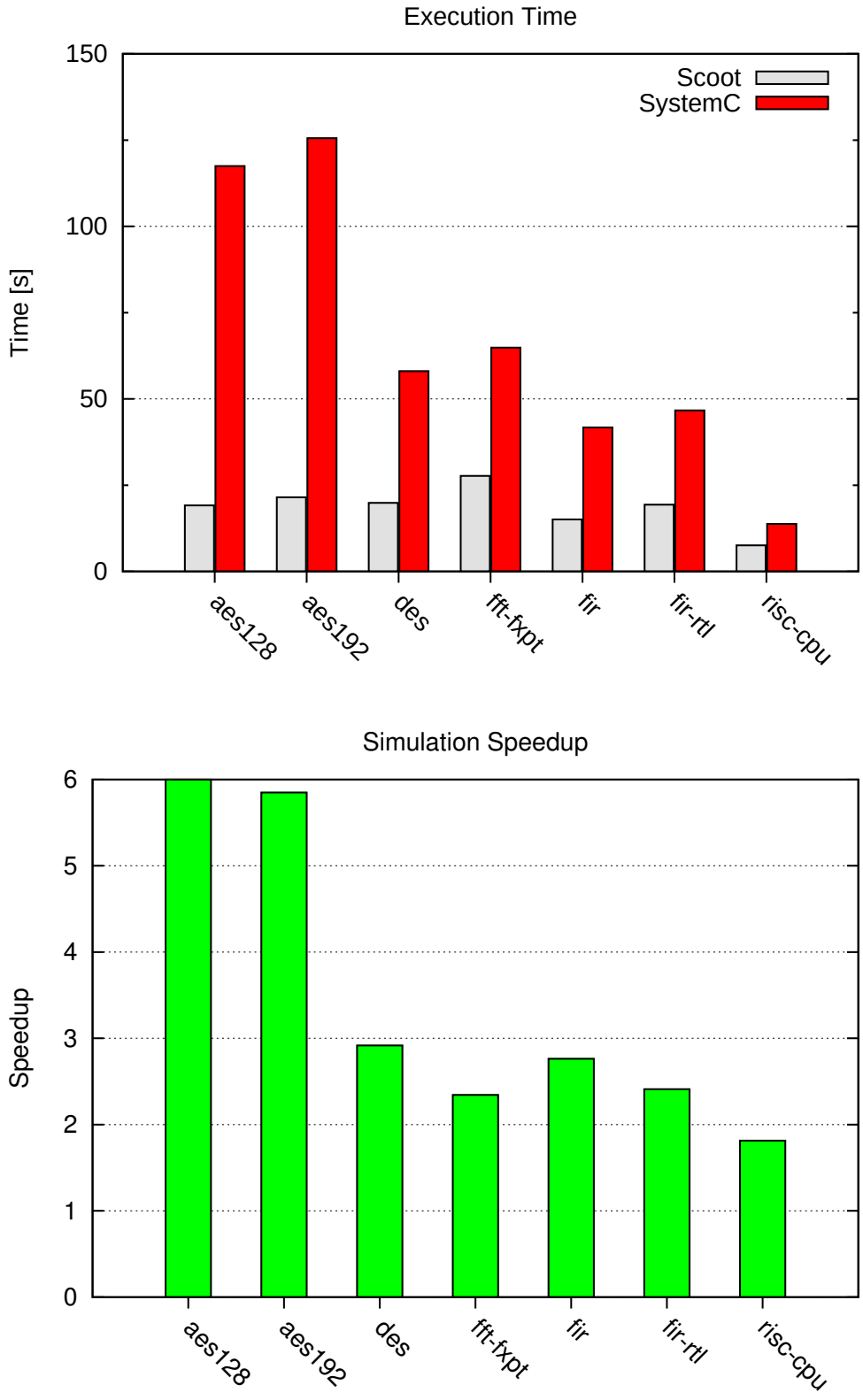
```
1  int i; int pc = 0;
2  void run()
3  {
4          if(pc == 1) goto PC1;
5          if(pc == 2) goto PC2;
6          if(pc == 3) goto EXIT;
7
8          i = 0;
9          while(true){
10             if(i < 10){
11                 pc = 1; goto EXIT;
12 PC1:             i = i+1;
13             } else {
14                 pc = 2; goto EXIT;
15 PC2:             i = 0;
16             }
17         }
18         pc = 3;
19 EXIT: ;
20 }
```

(2) Thread after conversion

Figure 7.1 Compilation statistics using SCOOT and the standard approach.

Model	#Lines	Comp. Time with SCOOT			Std. Approach
		Scoot [s]	g++ [s]	Total [s]	g++ [s]
AES-128	1483	46.36	46.94	93.3	100.41
AES-192	1528	53.18	47.22	100.4	114.2
DES	3001	45.78	37.74	83.52	145.72
FFT-FXPT	566	5.32	9.97	15.29	27.19
FIR	349	4.17	5.39	9.56	23.98
FIR-RTL	382	5.37	6.13	11.5	41.83
RISC-CPU	2311	85.89	49.13	135.02	92.49

Figure 7.2 Simulation performance



8

Race-Analysis for SystemC

8.1 Introduction

Methods of SystemC modules may be designated as *threads* or *processes*. Interleaving between those threads is performed at pre-determined program locations, e.g., at the end of a thread or when the `wait()` method is called. When multiple threads are ready for execution, the ordering of the threads is nondeterministic. Nevertheless, the SystemC standard allows simulators to adopt a deterministic scheduling policy. Consequently, simulators can avoid problematic schedules, which often prevents the discovery of concurrency-related design flaws. SystemC offers a wide range of language features such as hierarchical design by means of a hierarchy of modules, arbitrary-width bit-vector types, and concurrency with related synchronization mechanisms. SystemC permits different levels of abstraction, from a very high-level specification with big-step transactions down to the gate level. The execution model of SystemC is driven by *events*, which start or resume processes. In addition to communication via shared variables, processes can exchange information through predefined communication channels such as signals and FIFOs.

When describing synchronous circuits at the register transfer level, system designers can prevent races by restricting inter-process communication to deterministic communication channels such as *sc_signals*. However, the elimination of races from the high-level model is often not desirable: In practice, system designers often use constructs that yield races in order to model nondeterministic choices implicit in the design. In particular, models containing standard transaction-level modeling (TLM) interfaces are frequently subject to race phenomena. TLM designs usually consist of agents sharing communication resources and competing for access to them. An example is a FIFO with two clock domains – the races model the different orderings of the clock events that can arise.

Contribution Due to the combinatorial explosion of process interleavings, testing methods for concurrent software alone are unlikely to detect bugs that depend on subtle interleavings. Therefore, we propose to employ formal methods to statically pre-compute thread-dependency relations and predicates that predict

Program 8.1 Example of race condition

```
SC_MODULE(m){
  sc_clock clk; int pressure;

  void guard() {
    if(pressure == PMAX) pressure = PMAX-1;
  }

  void increment(){ pressure++; }

  SC_CTOR(m) {
    SC_METHOD(guard); sensitive << clk;
    SC_METHOD(increment); sensitive << clk;
  }
};
```

race conditions, and to use this information subsequently during the simulation run to prune the exploration of concurrent behaviors. There are several possible ways of exploiting the information:

1. From a verification perspective, those predicate provide valuable insight into the behavior of a design. They can be used to assert independence of processes. Section 8.4.3 illustrates such an application.

2. The statically computed race conditions improve the performance of partial order reduction, which results in a greatly reduced number of interleavings. The remaining interleavings can then be explored exhaustively, which is a valuable validation aid.

We have implemented this technique in SCOOT [Blanc et al., 2008], a novel research compiler for SystemC. The static computation of the race conditions relies on the Model Checking engine of SATABS [Clarke et al., 2005], a SAT-based model checker implementing predicate abstraction (the technique we propose is independent of the specific formal engine, however). Our experimental results indicate that strong race conditions can be computed statically at reasonable cost, and result in a simulation speedup of a factor of ten or better.

8.2 Introductory example

Program 8.1 serves as running example and illustrates the need for a Model Checking approach. The module *m* declares two processes *guard* and *increment*. The process *guard* watches the value of shared variable *pressure*, which shall not exceed the value *PMAX* and is incremented by process *increment*. Both processes

are sensitive to the clock signal *clk*. The semantics of the SystemC scheduler guarantees that a method process is executed without interruption up to the point where it returns. Thus, the scheduler has to choose either the scheduling sequence $(guard; increment)$ or $(increment; guard)$ each time the clock is updated. Consequently, the pressure can exceed the limit if its value reaches *PMAX* and process *increment* is triggered before *guard*. It is clear that the number of traces grows exponentially with the number of clock cycles. As a result, systematic exploration of all interleavings rapidly becomes unmanageable, and the bad behavior might go unnoticed.

A conventional static analysis can discover that *guard* reads the pressure and that *increment* modifies the pressure, concluding that the processes are indeed dependent and that all interleavings must be explored. In a similar way, a conventional dynamic analysis can observe at runtime that *guard* reads the pressure and that *increment* modifies the pressure, concluding that the alternative schedule needs to be explored. However, such analyses fail to detect that *guard* and *increment* are commutative in most cases. Our tool uses a Model Checker to compute the weakest predicate over the pre-state variables that guarantees the absence of races between the processes. In this example, it is easy to see that the execution of *increment* and *guard* is commutative if and only if

$$pressure \neq PMAX - 1 \quad \wedge \quad pressure \neq PMAX$$

holds. SCOOT generates a simulator for the systematic exploration of the state space that checks this condition at runtime to avoid exploring redundant schedules.

8.3 Implementation

8.3.1 A Scheduler with Partial Order Reduction

Algorithm 8.1 is SCOOT's implementation of the evaluation phase. In contrast to the related work, *evaluation_phase* schedules runnable processes using information *statically* collected to reduce the number of interleavings explored. We are not aware of tools that compute equally strong conditions statically.

The evaluation phase terminates once the set of runnable processes is empty. The algorithm performs partial order reduction using persistent sets and sleep sets, and is a variation of techniques presented in [Godefroid, 1996]. On line 3, the procedure calls the function $runnable()$ to check if the set of runnable processes is empty before proceeding to the next iteration.

At simulation time, the scheduler calls *get_pers* to compute the set *persistents* of persistent processes. The subsequent part of the algorithm uses the set *sleeps*, declared outside the main loop on line 2, to perform partial order reduction. On line 5, the set *awakes* consists of the persistent processes *not* in *sleeps*. If the set

Algorithm 8.1 Evaluation Phase: the commutativity condition checked by commutative(p_i, p_j) is a predicate over states computed statically at compile-time.

```
   void evaluation_phase()
2    Set sleeps := ∅;
     while (runnable()≠∅ ) do
4      persistents := get_pers();
       awakes := persistents \ sleeps;
6      if(awakes=∅) then exit(0);
       Map next_sleeps; // Process -> Set
8      for all (Process p_i ∈ awakes) do
         for all (Process p_j ∈ sleep) do
10         if(commutative(p_i,p_j))
             next_sleeps[p_i] := next_sleeps[p_i]∪{p_j};
12       end for
         sleep := sleep∪{p_i};
14     end for
       Process p := nondet_select(awakes);
16     run(p);
       sleeps := next_sleeps[p];
18   end while
```

of awaken processes is empty (line 6), then other traces are covering all subsequent behaviors, and therefore, the simulator stops the execution. Otherwise, the scheduler computes the sleep sets for the next iteration using the map *next_sleeps*, which maps processes to a set of processes (lines 7–14). One line 10, the call to *commutative* returns *true* if the processes p_i and p_j are commutative in the current state. The scheduler reduces the computation of conditional independence to the computation of commutativity conditions by considering that all the processes are always enabled – if $\rho \notin Runnable(s)$, then this is interpreted as $s \xrightarrow{\rho} s$. This way, two processes are independent in the current state if and only if they are commutative in this state. SCOOT relies on Model Checking to compute a conservative condition that guarantees commutativity of the processes in the current state; the details of this pre-computation are presented in the following subsection. In contrast, traditional approaches need to rely on either executing the processes to determine which transitions are independent in the current state, which adds overhead, or on an imprecise data-flow analysis.

Finally, in lines 15–17, the scheduling algorithm nondeterministically runs a process from *awakes* and computes the sleep set of the next iteration.

Algorithm 8.2 computes the set of persistent processes and is the implementation of the function *get_pers()*. On line 2, the set $persistents$ of persistent processes is initialized with the runnable processes. Subsequently, the algorithm removes all independent processes from $persistents$ to defer their execution, reducing thus the nondeterminism of the system. On line 7, a call to the function *independent()*

Algorithm 8.2 Computation of persistent sets. The call to independent(p_i, p_j) returns true if the commutativity condition of p_i and p_j is equivalent to *true*. This information is computed at compile time using Model Checking.

```
Set get_pers()
 Set persistents := Runnable();
 for all(Process p_i ∈ Runnable()) do
  Bool pers := false;
  for all (Process p_j) do
   if( p_j ≥ p_i ) then continue;
   if(independent(p_i,p_j) = false) then
    pers := true; break;
  if( pers = false)
   persistents := persistents\{p_i};
 if(persistents = ∅) then
  return select_first(runnable());
return persistents;
```

returns *true* if the processes p_i and p_j are independent, i.e., the commutativity condition for p_i and p_j is equivalent to true. If $persistents$ is empty at the end of the computation, the algorithm deterministically returns a runnable process using the function $select\_first()$. Otherwise, $persistents$ is returned.

8.3.2 Computing the Process Commutativity Conditions

Program 8.2 Harness for the analysis of race conditions for a given pair of processes p1 and p2. The pre-condition ϕ is true initially, and is iteratively strengthened by the algorithm in Fig. 8.2.

```
assume(φ);
s_0 := current_state;
p_1(); p_2();
s_1,2 := current_state;
current_state := s_0;
p_2(); p_1();
s_2,1 := current_state;
assert(s_1,2 ≠ s_2,1);
```

We present an iterative technique to compute the commutativity condition for a given pair of processes p_1 and p_2 based on formal analysis. The condition is checked during simulation by Alg. 8.1. In general, SystemC processes need not terminate, and thus computing the strongest possible commutativity condition for a given pair of processes p_1 and p_2 is undecidable. We compute a conservative approximation by applying a Model Checker to the harness given as Program 8.2.

The basic idea of the harness is to run $p_1(); p_2()$, and compare the result with the result of running $p_2(); p_1()$ on the same initial state. The harness operates as follows: Initially, ϕ is set to *true*. The assume statement in the first line restricts the search to states that satisfy ϕ. Then the values of the visible variables are stored in s_0, the pair of processes $p_1(); p_2()$ is run, and the state is stored in $s_{1,2}$. The state is restored to s_0, and $p_2(); p_1()$ is run. The state is stored in $s_{2,1}$.

SCOOT passes the harness to a Model Checker to check the reachability of the last line, which is modeled by means of an assertion. If the Model Checker returns a counterexample, we have a trace π with an initial state satisfying the initial condition ϕ, passing through both processes, and ending in a state that violates the assertion. The path therefore begins in a state in which the two processes are commutative. SCOOT then computes the weakest precondition of $s_{1,2} = s_{2,1}$ alongside that path. Let P_π denote this condition. The executions of $p_1(); p_2()$ and $p_2(); p_1()$ from a state s that satisfies P_π:

1. terminate and
2. yield an equal state.

Consequently, P_π is an under-approximation of the commutativity condition for p_1 and p_2. At this point, SCOOT strengthens ϕ using $\neg P_\pi$, yielding ϕ'. This removes the trace π and any trace similar to π that goes through the same control locations. SCOOT iterates this process until the Model Checker stops reporting counterexamples. At this point, the predicate $P = \bigvee_\pi P_\pi$ represents the weakest condition such that the executions of $p_1(); p_2()$ and $p_2(); p_1()$ terminate and that p_1 and p_2 are commutative.

8.3.3 The Running Example

We illustrate the execution of the strengthening loop presented in Figure 8.2 on Program 8.1. In the first iteration, the verification engine verifies Program 8.3.1, and reports a counterexample π following the lines $\{1, 2, 3, 5, 6, 7, 8, 9, 11\}$.

Subsequently, we use classic Floyd-Hoare logic [Hoare, 1969, Floyd, 1967] to compute the weakest pre-condition of $s_{1,2} = s_{2,1}$ alongside π. We use the following axiom schmata for assignments and assume statements, in backward reasoning style:

$$\frac{}{\{P[x/E]\}\ \texttt{x := E;}\ \{P\}}\ \text{R0} \qquad \frac{}{\{P \wedge C\}\ \texttt{assume(C);}\ \{P\}}\ \text{R1}$$

We provide in Figure 8.1 the computation of the pre-condition P_π alongside π. The proof starts from Line 11 with the condition $s_{1,2} = s_{2,1}$.

Subsequently, we strengthen the set of initial sates in Program 8.3.2 with $\neg P_\pi$ to block any counterexample alongside the same path. In this example, the strengthening loop terminates after one iteration, yielding the predicate $P \triangleq \mathit{pressure} +$

Program 8.3 Harnesses verified during the first and second strengthening iteration in Figure 8.2. Initially, the pre-condition ϕ over `pressure` is *true*. In the second iteration, the pre-condition is set to $pressure{+}1 = PMAX \vee pressure = PMAX$.

```
1  assume(true);
2  s0 := pressure;
3  if(pressure = PMAX)
4      pressure := PMAX-1;
5  pressure := pressure + 1;
6  s1,2 := pressure;
7  pressure := s0;
8  pressure := pressure + 1;
9  if(pressure = PMAX)
10    pressure := PMAX-1;
11 s2,1 := pressure;
12 assert(s1,2 ≠ s2,1);
```

(1) Initial harness.

```
1  assume(pressure + 1 = PMAX ∨
     pressure = PMAX);
2  s0 := pressure;
3  if(pressure = PMAX)
4      pressure := PMAX-1;
5  pressure := pressure + 1;
6  s1,2 := pressure;
7  pressure := s0;
8  pressure := pressure + 1;
9  if(pressure = PMAX)
10    pressure := PMAX-1;
11 s2,1 := pressure;
12 assert(s1,2 ≠ s2,1);
```

(2) Harness in the second strenghtening iteration.

$1 \neq PMAX \ \wedge \ pressure \neq PMAX$. Observe that our technique enumerates only a subset of feasible paths. Given a state s, this predicate evaluates to *true* if the processes *guard* and *increment* are independent in s.

8.3.4 Implementation of the Strengthening Loop

In the following, we elaborate on our integration of the strengthening loop into two Model Checking engines, SATABS and CBMC. Note that our approach is independent of the particular Model Checking engine. The general idea can be extended in different directions: In Figure 8.2, we use the Model Checker to enumerate terminating paths and to ensure progress. In a similar spirit, we can adapt the strengthening loop to operate on infinite traces using a Model Checker for liveness properties such as Terminator [Cook et al., 2006], or we can replace the Model Checker with a testing engine to discover terminating traces – in which case the initial assume statement in Program 8.2 would be ignored.

Strengthening using Predicate Abstraction

Predicate Abstraction is a technique that abstracts a transition system by mapping sets of concrete states to a new, smaller abstract state space in a way that conserves the relevant behaviors of the system [Graf and Saïdi, 1997, Ball and Rajamani, 2000a]. Each predicate in the abstract model is represented by a Boolean variable, while the original variables are removed. The abstract program is created

Figure 8.1 Computation of the pre-condition $P_\pi \triangleq pressure + 1 \neq PMAX \ \wedge pressure \neq PMAX$. The proof starts from Line 11 with the post-condition $s_{1,2} = s_{2,1}$.

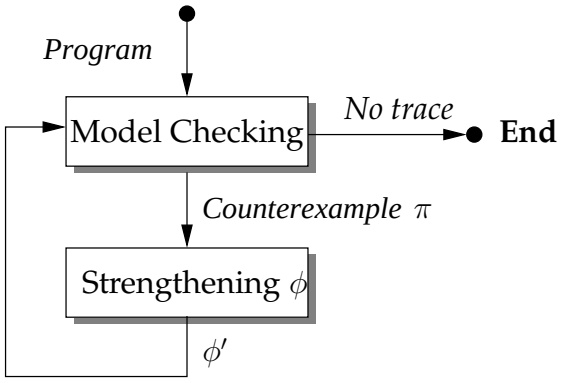
1	`assume(true);`	
	$\{pressure + 1 \neq PMAX \ \wedge \ pressure \neq PMAX\}$	
2	s_0 `:= pressure;`	$R0$
	$\{pressure + 1 = s_0 + 1 \ \wedge s_0 + 1 \neq PMAX \ \wedge \ pressure \neq PMAX\}$	
3	`assume(pressure` $\neq$ `PMAX);`	$R1$
	$\{pressure + 1 = s_0 + 1 \ \wedge s_0 + 1 \neq PMAX\}$	
5	`pressure := pressure + 1;`	$R0$
	$\{pressure = s_0 + 1 \ \wedge s_0 + 1 \neq PMAX\}$	
6	$s_{1,2}$ `:= pressure;`	$R0$
	$\{s_{1,2} = s_0 + 1 \ \wedge s_0 + 1 \neq PMAX\}$	
7	`pressure :=` s_0`;`	$R0$
	$\{s_{1,2} = pressure + 1 \ \wedge pressure + 1 \neq PMAX\}$	
8	`pressure := pressure + 1;`	$R0$
	$\{s_{1,2} = pressure \ \wedge \ pressure \neq PMAX\}$	
9	`assume(` `pressure` $\neq$ `PMAX);`	$R1$
	$\{s_{1,2} = pressure\}$	
11	$s_{2,1}$ `:= pressure;`	$R0$
	$\{s_{1,2} = s_{2,1}\}$	

using existential abstraction, which is a conservative abstraction for reachability properties. If the property holds on the abstract model, it also holds on the original program. In case a trace in the abstract model violates the property, the feasibility of the counterexample must be tested in the concrete model. If the counterexample can be simulated on the original program, it is reported to the user. The counterexample is called *spurious* if it does not correspond to a concrete trace. In that case, a refinement procedure adds new predicates in a way that removes the spurious trace. This is automated by *Counterexample Guided Abstraction Refinement* (CEGAR) [Clarke et al., 2000] and promoted by the Model Checker SLAM [Ball and Rajamani, 2002]. Predicate abstraction has been applied to SpecC [Clarke et al., 2007] and SystemC [Kroening and Sharygina, 2005]. Figure 8.3 shows the integration of our technique into SATABS. After strengthening, SATABS retains the abstract model obtained during previous iterations.

Strengthening using BMC

In *Bounded Model Checking* (BMC), a program and a specification are jointly unwound up to a given bound k to form a formula that is satisfiable if and only if the program paths with length k can violate the specification [Biere et al., 2003]. This formula is then passed to a SAT solver. In case the formula is satisfiable, the Model Checker constructs a counterexample for the original program from the

Figure 8.2 Iterative computation of the process commutativity condition using a Model Checker. Condition ϕ represents the set of initial states. The loop strengthens ϕ until the Model Checker stops reporting counterexamples.



satisfying assignment. The method is complete only if k exceeds the completeness threshold [Kroening and Strichman, 2003].

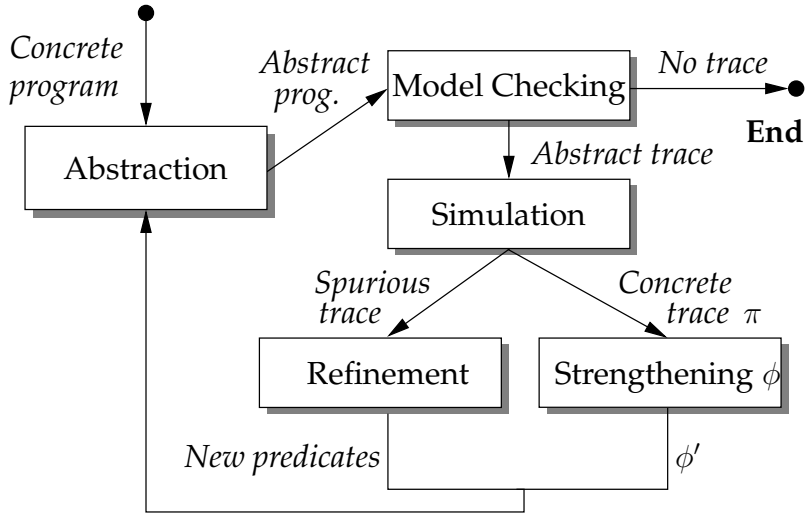
We use CBMC as Bounded Model Checker [Kroening et al., 2004]. In some cases, the symbolic simulator within CBMC is able to determine a sufficient depth automatically; otherwise, it inserts assertions to verify that k is sufficiently large. CBMC combines Bounded Model Checking with slicing techniques to remove statements unrelated to the property that is checked.

Figure 8.4 illustrates the integration of our technique into CBMC. First, CBMC unrolls Program 8.2 and builds a CNF formula for checking the reachability of the assertion on the last line. Upon discovery of a counterexample π, we compute the weakest precondition P_π alongside π and strengthen the set of initial sates by adding blocking clauses to the Boolean formula. Note that the SAT solver is operated in an incremental fashion, which allows it to retain all the clauses learned in previous iterations.

8.3.5 Model Checking SystemC Threads

The construction of the harness presented in Program 8.2 is straightforward for method processes, as no context switch is taking place. In contrast, a thread can suspend its execution using wait statements. In order to facilitate the analysis of threads, we use the conversion technique presented in Section 7.4.1 to simplify context switches. Using this technique, SCOOT suppress the stacks of threads and implements context switches with goto statements. This approach enables SCOOT to handle threads in the same way as method processes (see Prog. 7.5).

Figure 8.3 Iterative computation of the process commutativity condition using predicate abstraction.



8.4 Experimental Evaluation

In this section, we evaluate the benefits of integrating our partial-order reduction into a simulator that examines all schedules exhaustively using a backtracking search. We also quantify the cost of the computation of the commutativity condition using Model Checking.

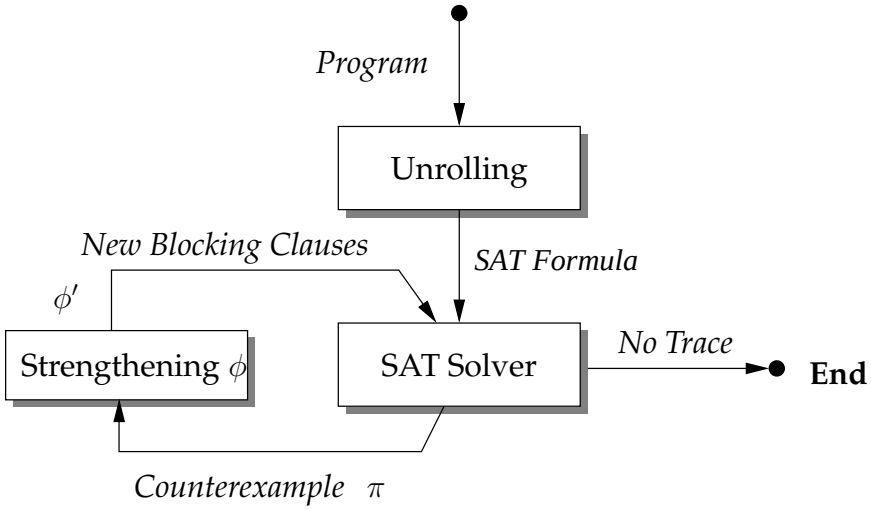
The experiments that we present are difficult instances. Commutativity of processes depends on control flow and data, and the computation of the condition is susceptible to the state-space explosion problem. As a first step, our tool performs a light-weight data-flow analysis to detect independent processes. This reduces the burden on the heavy-weight verification engines. As a result, SCOOT needs to run a Model Checker only on very few pairs of processes per design. All our results are obtained using a 3 GHz Linux machine. We make the benchmarks and the tool available for experimentation by other researchers at `www.cprover.org/scoot/`.

8.4.1 The Running Example

We continue our running example (Program 8.1). Figure 8.5 depicts the number of explored traces as a function of the number of simulation steps using iterative strengthening *(Full Precision)*. We set *PMAX* to *10*. The number of explored traces corresponds to the number of backtracks. Our simulator performs a state-less search, that is, the simulator replays transitions to backtrack.

Using our technique, the number of traces explored during simulation grows only quadratically with the number of steps, instead of exponentially. Note that at runtime there is always a data dependency between the processes *guard* and

Figure 8.4 Iterative computation of the process commutativity condition using Bounded Model Checking.



increment: Process *guard* always reads *pressure* and process *increment* always writes to *pressure*. Consequently, traditional dynamic partial order reduction techniques achieve no reduction in this example.

8.4.2 State Machines

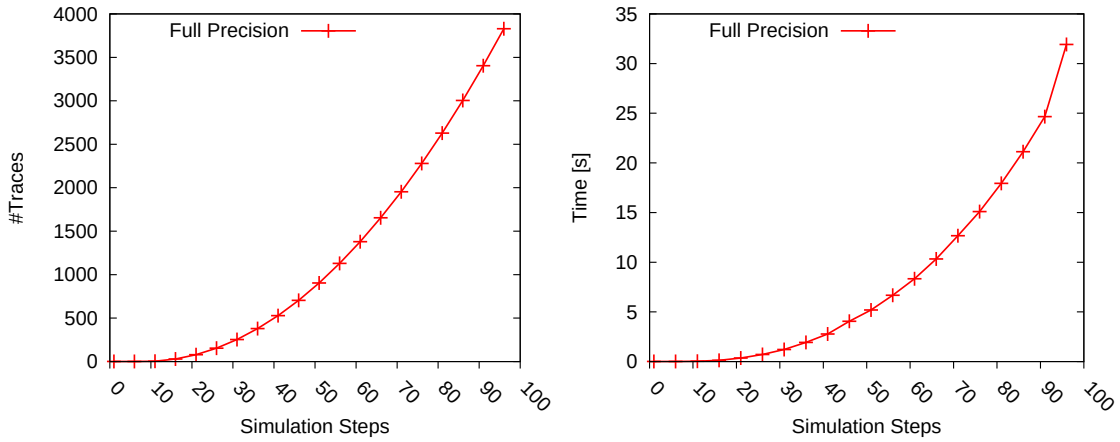
State machines are a typical ingredient of SystemC models. We use two different benchmarks of this kind.

The first benchmark (B1) consists of a synchronous model with three dependent processes. One process plays the role of a server waiting for requests, while the other two compete for access to the service. Program 8.4 contains the skeleton of the benchmark. When triggered, the clients and the server execute functions *process_client* and *process_server*, respectively. The clients communicate with the server via two shared variables *op* and *locked*. If *locked* is set, then the server is busy processing the request *op*. Otherwise, the clients compete for access to the service. The processes are sensitive to a clock.

Figure 8.6 compares the number of explored traces (simulator backtracks), and the total exploration time as a function of the number of simulation steps. We compare the precision of the commutativity conditions obtained by the Model Checking engines ("Full Precision") and the light-weight static analysis ("Precision 0"). The exploration time is limited to thirty minutes (1800 seconds).

We observe that our precise analysis results in a reduction of both the number of explored traces and the exploration time by about two to three orders of magnitude. Using our technique, the simulator can exhaustively cover all the relevant behaviors up to fifteen simulation steps in less than thirty minutes, whereas the simulation using the light-weight analysis already times out after seven simula-

Figure 8.5 Total time and number of traces explored at runtime as a function of the number of simulation steps, for the running example.



Program 8.4 Skeleton of Benchmark B1

```
bool locked; int op;
void process_client() {
 if(!locked){ op=get_pid(); locked=true;}
}
void process_server(){
  switch(state) {
  ...
  case Idle: {switch(op) {...} break;}
  case End: {state = Idle; locked = false;}
  }
}
```

tion steps.

Our second state-machine benchmark (B2) consists of two synchronous state machines communicating via shared variables. The model has three interdependent processes, which are sensitive to the clock. The state machines are implemented using case switches. Figure 8.7 is a comparison of the simulation times and the number of explored traces. The reduction is in the order of one magnitude.

We quantify the additional cost of obtaining the full precision dependency conditions prior to simulation. For each pair of processes, Table 8.1 shows the number of strengthening iterations and the time required for the static analysis running SATABS and CBMC. The difference in the number of strengthening iterations required by SATABS and CBMC is due to code transformations inherent to BMC.

The additional cost for B1 is negligible using either SATABS or CBMC. The re-

Figure 8.6 Performance effect of static partial-order reduction on B1

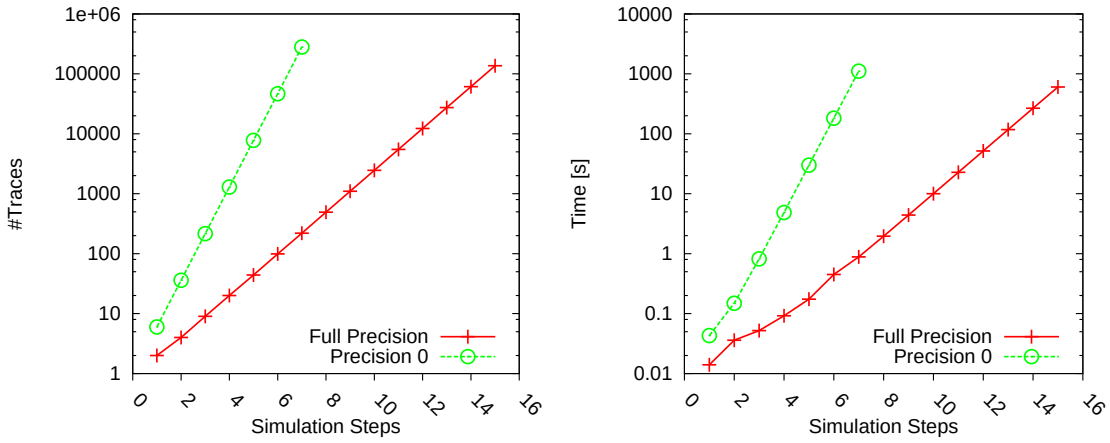


Figure 8.7 Performance effect of static partial-order reduction on B2

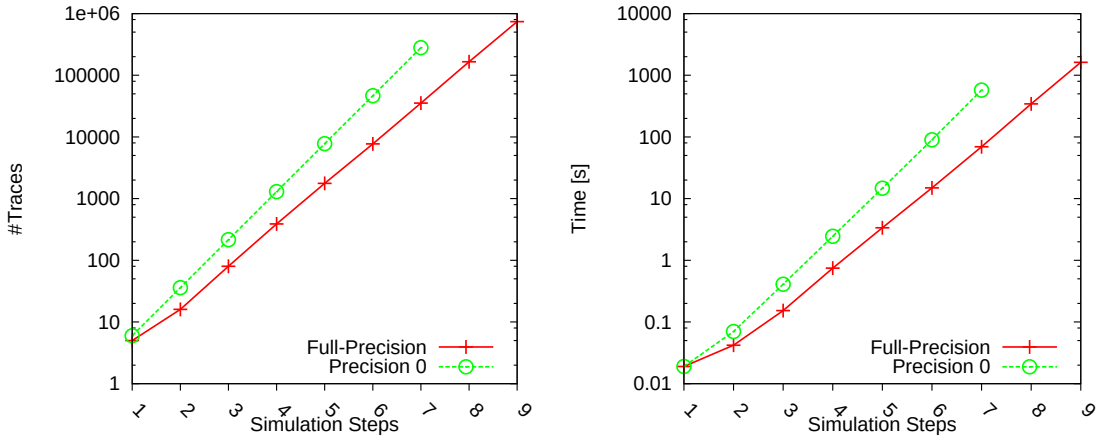


Figure 8.8 Details of the strengthening process for the analysis of the memory model (Program 8.5). Each figure corresponds to a distinct verification task and gives the time for the individual strengthening iterations using SATABS. The last iteration proves the absence of any further counterexamples.

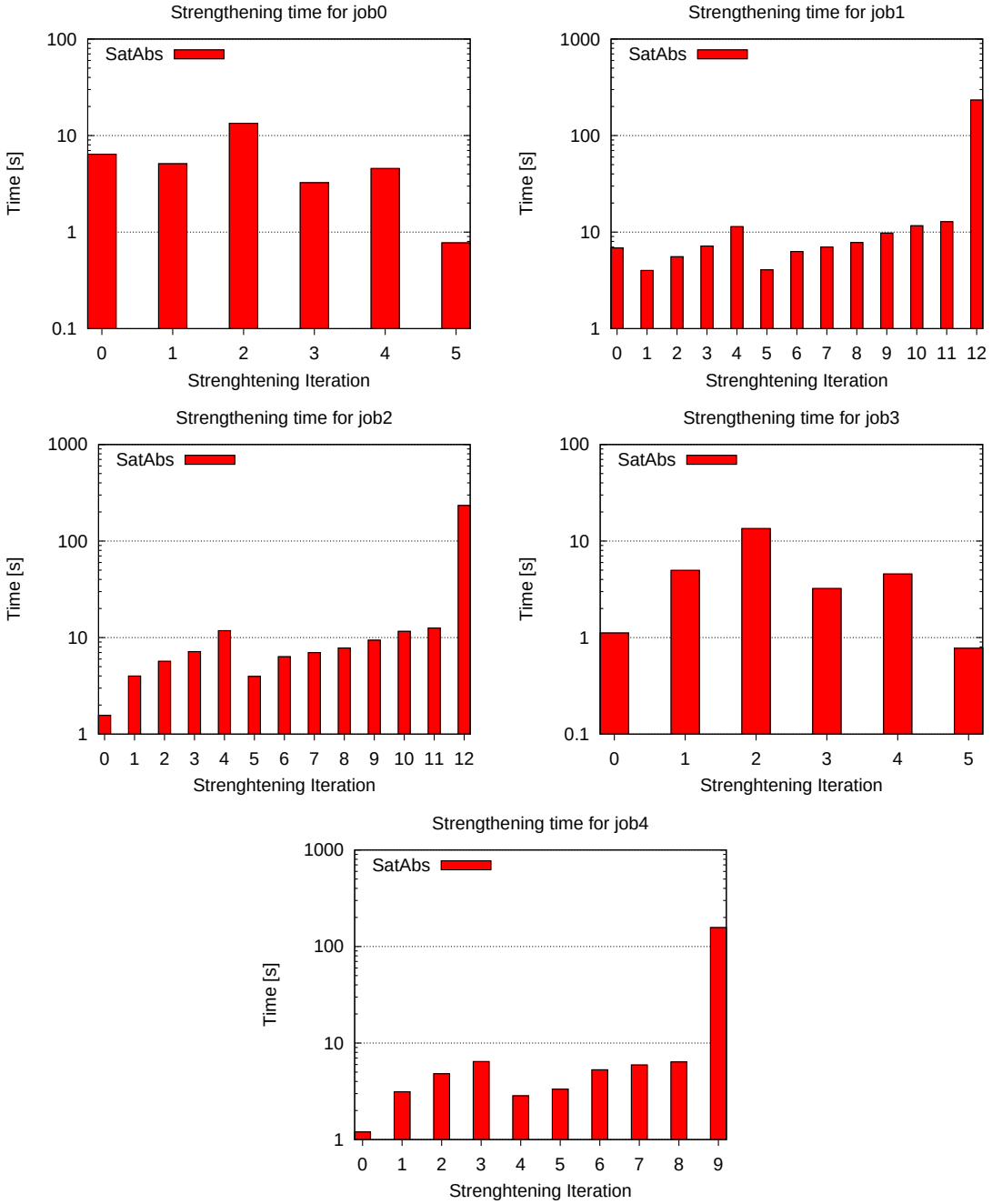


Figure 8.9 Details of the strengthening process for the analysis of the memory model presented in Program 8.5. Each figure corresponds to a distinct verification task and gives the time for the individual strengthening iterations using CBMC. The last iteration proves the absence of any further counterexamples.

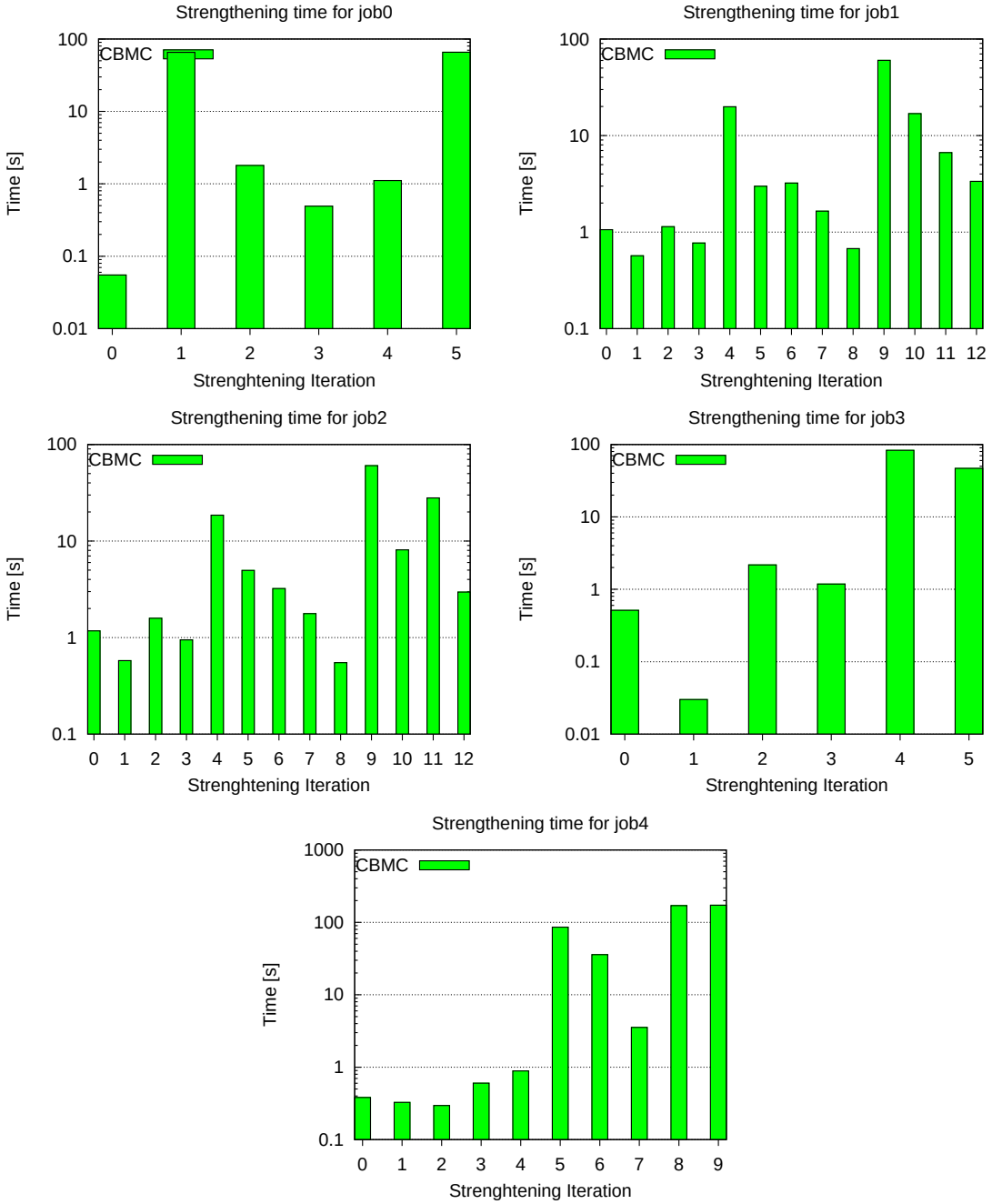


Figure 8.10 Runtime for the RISC-CPU model.

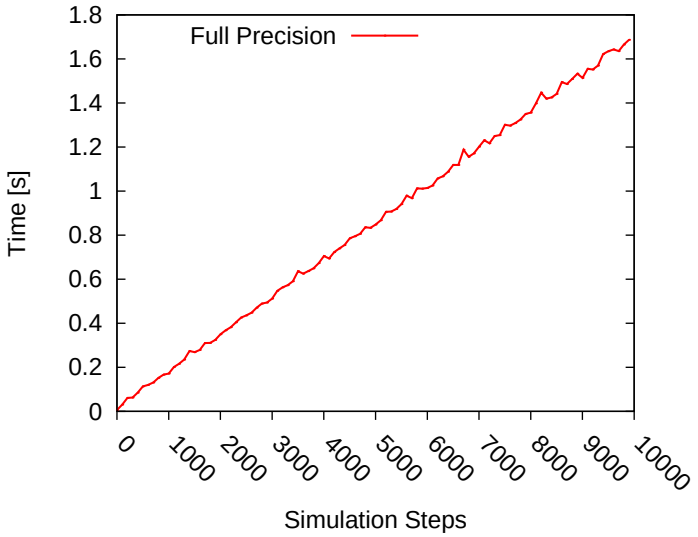


Figure 8.11 Impact of the precision of the static analysis on the performance of the simulation on the RISC-CPU model. The precision of the analysis is measured in terms of the bound on the number of strengthening iterations. "Precision 0" stands for the light-weight static analysis. The highest precision on this model is six. A precision of two is already sufficient to obtain an optimal simulation.

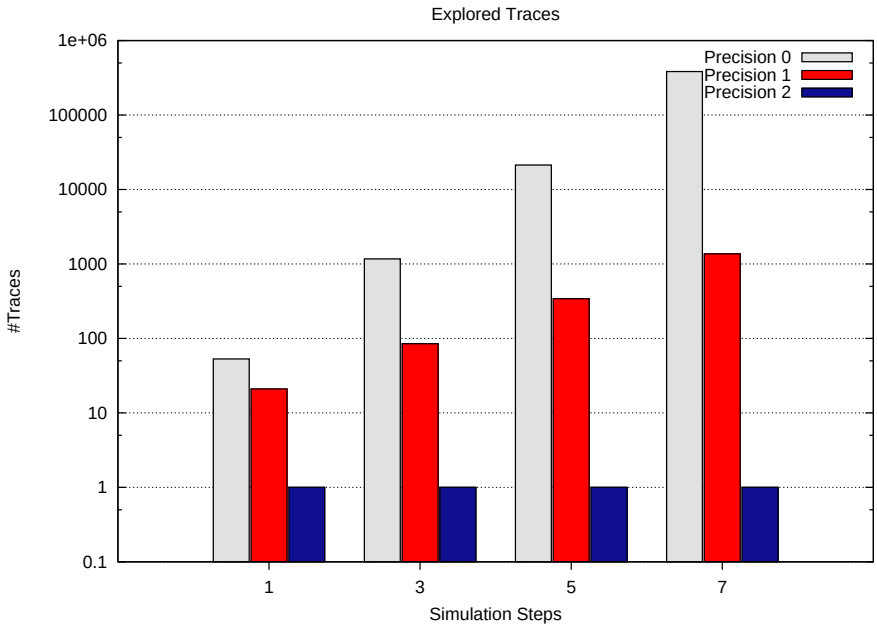


Table 8.1 Time to compute the race conditions for the state-machine benchmarks for each of the process-pairs (jobs) using SATABS and CBMC. The timeout is set to sixty minutes.

Benchmark	Jobs	SATABS		CBMC	
		#Strength.	Time[s]	#Strength.	Time[s]
B1	Job0	2	2.51	2	< 1
B1	Job1	10	12.40	9	1.86
B1	Job2	10	11.61	9	1.88
B2	Job0	44	437.75	-	TO
B2	Job1	19	84.67	4	13.37
B2	Job2	12	71.03	4	2246.48

sults for B2 indicate that the choice of the verification engine is important: CBMC is faster than SATABS on the second pair of processes but times out on the first, whereas SATABS provides a result within two minutes. Note that the computation of these conditions can be distributed onto multiple machines, as the computation for each pair of processes is independent. Furthermore, the precision of the analysis can be controlled by bounding the number of strengthening iterations, which yields a conservative approximation. Finally, as demonstrated by the simulation runs, the time required for a full exploration grows exponentially with the number of simulation steps, and therefore, the time spent for a precise static analysis eventually pays off.

8.4.3 An Asynchronous Dual-Port Memory

We present an instance that is difficult for any dependency analysis. Memory modules are frequently modelled using nondeterminism, as the priorities of read and write operations are often left unspecified. Memories are widely employed in system designs to implement communication buffers, caches, and register files. We evaluate our technique using a model of an asynchronous dual port memory. The model has four method processes. Program 8.5 illustrates the structure of the memory module. The memory model is implemented as an array of unsigned integers (line 5). This array is shared among the four processes *rd0*, *wr0*, *rd1*, and *wr1* (lines 6 to 9). These processes are sensitive to control signals that trigger the different operations. We provide the body of the functions *rd0* and *wr1* on lines 13 and 18, respectively. The functions *rd1* and *wr1* are implemented in a similar way.

For each pair of processes, Table 8.2 shows the time required for the static analysis using SATABS and CBMC. Additionally, the second column indicates whether the outcome of the analysis yields a predicate equivalent to *true*; that is, this column indicates whether the processes are completely independent. This information provides valuable insight into the behavior of the memory. For in-

Program 8.5 A Model for Asynchronous Dual-Port Memory

```
1  SC_MODULE(ram) {
   ...
3  sc_in<bool> cs0, cs1, oe0, oe1, we0, we1;
   sc_inout<unsigned> data0, data1;
5  sc_uint<DATA_WIDTH> mem [RAM_DEPTH];
   void rd0();
7  void wr0();
   void rd1();
9  void wr1();
   };
11
   void ram::rd0() {
13   if (cs0.read() && oe0.read() && !we0.read())
       data0 = mem[address0.read()];
15 }

17 void ram::wr0() {
     if(cs0.read() && we0.read())
19     mem[address0.read()] = data0.read();
   }
21 ...
```

stance, our static analysis is able to show that the processes *rd0* and *wr0* are independent; the same holds for the processes *rd1* and *wr1*. In both cases, write accesses have priority over read accesses. However, the processes *rd0* and *wr1* are not independent, and neither are *rd1* and *wr0*. Therefore, the effects of two concurrent read and write operations using different ports may depend on the scheduling order of the processes. Typically, this situation arises if both accesses address the same memory location, in which case the read operation can either retrieve the old value or the new one.

Figures 8.8 and 8.9 depicts the time for each strengthening iteration using SATABS and CBMC, respectively. The last strengthening iteration is used for proving the absence of additional counterexamples. Job1 and Job2 using SATABS spend most of the time in the very last strengthening iteration to prove that the assertion of the harness holds (finding a bug is usually easier than proving correctness). Note that we have designed our algorithm to compute a sequence of safe under-approximations of the commutativity condition. Consequently, the user can stop any excessively long computation and proceed in a sound way with partial results. This enables a trade-off between time and precision. Furthermore, skipping the last strengthening iteration results in no loss of precision.

A proof of independence of processes requires a formal analysis. For instance, to discover that the processes *rd0* and *wr0* are two mutually exclusive operations,

a static analyser must prove that on lines 13 and 18, the guards of the if-statements cannot be satisfied simultaneously; this is a fact that a verification engine based on predicate abstraction can easily establish by tracking the value of signal *we0*.

Table 8.2 Runtime and number of iterations required to compute the race conditions for each of the process-pairs. The column $P \Leftrightarrow true$ indicates whether the processes are proven independent; that is, whether condition P is equivalent to *true*.

Jobs	Proc.	SATABS			CBMC		
		$P \Leftrightarrow true$	#Strth.	Time [s]	$P \Leftrightarrow true$	#Strth.	Time [s]
Job0	wr0, rd0	yes	6	34.39	yes	6	204.08
Job1	wr0, rd1	no	13	344.14	no	13	203.53
Job2	wr1, rd0	no	13	339.26	no	13	219.24
Job3	wr1, rd1	yes	6	28.98	yes	6	203.97
Job4	wr1, wr0	no	10	221.24	no	10	501.96

8.4.4 A RISC Processor

Table 8.3 Runtime and number of iterations required to compute the race conditions for the RISC-CPU model. The column $P \Leftrightarrow true$ indicates whether the condition P is equivalent to *true*.

Jobs	SATABS			CBMC		
	$P \Leftrightarrow true$	#Strength.	Time [s]	$P \Leftrightarrow true$	#Strength.	Time [s]
Job0	no	7	42.02	no	7	3.41
Job1	no	7	42.02	no	7	3.42
Job2	yes	5	42.36	yes	5	4310.85
Job3	yes	5	42.34	yes	5	4253.48
Job4	yes	5	32.244	yes	5	3986.19

We demonstrate the scalability of our race-analysis using an updated version of the RISC-CPU model that is shipped with the SystemC library. The processor has nine modules, which include an MMX and a floating-point unit. In total, the model declares fourteen processes and contains 2153 lines of C++ code.

Table 8.3 quantifies the computational cost of the verification tasks that SCOOT generates. Out of 91 pairs of processes, light-weight static analysis can already refute 86 dependencies, leaving only five pairs for the heavy-weight engine. SCOOT can prove that three out of these five remaining pairs are completely independent and identifies the register file as a potential source of nondeterminism.

Comparing the performance of SATABS and CBMC, we observe that the latter is faster on the two first tasks, whereas SATABS clearly outperforms CBMC on

the remaining ones. The bad performance of CBMC on these tasks is caused by a specific loop in one of the processes.[1] To solve this issue, CBMC can replace a loop with an assume-false statement, which blocks any trace that reaches this location. CBMC then returns a (conservative) condition that is almost as precise as the original one within only $3\,s$ and four strengthening iterations. The loss of precision has no negative impact on the simulation performance in this example.

Figure 8.10 depicts the simulation performance. Using our precise conditions, the simulator explores only a single trace, and thus, the time for exhaustive simulation is linear and optimal. The simulator is able to exhaustively search ten thousand simulation steps in less than two seconds. In contrast, when relying exclusively on light-weight static analysis, the number of traces grows exponentially (Figure 8.11). This means that only shallow exploration is possible.

We also quantify the precision obtained by limiting the number of strengthening iterations. "Precision 0" indicates that no iterations are performed, i.e., the result of the light-weight analysis is used. "Precision 1" means that one strengthening iteration is performed, and so on. The number of traces grows exponentially when the precision is set to zero or one. The experiments indicate that a precision of two is already sufficient to achieve the maximal runtime reduction for this model. The strengthening loop terminates after seven iterations.

8.5 Bibliographic Notes

Concurrent threads with nondeterministic interleaving semantics may give rise to *races*. A data race is a special kind of race that occurs in a multi-threaded application when several processes enter a critical section simultaneously [Netzer and Miller, 1992].

Flanagan and Freund use a formal type system to detect race-condition patterns in Java. *Eraser* is a dynamic data-race detector for concurrent applications [Savage et al., 1997]. It uses binary rewriting techniques to monitor shared variables and to find failures of the locking discipline at runtime. Other tools, such as *RacerX* [Engler and Ashcraft, 2003] and *Chord* [Naik et al., 2006], rely on classic pointer-analysis techniques to statically detect data races.

Model Checkers are frequently applied to the verification of concurrent applications, and SystemC programs are an instance; see [D'Silva et al., 2008] for a survey on software Model Checking. Vardi identifies formal verification of SystemC models as a research challenge. Prior applications of formal analysis to SystemC or similar languages are indeed limited. We therefore briefly survey recent advances in the application of such tools to system-level software. *DDVerify* is a tool for the verification of Linux device drivers [Witkowski et al., 2007]. It places the modules into a concurrent environment and relies on SATABS for the verification. *KISS* is a tool for the static analysis of multi-threaded programs written in C [Qadeer and Wu, 2004]. It reduces the verification of a concurrent

[1] The loop sequentially resets all the entries of the register file.

application to the verification of a sequential program with only one stack by bounding the number of context switches. The reduction never produces false alarms, but is only complete up to a specific number of context switches. *KISS* uses SLAM [Ball and Rajamani, 2002], a Model Checker based on *Predicate Abstraction* [Graf and Saïdi, 1997, Ball and Rajamani, 2000a], to verify the sequential model.

Verisoft is a popular tool for the systematic exploration of the state space of concurrent applications [Godefroid, 2005] and could, in principle, be adapted to SystemC. The execution of processes is synchronized at *visible operations*, which are system calls monitored by the environment. *Verisoft* systematically explores the schedules of the processes without storing information about the visited states. Such a method is, therefore, referred to as a *state-less search*. *Verisoft*'s support for partial-order reduction relies exclusively on dynamic information to achieve the reduction. In a recent paper, Sen et al. propose a modified SystemC-Scheduler that aims to detect design flaws that depend on specific schedules. The scheduler relies on dynamic information only, i.e., the information has to be computed during simulation, which incurs an additional run-time overhead. In contrast, SCOOT statically computes the conditions that guarantee independence of the transitions. We can control the degree of precision of our analysis to compute very accurate predicates. The simulator can then test these conditions at runtime to detect reduction opportunities with little overhead.

Flanagan and Godefroid describe a state-less search technique with support for partial-order reduction. Their method runs a program up to completion, recording information about inter-process communication. Subsequently, the trace is analyzed to detect alternative transitions that might lead to different behaviors. Alternative schedules are built using *happens-before* information, which defines a partial-order relation on all events of all processes in the system [Lamport, 1978]. The procedure explores alternative schedules until all relevant traces are discovered. Helmstetter et al. present a partial-order reduction technique for SystemC. Their approach relies on dynamic information and is similar to Flanagan and Godefroid's technique 2005. Their simulator starts with a random execution, and observes visible operations to detect dependencies among the processes and to fork the execution. In contrast, our technique performs an extremely precise static analysis that is able to discover partial-order reduction opportunities not detectable using dynamic information alone. In addition, static analysis can reveal races that are not exercised during simulation and is able to formally refute process dependencies.

Kundu et al. propose to compute read/write dependencies between SystemC processes using a path-sensitive static analysis. At runtime, their simulator starts with a random execution in a way similar to Flanagan and Godefroid and detects dependent transitions using the static information computed previously. The novelty of our approach is to enhance this conventional, light-weight static analysis with Model Checking to compute sufficient conditions over the global variables of the SystemC model that guarantee commutativity of the processes.

Wang et al. introduce the notion of *guarded independence* for pairs of transitions. Their idea is to compute a condition (or guard) that holds in the states where two specific transitions are independent. Our contribution in this context is to compute these conditions for SystemC using a Model Checker.

8.6 Summary

We presented SCOOT, a novel compiler for SystemC that integrates static analysis and formal verification techniques in order to improve simulation performance. The structure of the SystemC model (the hierarchy and the port bindings) is computed at compile time by means of a data-flow analysis. We use a second data-flow analysis to perform a light-weight detection of independent processes. The next step is to invoke a modified software Model Checker on each pair of possibly dependent transitions in order to compute a sufficient condition for commutativity of the transitions. Our technique benefits from the fact that SystemC processes are not preempted, and thus, only few such pairs have to be checked. Note that the Model Checker is never applied to the entire model, but only to pairs of transitions – the static part of the analysis is therefore typically polynomial in the size and number of processes.

SCOOT uses the commutativity condition during simulation in order to eliminate unnecessary interleavings. Our analysis is fully automatic and requires no annotation of the source code by the user. Using Model Checking, our analysis is able to prove or refute process dependencies statically and to detect reduction opportunities not covered by other dynamic approaches at runtime.

The experimental results indicate that our formal race analysis produces valuable information for pruning the state space at runtime. To the best of our knowledge, this work uses the strongest conditions for commutativity of processes reported in the literature. Furthermore, the trade-off between precision and computational cost can be controlled, and the entire flow can be distributed on multiple machines.

9

Conclusion

SCOOT is the first research compiler for SystemC that combines static classic analysis and precise Model-Checking techniques. Given the extraordinary complexity of modern designs, verification is clearly the most time-consuming task of any design flow. Formal Methods have made enormous capacity progress since the eighties. Still, the size and the complexity of modern computer architectures is growing much faster.

In this thesis, we showed that it is possible to identify key problems that involve only fragments of large systems and to compute very precise conditions whose utilities go beyond verification, namely for simulation and testing. Race-analysis for SystemC is an excellent example of such a case: even so SystemC designs may contain thousands lines of code, processes taken pairwise are much more shorter, and their commutativity conditions can be computed in parallel.

Additionally given the complexity of the models and the surrounding environment, we have shown that it is useful to reduce the size of the models as much as possible for verification. *We have demonstrated with* SCOOT *that domain-specific compilers can be used to abstract away complex code-machinery that is extremely tedious to analyse statically and to exploit this information to significantly improve simulation performance.* The integration of verification procedures directly into the compilation flow has several advantages:

1. First, risks of mismatches between verification and simulation models are reduced;

2. second, the compiler can employ Formal Methods to optimize code; and third,

3. better integrated tools help designers to focus on their task rather on the details of different environments.

Finally, we employ Formal Methods only in last resort to make up for the imprecisions of light-weight static analysis techniques. We showed that formal techniques have applications outside their traditional scope, which is namely verification, for optimization purposes.

A

Verification of Concurrent Device Drivers

A.1 introduction

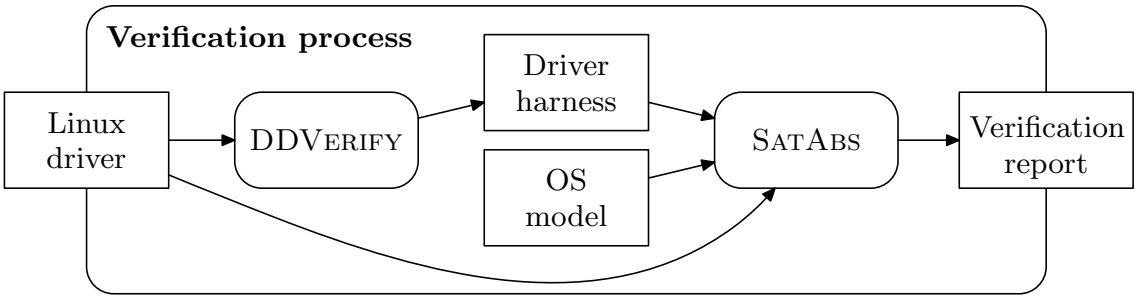
In an operating system, a device driver is software program that is responsible for controlling a peripheral. Verification of device-driver contains thus both software and hardware aspects. Device drivers interact with the low parts of the system API and run in kernel mode. Correct usage of the kernel API is, therefore, critical for the stability of the system. In this Chapter, we describe Model Checking using Predicate Abstraction in the context of device-driver verification, and introduces the challenges faced when verifying highly concurrent software.

We have built a tool called *DDVerify* [Witkowski et al., 2007] for the verification of Linux device drivers that employ formal engines based on predicate abstraction in a similar spirit as Microsoft's SLAM [Ball et al., 2004] project for the verification on Windows device drivers. In contrast to the execution model of SystemC, which provides strong atomicity guarantees, the execution model of operating systems such as Linux or Windows is inherently concurrent: Context switches can happen at any time and several processes can run simultaneously. Device-drivers run therefore in highly concurrent environment, and their verification is subject to the state-space-explosion problem induced by data races. Predicate abstraction succeeds in checking such properties by separating the control and the data of the program. This technique enables the detection of control-flow related bugs. Even though SLAM is able to verify properties related to locking, its scope is restricted to *sequential* programs. The most vicious bugs emerge in systems with threads that communicate via shared memory. These bugs are hard to comprehend, and it is almost impossible to reproduce them by means of testing. Fortunately, predicate abstraction can also be used to generate abstract models for multi-threaded programs: Attempts to integrate a model checker for concurrent abstract models into SLAM have been made, but not reported due to scalability issues and the lack of convincing benchmarks. Our contribution is that we integrate the model checkers Cadence SMV [K.L. McMillan, 1992] and BOPPO [Cook et al., 2005] into the predicate abstraction-based verification tool

SATABS [Clarke et al., 2005], thus enabling the verification of concurrent programs that communicate using shared memory. In order to *detach* the code of the device driver under test from the operating system, the service routines are replaced with a model that over-approximates their behavior. The device driver is then exposed to a hostile environment, simulated by a driver harness, trying to reveal "misuses" of the kernel API. These misuses are specified by means of assertions added to the operating system model.

The operating system as well as the driver harness is modeled using nondeterminism. An inaccurate model may result in falsely reported bugs that do not exist in the actual system. In particular, in the presence of threads it is essential to truthfully model the synchronization primitives of the operating system.

We provide a concurrent model of the relevant parts of the Linux kernel API. More information on our formalization of the Linux kernel is presented in [Witkowski, 2007]. We provide the fully automated verification tool *DDVerify*, which, given the source code of a Linux device driver, generates an appropriate driver harness and uses SATABS to check whether the driver violates the pre- or post-conditions of our kernel model. The following activity diagram shows the corresponding verification process:



We present benchmarks generated by applying *DDVerify* to concurrent programs and report two previously unknown bugs that were detected in Linux device drivers.

A.2 Predicate Abstraction in Presence of Concurrency

In the rest of this section,we write $\hat{T}$ to denote a finite state abstraction the original program T and T_ℓ to denote the transition relation at program location ℓ.

DDVerify attaches the driver to concurrent harness and verifies the whole model using SATABS. The concurrent harness uses a statically bounded number of threads. The abstraction algorithm can be applied without major modifications. The state space of the resulting concurrent abstract model is still finite and can therefore be checked using SMV. Unlike SMV, BOPPO is able to handle infinite dynamic thread creation by over-approximating the abstract transition relation [Cook et al., 2005]. Currently, *DDVerify* does not use this feature.

Linux provides different mechanisms to support thread creation and synchronization of threads. We support these functions by modeling them using only thread creation and atomic blocks. These primitives are also supported by Cadence SMV and BOPPO Therefore, the concurrency related transitions in T can be directly mapped to the corresponding primitives in $\hat{T}$.

The counterexample trace obtained by checking a multi-threaded abstract model may contain context switches. The model checker passes this information on to the feasibility checker by attaching a thread identifier to each transition in the counterexample. We modified the implementation of the feasibility checker such that a new stack is created whenever a new thread identifier is encountered in the counterexample. Naturally, a context switch involves saving the stack and program counter of the current stack and restoring the stack and program counter of the target thread. While adding threads to Boolean programs increases the complexity of the model checking phase significantly, adding threads to the counterexample trace does not make feasibility checking a harder problem. The refinement step is followed by another iteration of the abstraction refinement cycle.

A.2.1 Concurrency

In this section, we describe the modifications necessary to verify concurrent programs with predicate abstraction and CEGAR.

In the abstraction phase, each abstract transition relation $\hat{T}_\ell$ is computed independently for each location ℓ of the program. The context in which T_ℓ (and $\hat{T}_\ell$, respectively) is executed has no impact on the abstraction. Therefore, there is no necessity to modify the abstraction algorithm. The abstraction mechanism imposes the following restriction with respect to scheduling: Each transition in T_ℓ is considered to be executed atomically. A higher granularity can be achieved by a pre-processing step.

Linux provides different mechanisms to support thread creation and synchronisation of threads. We support these functions by modelling them using only thread creation and atomic blocks (see Section A.3). These primitives are also supported by Cadence SMV and BOPPO (even though the former does not support *dynamic* thread creation). Therefore, the concurrency related transitions in T can be directly mapped to the corresponding primitives in $\hat{T}$.

The decidability result for sequential abstract models mentioned in Section A.2 does not hold in the presence of multiple threads [Ramalingam, 2000]. In particular, the summarization approach used by SLAM's model checker BEBOP does not work in a concurrent setting. The explicit state model checker ZING [Andrews et al., 2004], which we integrated into SLAM, can handle unbounded thread creation, but may not terminate. Unfortunately, explicit state enumeration turns out to be an unsuitable technique for verifying the abstract models generated by predicate abstraction. The large number of non-deterministic transitions in these models leads to an immediate explosion of the state space [Cook et al., 2005]. Symbolic

model checking tries to overcome this problem by representing sets of states in terms of propositional formulæ. This approach was introduced by McMillan et al. [Burch et al., 1990] in 1990. His model checker Cadence SMV[1] is still competitive, but does not support unbounded thread creation or function calls. SATABS is able to use Cadence SMV as well as BOPPO as model checker for concurrent abstract models. BOPPO combines symbolic model checking with *partial order reduction*, a technique to reduce the number of interleavings that are considered. Partial order reduction is a common technique in explicit state model checkers (and is also used by ZING).

A.3 Modelling the Linux Kernel

In Linux, applications access the functions of a device driver via the file system. Each device has a corresponding entry in the `/dev` directory, and each driver may be in charge of several of these entries. The implementation of a device driver has to provide an initialisation function that is called when the device driver is loaded into memory. One of its duties is to provide information about the operations that are provided to user-space applications. These may depend on the type of the driver. Linux distinguishes between *character devices*, *block devices*, and *network devices*. Once the driver is initialized, it obtains a set of device numbers and registers the devices it manages. For each device, the driver provides a data structure that contains a collection of function pointers that point to the functions provided by the driver. The kernel translates accesses to the files in the `/dev` directory to calls of these functions. File operations such as `open`, `close`, `read`, and `write` trigger the execution of the corresponding function of the device driver. Finally, the driver has to provide cleanup functions that allow to unregister devices and exit the driver.

In order to exhaustively verify a device driver, it is necessary to consider all possible combinations of calls to the driver functions. For this purpose, *DDVerify* determines the type of the driver, the used interface (PCI), and the initialisation and cleanup function. Using this information, *DDVerify* automatically generates a test harness that initializes the driver, executes a loop in which it calls the driver functions non-deterministically, and finally calls the exit function of the driver. *DDVerify* is able to generate a sequential as well as a concurrent test harness. In the latter case, *DDVerify* generates a test harness that concurrently executes device driver functions and deferred calls to driver functions. Currently, *DDVerify* generates a harness that uses two threads for reasons of scalability. However, *DDVerify* is designed to be easily adaptable, and therefore, this restriction can be easily lifted (at the cost of a significant increase of the runtime of SATABS).

Linux device drivers make extensive use of the Linux kernel API. Due to the complexity and the size of the kernel it is not possible (and not our intention) to verify the device driver and the kernel as a compound program. Therefore, we

[1] Available from `http://www.kenmcmil.com`

provide a conservative operational model for all relevant kernel service routines, which is used to detach the driver from the kernel. The pre-conditions of these routines are enforced by means of assertions, i.e., the correctness properties that are verified by *DDVerify* are an integral part of the operating system model. An inaccurate model may result in spurious counterexamples. Since the Linux kernel lacks a formal specification, we define the semantics of the service routines we support using Milner's π-calculus [Milner, 1982]. A detailed description of this work can be found in [Witkowski, 2007]. In this section, we present the model of several kernel services exemplarily.

Mutual Exclusion Device driver code runs in parallel with other routines. Several sources of concurrency exists: tasklets, work-queues, handlers for timers, and interrupts are concurrently executed with the *main code* of the module, and device drivers have to support simultaneous access to their functions (i.e., the driver code must be *reentrant*). The API of the Linux kernel provides several mechanisms to protect critical sections of the driver code: *semaphores*, *spinlocks*, and *mutexes*. For the sake of simplicity, semaphores can be represented as variables with only the two values *locked* and *unlocked*. If a process wants to enter a critical section, a corresponding semaphore must be acquired for this purpose. Mutexes were introduced in the Linux kernel 2.6.17, and are semaphores implemented in a more efficient way. Spinlocks are busy-waiting semaphores meaning that the waiting processes are not put to sleep by the scheduler. *DDVerify* supports semaphores, spinlocks and mutexes, as well as interruptible and uninterruptible sleep.

As example, Program A.1 shows the body of the function `down_interruptible` that locks a semaphore. As the operation of locking a semaphore causes the process to sleep if the the semaphore is held, the call has to be issued within the context of a process (since the kernel itself is executed non-preemptively). This is asserted in line 3. Another pre-condition of the function is that the semaphore shall be initialized (line 8).

The internal data of the semaphore are shared resources. Hence, the critical sections are protected using atomicity. The special functions `atomic_begin`, and `atomic_end` mark the beginning and the end of an atomic section, and are interpreted accordingly by SATABS.

If the semaphore is not locked by another process, the current process locks it in line 16 and `down_interruptible` returns successfully. Otherwise, the process has to wait for the semaphore. In that case, a potential interrupt is simulated by using the non-deterministic function `nondet_int` in line 22. This function non-deterministically returns an integer value. If this value is non-zero, the value -1 is returned to indicate that the call to `down_interruptible` has been interrupted. The code loops until the lock is acquired or the operation is interrupted. Note that this implementation for locking a semaphore is based on busy-waiting, and thus, is meant for verification purpose only.

Checking deadlocks requires to argue about the ownership of a semaphore.

Program A.1 Example of the function *down_interruptible*

```
1  int down_interruptible(struct semaphore * sem)
2  {
3    assert_context_process();
4
5    atomic_begin();
6
7    assert(sem->init,
8              "Semaphore is  not initialized");
9
10   atomic_end();
11
12   do {
13
14    atomic_begin();
15    if(sem->locked  == 0) {
16       sem->locked = 1;
17       atomic_end();
18       return 0;
19     }
20     atomic_end();
21
22     if (nondet_int()) {
23         return -1;
24     }
25   }
26   while(TRUE)
27 }
```

Currently, SATABS and the underlying model checker for abstract models have no feature to retrieve the identifier of the currently running process. As in SLAM, we support a simple form of deadlock detection by checking in a sequential model whether the process is trying to lock a semaphore twice.

Deferring Tasks In certain situations it is necessary to defer work to a later point in time. For instance, critical kernel routines such as interrupt handlers disable new interrupts, and thus, should only run for the minimal amount of time possible. Vital responses are performed immediately, while the longer management tasks are delayed. The API of the Linux kernel provides several mechanisms to defer tasks: *softirqs*, *kernel timers*, *tasklets* and *work queues*. *DDVerify* currently supports the three latter ones, which are frequently found in driver codes.

Kernel timers allow to delay a task for at least a certain amount of time. Tasklets are similar to kernel timers, except that the execution of the delayed task

Program A.2 Implementation of work queues

```
1  int schedule_work(struct work_struct *work)
2  {
3    atomic_begin();
4    assert(work->init,
5                    "Work queue is initalized");
6    int i, slot = MAX_WORKQUEUES;
7    for (i = 0; i < MAX_WORKQUEUES; i++) {
8
9      if (shared_workqueue[i] == work) {
10       atomic_end();
11      return -1;
12     }
13
14     if (shared_workqueue[i] == NULL) {
15        slot = i;
16        break;
17     }
18   }
19   assert(slot == MAX_WORKQUEUES,
20                   "Work queue is full");
21
22   shared_workqueue[slot] = work;
23   atomic_end();
24   return 0;
25 }
```

is guaranteed to happen before the next timer tick. A tasklet can be executed in parallel with other tasklets, but is strictly serialized with respect to itself. Scheduling the same tasklet multiple times only triggers its execution once. Priorities may be assigned to tasklets but this this feature is currently unsupported. Finally, *DDVerify* also offers the possibility to disable or enable scheduled tasklets.

Work queues are very similar to tasklets. Tasklets run in *interrupt mode* and have to adhere to certain restrictions, most notably, scheduling, sleeping, and accesses to user space memory are strictly forbidden. Work-queue functions run in the context of a special kernel process, and the restrictions of tasklets do not apply.

As an example, we present the our model of the function `schedule_work`, in Figure A.2, which schedules a task using the default work queue of the kernel. The parameter `work` is a pointer to a structure representing the task to execute. In line 3, the implementation checks whether the object referenced by `work` is initial-

ized. Currently, we model the queue as an array a capacity of `MAX_WORQUEUES` tasks. The task is stored in the first empty slot of this queue (line 22). If the task is already present then the function returns the value -1 (line 11). The driver harness non-deterministically selects tasks from `shared_workqueue` and executes them in parallel to the driver functions.

Wait Queues Putting a process to sleep is an important technique in device driver programming. A wait queue is a simple list of processes waiting for an event to occur. In particular, *DDVerify* supports the macros `wait_event` and `wait_event_interruptible`.

Both macros take two parameters: The first parameter corresponds to a queue data structure, and the second is an arbitrary Boolean condition. The process is put to sleep until the condition holds. The interruptible version allows signals to wake up the process. In that case, the macro returns a non-zero value. Sleeping processes waiting for an event can be awaken using the function `wake_up`.

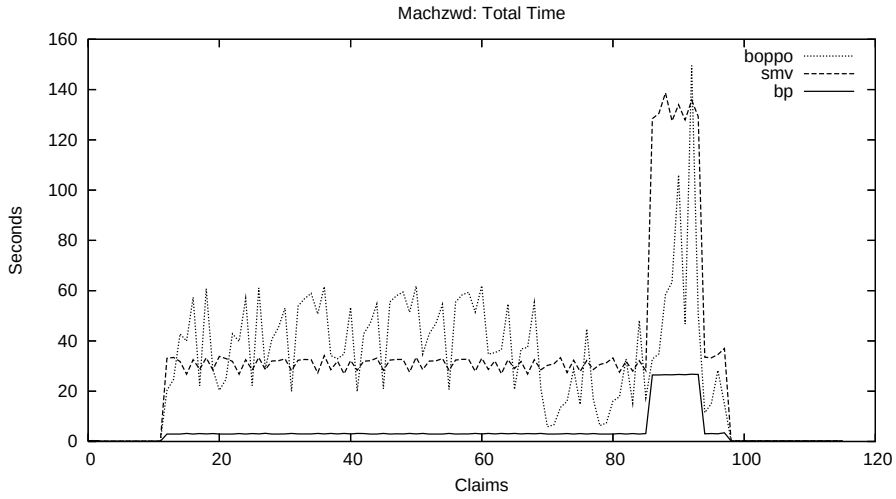
Note that several processes may be waiting for the same event. Hence, no guarantee can be provided about the status of the condition when a particular process gains back the processor since other previously scheduled processes may have already changed the value.

Interrupts The interrupt mechanism is fundamental to communicate with the devices. By means of an interrupt a device can notify the processor of some event. Interrupts transfer the control from the current process to an interrupt handler, and therefore may introduce concurrent accesses to resources. *DDVerify* supports registering and unregistering handlers for interrupts.

Interrupt handlers run outside of a process context, and thus, restrictions exists in the kind of operations that can be performed. First, interrupt handlers cannot go sleeping, or perform any action that would result in a rescheduling. Second, they cannot access user-space memory. Therefore, they are usually restricted to small important tasks such as saving new data. The more complex tasks are performed by deferred functions. This way of splitting the work is known as the *top-* and *bottom- half* interrupt handlers. The top-half handler is the code that actually responds to the interrupt and schedules the bottom-half function, which is responsible for completing the request.

Verification Conditions *DDVerify* introduces verification conditions for the device driver by means of assertions in the operational model. For instance, we check whether the mutual exclusion mechanisms of the kernel are used correctly, whether kernel objects are initialized before they are used, and whether kernel service functions are called in the correct context. In addition to the verification conditions resulting from assertions, SATABS is able to generate claims for more general properties like buffer overflows, division by zero, and invalid pointers.

Figure A.1 Sequential model: total time to prove the claims using BOPPO, SMV and BOOM.



A.4 Experiments

DDVerify introduces verification conditions for device drivers by means of assertions in the operational model of the operating system. Due to inlining, each assertion occurring in the code results in at least one claim that has to be discharged in order to show the correctness of the program.

Our framework contains a collection of drivers that come with the Linux kernel distribution. Using *DDVerify*, we found two previously unknown bugs in two device drivers. Both were discovered using a sequential driver harness.

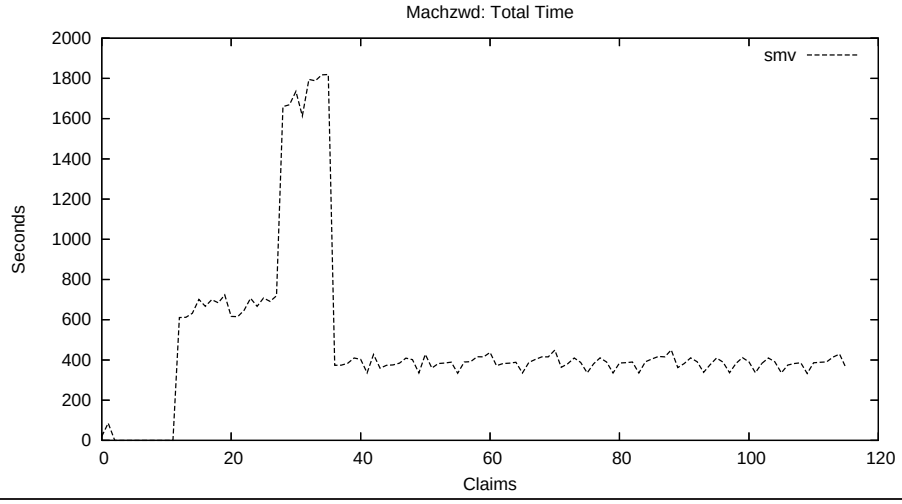
As an illustrating example, we present the verification results for a watchdog driver. The experiments were conducted on an Intel Xeon processor running at 3GHz with 4 GB of RAM.

The *machwzd* benchmark code contains 494 lines of code and uses spinlocks, IO ports and timer functions. We checked the driver for the correct usage of IO ports, which are resources that need to be requested by the driver in order to prevent access conflicts. Drivers can call the *request_region* function to acquire a range of consecutive port numbers. Using the following assertion, we verify that any access to the port `port` is valid:

```
assert(
  port >= ioport_request_start &&
  port < ioport_request_len + ioport_request_start)
```

In total, SATABS generates 116 claims for our sequential model. SATABS reports one previously unnoticed bug. This bug has gone undetected so far despite the fact that the faulty code is executed each time the module is loaded. First, the driver attempts to detect the presence of its related hardware logic, and only subsequently requests IO resources. This initialization step is faulty since the de-

Figure A.2 Concurrent model: total time to prove the claims using SMV.



tection is performed by reading a port not yet requested. As a result, the IO access has unpredictable consequences.

SATABS produces a counterexample using two predicates derived from the condition of the assertion. Fig. A.1 reports the total runtime of SATABS for each of the 116 claims generated for *machzwd*. Some of the claims are proved correct using constant propagation only. In that case, the CEGAR loop is skipped.

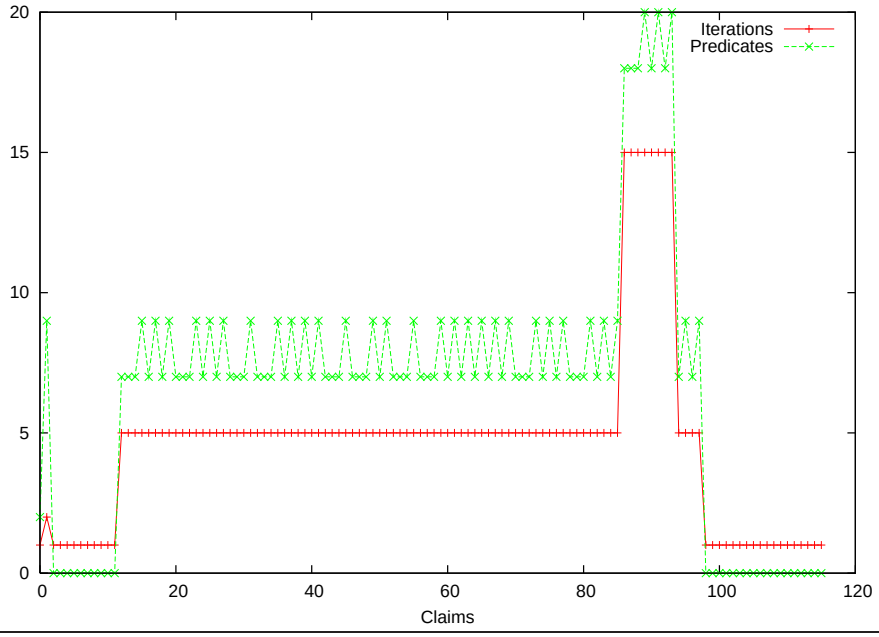
As most of the verification time is spent verifying the abstract program, the choice of the model checker has a significant impact on the runtime. We demonstrate this by using three different model checkers: Cadence SMV [K.L. McMillan, 1992], BOPPO [Cook et al., 2005], and BOOM [Basler et al., 2007]. The latter is based on efficient satisfiability checking techniques and has been presented only recently. It performs significantly better than Cadence SMV and BOPPO, but lacks support for concurrency. The number of abstraction refinement iterations, and the number predicates generated to prove the claims shown in Fig. A.1 are presented in Fig. A.3. The number of predicates generated in the refinement step depends on the counterexample returned by the model checker. Therefore, different numbers of predicates and iterations may be observed depending on the model checkers. However, this effect do not occur in this benchmark, and thus, Fig. A.3 is valid for all three model checkers.

Claim 0 corresponds to the invalid IO port access described above and turns out to be wrong. Since the broken assertion occurs at the very beginning of each execution trace of the driver, other assertions are unreachable. As a result, the verification process terminates as soon as SATABS is able to show that the assertion that corresponds to claim 0 cannot be bypassed. To verify certain claims, SATABS performs up to 15 refinement iterations, discovering 20 predicates. We observed no difference in the number of discovered predicates and iterations when switching between SMV, BOPPO, and BOOM.

We are able to verify the same properties using a concurrent model with two

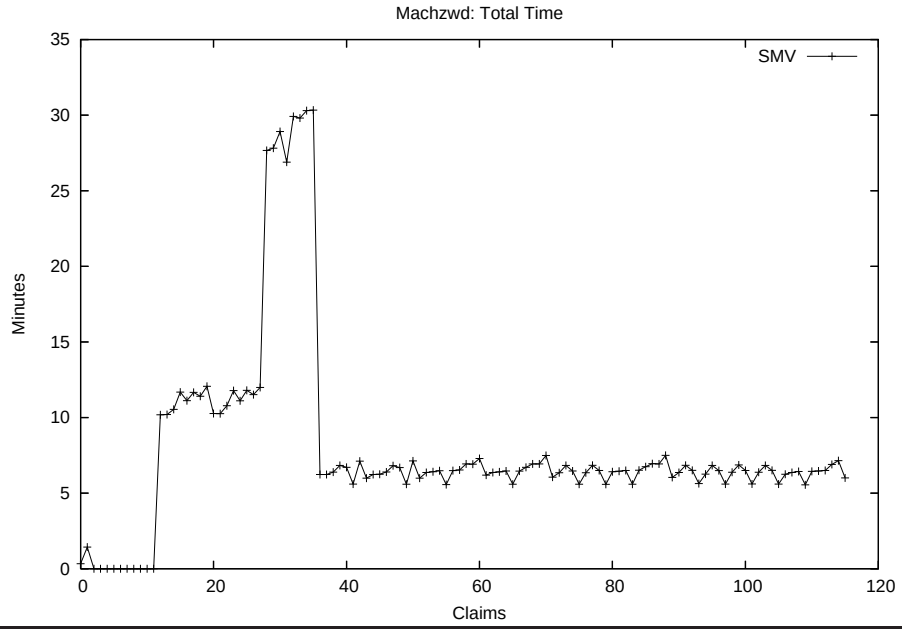
threads. The first thread calls the driver functions, while the second one simulates the arrival of interrupts and the execution of deferred tasks. The results we obtain by using SMV are presented in Fig. A.2 (BOPPO actually timed out on almost all claims). Note that the order for proving the claims depends on the model, and thus, differs from Fig. A.1. However, Fig. A.1 and Fig. A.2 share some similarities, such as a unique peak. As the number of iterations and predicates increases only slightly, we omit their presentation. SATABS needs at most 2.5 minutes to verify a claim using the sequential harness, while this number increases to 30 minutes when switching to the concurrent one. These results illustrate the state-space explosion induced by the additional thread. When the property being checked requires more predicates, the verification time increases rapidly. In particular, fixing the bug mentioned above results in a significant increase in the number of predicates required to verify a claim using the sequential model: The number of predicates increases by a factor of 10. In that case, running SATABS using the concurrent model yields time-outs for most of the claims.

Figure A.3 Sequential model: Iterations and predicates per claim.



nbd As a second benchmark, we discuss the results obtained by verifying the device driver *nbd*, which enables the use of block devices over the network. For a total of 805 lines of code, SATABS generates 62 claims related to spinlocks. We do not list trivial claims that are verified in one iteration. All claims are shown to be correct. Table A.5 reports the verification results using SMV and *Bp*. The topmost table shows the average time spent for abstraction (Abst.), model-checking (Mc.), feasibility checking (Sim.), and refinement (Ref.). It also reports the average of iterations and predicates per claim. The table in the middle presents the standard

Figure A.4 Concurrent model: total time to prove the claims using SMV.



deviation. Finally, the last table reports the maximum value observed in each column.

As previously observed, *Bp* performs better than Cadence SMV. In average, SATABS using SMV runs for 43 minutes to verify a claim, as opposed to 8 minutes only for *Bp*. Furthermore, note that the standard deviation is low, meaning that the data are distributed close to the mean.

A.5 Bibliographic Notes

Predicate abstraction [Graf and Saïdi, 1997] is an abstraction mechanism that generates finite-state models for programs with unbounded state space. The states of the abstract model are determined by evaluating the concrete states under a finite set of first order logic predicates, which reflect properties of the original program. An insufficient set of predicates leads to falsely reported counterexamples. Such inaccurate abstractions can be refined by means of adding predicates that are extracted from these false counterexamples. This technique is known as counterexample-guided abstraction refinement [Clarke et al., 2000].

Several predicate-abstraction based CEGAR verification tools are available. BLAST [Henzinger et al., 2002] improves the approach implemented in SLAM by using a "lightweight" refinement abstraction loop that refines only relevant parts of the abstract model [Henzinger et al., 2002]. Henzinger et al. present a BLAST-based automatic race checker for multithreaded C programs [Henzinger et al., 2003b], which examines each thread separately using assume-guarantee reasoning. MAGIC is able to handle concurrent programs that communicate via mes-

Figure A.5 Summary of the results obtained for Benchmark *nbd*.

	Avg. Time (m)						
	Tot.	Abst.	Mc.	Sim.	Refi.	Avg. Iter.	Avg. Pred.
Bp	22.73	6.16	6.97	6.53	3.07	28.08	102.36
SMV	63.36	8.04	43.19	8.28	3.84	34	102.36

	Stdev. Time (m)						
	Tot.	Abst.	Mc.	Sim.	Refi.	Stdev. Iter.	Stdev. Pred.
Bp	1.99	0.57	1.66	0.47	0.26	1.58	0.78
SMV	1.88	0.08	1.79	0.1	0.01	0	0.78

	Max. Time (m)						
	Tot.	Abst.	Mc.	Sim.	Refi.	Max. Iter.	Max. Pred.
Bp	26.42	7.2	10.1	7.5	3.69	31	104
SMV	67.49	8.24	47.06	8.91	3.89	34	104

sage passing, but do not use shared memory [Chaki et al., 2004]. Therefore, this approach is inappropriate for verifying concurrent Linux device drivers. SATABS uses efficient satisfiability checking algorithms to generate the abstract model and to simulate the counterexamples. Unlike the other CEGAR tools mentioned in this section, SATABS [Clarke et al., 2005] uses a bit-level accurate decision procedure, and therefore, can detect errors related to bit-level operators (e.g., overflows). Using the model checker BOPPO [Cook et al., 2005], SATABS is able to check concurrent programs that communicate via shared memory. YASM uses a modified abstraction mechanism that enables verification of liveness properties [Gurfinkel et al., 2006], while all other tools mentioned here support only safety properties. Qadeer's KISS tool uses SLAM to check concurrent programs by generating a sequential model that reflects a subset of the execution traces of the original program [Qadeer and Wu, 2004].

Besides predicate abstraction, several other model checking techniques to verify ANSI-C programs have been proposed. (We do not discuss verification tools for other languages.) The tools described below operate on the original program, i.e., no abstraction is applied. CBMC [Kroening et al., 2004] is based on *bounded model checking* (BMC), and symbolically executes all traces of a program up to a user-specified length. Due to its bounded nature, the approach is appropriate for detecting shallow bugs only. SATURN [Xie and Aiken, 2005] performs BMC for each function of the C program separately. Loops are modeled by unrolling them a predetermined number of times. Function calls are handled by maintaining concise summaries of functions.

Post and Küchlin [Post and Küchlin, 2006] present a heuristic for automatic generation of test harnesses for the BMC-based verification of Linux device drivers. Their method automatically infers parameters for the dispatch function calls, in-

cluding pointers and data structures that typically occur in device drivers.

Other verification tools such as Daikon [Ernst et al., 1999], Eraser [Savage et al., 1997], and *Verisoft* [Godefroid, 2005] are based on explicit execution rather than static analysis. [Musuvathi et al., 2002] verify file system implementations using CMC, which is a tool that explicitly executes the unmodified kernel code *within* a model checker. CMC prevents the repeated examination of the same part of the state space by storing states that have already been visited. However, explicit state enumeration is more even more susceptible to the state-explosion problem than model checkers based on symbolic techniques. The *Verisoft* tool follows a similar approach to verify concurrent C programs. *Verisoft* performs dynamic execution of C code, but does not keep track of visited states (this technique is called *state-less* search) [Godefroid, 2005]. In order to prevent the search algorithm from getting caught in infinite cycles, the depth of the execution traces is limited. Therefore, *Verisoft* is incomplete and unable to establish *correctness* results.

A.6 Summary

We present the *DDVerify* framework, which enables the automated verification of Linux device drivers based on the SATABS model checker. Given a device driver, *DDVerify* is able to automatically generate a test harness for this driver. *DDVerify* provides a sequential as well as a concurrent model of the driver service routines of the Linux kernel. These models are significantly more accurate than the operating system models provided by SLAM or BLAST. For instance, *DDVerify* supports synchronization constructs, interrupts, and deferred tasks. Furthermore, we integrate a concurrent model checker for abstract models generated by predicate abstraction into a CEGAR framework.

We present benchmarks that confirm that predicate abstraction is an adequate method for verifying sequential device drivers. In principle, it is possible to detect more bugs using a concurrent harness. However, in order to apply this verification technique to concurrent device drivers, the performance of the model checker tools for concurrent abstract programs has to be improved significantly. Currently, no CEGAR tool we are aware of is able to handle device drivers of a realistic size in the presence of threads.

DDVerify and a collection of test cases are available from `http://www.cprover.org/ddverify`. It comes with a collection of test cases, also including the one presented in this document. With the framework and the operating system models being in place, we believe that *DDVerify* is a first step to make the automated verification of Linux device drivers practical.

B

Synthesis of C/C++ test-benches for Formal Verification

THE development of a new product generally starts with a high-level software representation of the system often written in C or C++. The details of the model are then gradually refined to obtain a concrete description suitable for hardware synthesis. During the refinement iterations, engineers invest time to create simulation scenarios to validate the behaviors of their model. Simulation and validation account for most of the time of the development cycle. The ability to re-use modules and test benches from higher levels of description accelerates the development and allows to verify functional consistency between the implementation and the original high-level specification.

Hardware-manufacturing companies employ model checkers for validation of RTL designs. To use these tools, engineers must specify the correct behaviors of their model in term of system invariants, which is a task that requires expertise in formal methods. The development of techniques to automate the extraction of formal properties from test benches is therefore important to shorten time spent for validation and to facilitate adoption of formal methods.

We present a technique to encode test benches written in C/C++ into a set of System Verilog properties suitable for formal verification. Our approach fits well into existing verification flows as the properties that we generate are verified using existing model-checking tools.

Technically, the design and its test bench are synchronized at *timeframes*. A timeframe is a snapshot of the state of system at a specific point in time. The test bench drives the execution of the design by adding constraints to the snapshots. We encode the program into a propositional formula using a Bounded Model Checking technique. We rewrite then the formula as System Verilog properties that constrain the execution of circuit. The complete system can be subsequently verified using verification tools with support for the *System Verilog Assertion* language.

B.1 The Running Example

We support the discussion using a concrete verification example for an arbiter. Arbiters provide mechanisms to order conflicting accesses to shared resources in concurrent environments. They are crucial components of communication systems. In a single chip, arbiters control which devices can write to the buses. In computer networks, switches rely on fast arbitration schemes when multiple packets form different input ports are transmitted to the same output port. Designing high-performance arbiters with fairness guarantees is error prone. Typically, the complete verification of an arbiter using testing approaches is tedious due to the existence of numerous corner cases. In contrast, formal-analysis tools perform well on control-intensive tasks with little data computation such as arbiters.

As a general rule in our approach, the VERILOG design must be synthesizeable and contain a fastest clock. Schema B.1 shows the interface of an arbiter with 4 input lines for the requests (signal req) and one output port for the grant signal, which indicates the number of the selected device. We keep the example simple to facilitate the discussion. The arbiter can manage up to four parallel requests. In this example, we assume that the arbiter implements a fair round-robin scheduling policy and therefore, that any request is granted within 4 cycles.

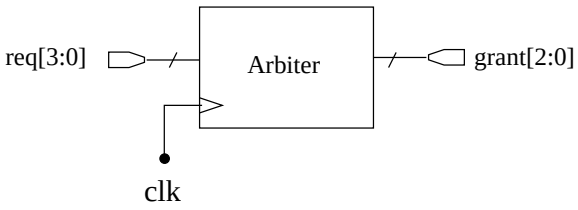


Figure B.1: arbiter

Program B.1 shows a C++ test bench to verify the fairness of the arbiter. The test bench can access the state of the circuit through variable $arbiter$ declared on line 4. The declaration qualifies $arbiter$ with the $volatile$ and $const$ modifiers to indicate to the compiler that only the environment can modify the variable. In this case, variable $arbiter$ is implicitly updated each time that the circuit makes a transition.

The execution of the test bench and the circuit are synchronized at timeframes, which correspond the clock of the system. On line 6, the test bench declares variable $simtime$ to keep track of the simulation time.

The circuit switches to the next timeframe each time that the test bench increments the simulation time. Our programming model excludes decrementing timeframes.

By default the inputs of the circuit are free. The test bench directs the execution using assume statements. Similarly, it verifies properties using C++ assert statements. The programmer can specify arbitrary C++ boolean expressions.

Program B.1 C++ test bench for verifying the fairness of the arbiter presented in Schema B.1.

```
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
```

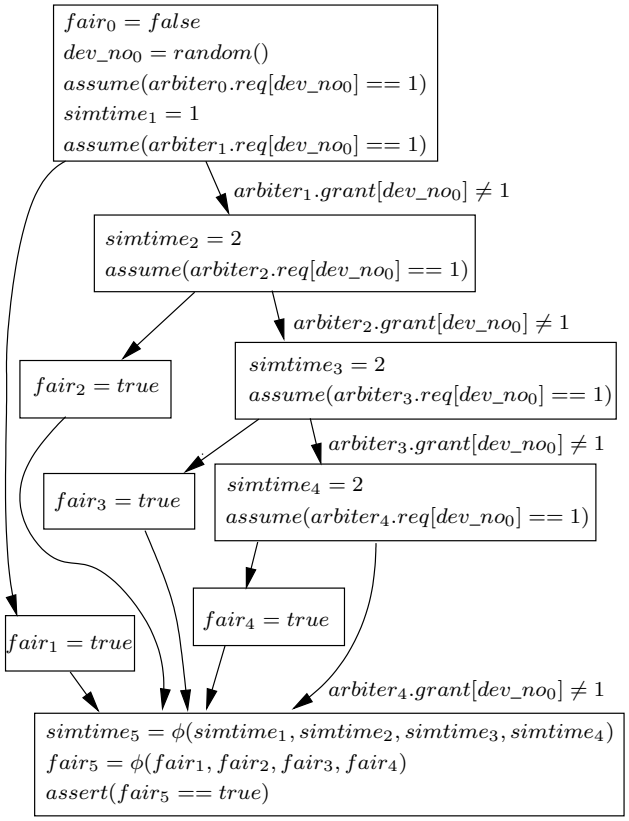
```
const volatile
struct {sc_uint<4> req;
  sc_uint<2> grant;} arbiter;

unsigned simtime;

int main() {
  bool fair = false;
  sc_uint<2> dev_no = random();

  assume(arbiter.req[dev_no] == 1);

  for(int i = 0; i < 4; i++) {
    simtime += 1;

    assume(arbiter.req[dev_no] == 1);

    if(arbiter.grant == dev_no) {
      fair = true;
      break;
    }
  }
  assert(fair == true);
}
```

In this example, the test bench randomly selects an input line (variable dev_no) and generates a request using the assume statement on line 12. According to the round-robin arbitration schema, a correct arbiter shall fulfill the request within 4 cycles. The assertion on line 24 enforces this property. Note that in this model, the inputs are set implicitly and randomly in each timeframe. Therefore, the execution can interleave requests arbitrarily.

B.2 Extraction of Symbolic Constraints

Bounded Model Checking [Biere et al., 2003, Kroening et al., 2004] denotes a symbolic verification technique that encodes a finite computation and a property into a propositional formula that is satisfiable if and only if there exists an trace that violates the property. In that case, every satisfiable assignment provides an example of incorrect execution. In general, modelcheckers convert transition systems into CNF formulae thanks to the growing scalability of SAT-solvers.

Figure B.1 Cotrol-Flow Graph of Program B.1



Bounded Model Checking has been applied to both hardware and software systems in [Kroening et al., 2003]. In contrast to this work, our technique allows unbounded verification of the hardware.

The first step to extract the formula consists of building a control-flow graph representation of the program and to unroll it. Subsequently, we translate the code into a Single Static Assignment form (SSA) [Cytron et al., 1991] meaning that every variable is assigned exactly in one location. This location is called a variable definition. We obtain this form by adding a version number to the variables so that each variable definition is uniquely identified. Consequently, several versions of a same variable may exists along different paths. At the locations where the control flow merges we introduce ϕ nodes to select the correct values of the variables.

Figure B.1 shows the control-flow graph of Program B.1 after unrolling and conversion to SSA form. As variable $arbiter$ must reflect the state of the circuit in each time frame, we increments its version number implicitly each time that variable $simtime$ is redefined.

Finally, we extract the symbolic constraints directly from the SSA representation. Figure B.2 shows these constraints. The statements are mapped directly the clauses of the formula. We implement ϕ nodes with a case split over the possible incoming edges (see constraints for $simtime_5$ and $fair_5$).

Figure B.2 Equation of Program B.1

$$
\begin{aligned}
& fair_0 = 0 \ \wedge \ arbiter_0.req[dev\_no_0] = 1 \\
& simtime_1 = 1 \ \wedge \ arbiter_1.req[dev\_no_0] = 1 \\
\\
G_1 \ &\Rightarrow \ fair_1 = 1 \\
G_2 \ &\Rightarrow \ simtime_2 = 2 \ \wedge \ arbiter_2.req[dev\_no_0] = 1 \\
G_3 \ &\Rightarrow \ fair_2 = 1 \wedge \\
G_4 \ &\Rightarrow \ simtime_3 = 3 \ \wedge \ arbiter_3.req[dev\_no_0] = 1 \\
G_5 \ &\Rightarrow \ fair_3 = 1 \\
G_6 \ &\Rightarrow \ simtime_4 = 4 \ \wedge \ arbiter_4.req[dev\_no_0] = 1 \\
G_7 \ &\Rightarrow \ fair_4 = 1 \\
\\
G_1 \ &\Rightarrow \ simtime_5 = simtime_1 \ \wedge \ fair_5 = fair_1 \\
G_3 \ &\Rightarrow \ simtime_5 = simtime_2 \ \wedge \ fair_5 = fair_2 \\
G_5 \ &\Rightarrow \ simtime_5 = simtime_3 \ \wedge \ fair_5 = fair_3 \\
G_7 \ &\Rightarrow \ simtime_5 = simtime_4 \ \wedge \ fair_5 = fair_4 \\
G_8 \ &\Rightarrow \ simtime_5 = simtime_4 \ \wedge \ fair_5 = fair_0 \\
\\
G_1 \ &\Leftrightarrow \ arbiter_1.grant[dev\_no_0] = 1 \\
G_2 \ &\Leftrightarrow \ arbiter_1.grant[dev\_no_0] = 0 \\
G_3 \ &\Leftrightarrow \ G_2 \ \wedge \ arbiter_2.grant[dev\_no_0] = 1 \\
G_4 \ &\Leftrightarrow \ G_2 \ \wedge \ arbiter_2.grant[dev\_no_0] = 0 \\
G_5 \ &\Leftrightarrow \ G_4 \ \wedge \ arbiter_3.grant[dev\_no_0] = 1 \\
G_6 \ &\Leftrightarrow \ G_4 \ \wedge \ arbiter_3.grant[dev\_no_0] = 0 \\
G_7 \ &\Leftrightarrow \ G_6 \ \wedge \ arbiter_4.grant[dev\_no_0] = 1 \\
G_8 \ &\Leftrightarrow \ G_6 \ \wedge \ arbiter_4.grant[dev\_no_0] = 0
\end{aligned}
$$

B.3 System Verilog Harness

SYSTEM VERILOG is a system-level modeling language implemented as an extension of VERILOG. It permits describing a system at several levels of abstraction, starting at a high-level functional description, down to synthesizable gate-level. Along with support for object-oriented software programming, SYSTEM VERILOG introduces a powerful assertion language that can express complex temporal behaviors. In addition to assertions, SYSTEM VERILOG also provides assume statements to restrict the execution to traces that satisfy certain properties.

Technically, SYSTEM VERILOG distinguishes two kind of assertions. *Immediate assertions* are statements inside processes that are evaluated when the execution reaches their location. In contrast, *concurrent assertions* are evaluated on a clock basis when the state of the circuit is stabilized. The same distinction applies for assume statements.

Internally, formal verification tools build a netlist representation of the tran-

sition system. The design and its properties must thus be synthesizeable. In general, synthesis tools are unable to handle test-bench constructs such as clock delays and immediate assertions. Consequently, test benches cannot be passed as is to formal verification engines.

Except for C++ assertions, which are converted into SYSTEM VERILOG concurrent assertions, we translate C++ programs in SYSTEM VERILOG concurrent assume statements that constrain the execution of the circuit. In this way, we circumvent the synthesis restrictions imposed on VERILOG modules.

Figure B.3 shows the schema of the SYSTEM VERILOG harness. We encode the variables of the test bench as flip flops that hold their initial value forever. The registers are set once at the beginning of the execution and keep their value constant. A model checker enforces the semantics of the test bench by solving SYSTEM VERILOG equations that constrain the values of the variables of the test bench.

As example, the following SYSTEM VERILOG code shows the declaration of variable $arbiter_0$ and dev_no_0 and their related constraints:

```
reg struct {...} arbiter0;
reg [31,0] dev_no0;

always @(posedge clk) arbiter0 <= arbiter0;
always @(posedge clk) dev_no0 <= dev_no0;

assume property (@posedge clk)
 arbiter0.req[dev_no0] ==1 ;
```

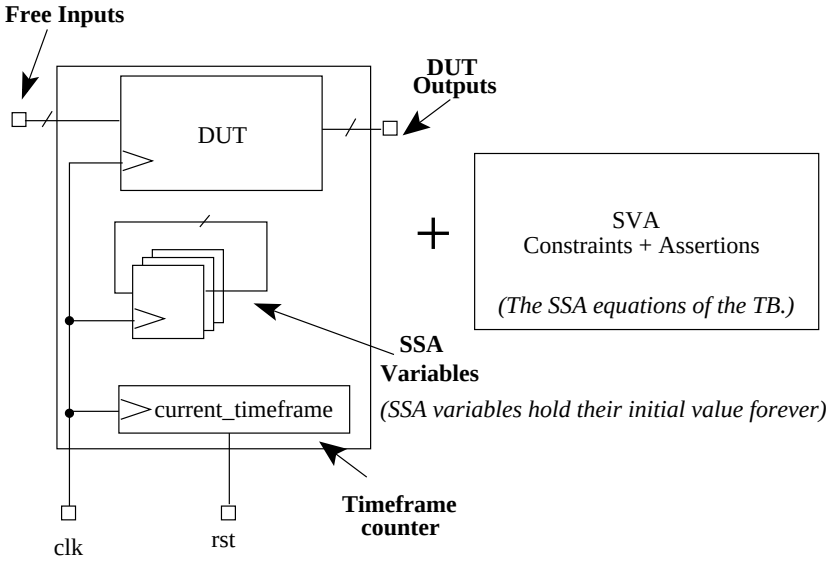
At this point, the execution of the test bench and the circuit are independent. We introduce additional synchronization constraints between the variables of the test bench and the registers of the design. The following code snippet shows the synchronization constraint for timeframe 4:

```
assume property (@posedge clk)
 (current_timeframe == sim_time4 && G6) |->
   DUT.req == arbiter4.req     &&
   DUT.grant == arbiter4.grant;
```

As synchronization happens at timeframes, we introduces a timeframe counter $current\_timeframe$ to keep track of the current time for the circuit. Unlike the registers of the test bench, which hold constant values, $current\_timeframe$ is incremented with the clock. Therefore, the previous property states that the state of the DUT must match the state of $arbiter_4$ in timeframe 4 if condition G_6, shown in Figure B.2, holds.

Similarly, the evaluation of assertions must be synchronized with the execution of the circuit. Thus, we translate the C++ assertion in Figure B.1 into the following SYSTEM VERILOG property:

Figure B.3 Schema of the SYSTEM VERILOG harness




```
assert property (@posedge clk)
 current_timeframe == sim_time5 |->
   fair5 == 1;
```

Any counterexample that violates this property provides a trace in both the original C++ test bench and the verilog circuit.

B.4 Bibliographic Notes

Hardware/Software co-verification techniques [Séméria et al., 2002, Rowson, 1994] allow to discover bugs early in the development process before production, at which time changes in the design are prohibitive both in term of delays and costs. Methods for Hardware/Software co-verification are based on simulation.

Test benches try to capture the intent of a design and test scenarios at which the designer thinks. Traditionally, simulation-based verification techniques belong to three different categories:

1. Directed-random verification denote testing approaches that use concrete values to drive the execution of the circuit. Such a test describes a single execution path. In order to build faith in the correctness of the model, the scenario must cover a significant part of the functionalities of the design.

2. In random testing the simulator chooses the values of the inputs randomly. Simulation must last for long periods to ensure that a significant part of the behaviors are covered. In particular, random testing can miss important corner cases.

3. Finally, directed-random verification denotes techniques that adjust the degree of freedom of the inputs using constraints. Languages such as *Vera*, *'e'*, and *System Verilog* allow to specify the domain of the inputs using symbolic equations. Simulators with support for constrained-random verification techniques can then solve the equations and generate random stimuli that extensively test the scenario [Behm et al., 2004, Haque and Michelson, 2001].

Engineers can estimate the percentage of behaviors that are covered during simulation using coverage heuristics. Several metrics for coverage exists, e.g, Tasiran and Keutzer [2001], Devadas et al. [1996]. *Code coverage* heuristics assume that the word-level representation of the design is available and quantify how much of the code has been visited during simulation. *Circuit coverage* describes a second family of heuristics based on the quantification of the activity of the elements in the circuits such as registers and outputs. Testing techniques can achieve high coverage with respect to these heuristics, typically more than 90 percents [Bartley et al., 2002], but may miss subtle corner cases.

HW-CBMC [Kroening et al., 2003] is a tool for checking behavioral consistency of C/C++ programs and Verilog using Bounded Model Checking. The test bench and the design are unrolled in tandem and translated into a boolean formula that is satisfiable if and only if there exists a mismatch between the test bench and the circuit. In that case, the satisfiable assignment provides a counter-example in the program and the design that exposes the bad behavior. Such approach provides formal safety guarantees for all execution traces up to a certain bound.

HW-CBMC maps the state of the circuit into the memory space of the test bench using arrays. By convention, the test bench must declare an external array with the same name as the top module. The i^{th} element of this array represents the state of the module at time i. The size of the array must match the number of time the circuit is unrolled.

Instead of encoding the test bench into a SAT formula, our technique translates the program into SYSTEM VERILOG properties and builds a harness for the circuit. The design under test and its harness are then verified with a separate tool. We see two important reasons for adopting a decoupled approach. First, our technique avoids to unwind the circuit upfront. In particular, the verification engine can choose different verification strategies other than bounded model checking. Second, the approach is more easy to integrate in already existing design flows as modelcheckers can be used unmodified.

We have presented a novel technique to encode a test bench written in C++ into a set of SYSTEM VERILOG properties that are suitable for formal verification. A similar approach can apply to any testing language with support for constrained-random verification. While in general pure simulation constructs such as delays and immediate assertions cannot be analysed with current formal verification tools, our technique enables to combine the verification of hardware and software using existing model checkers for SYSTEM VERILOG.

Bibliography

Tony Andrews, Shaz Qadeer, Sriram K. Rajamani, Jakob Rehof, and Yichen Xie. Zing: Exploiting program structure for model checking concurrent software. In *Proceedings of CONCUR*, pages 1–15, 2004.

T. Ball and S.K. Rajamani. Boolean programs: A model and process for software analysis. Technical Report MSR-TR-2000-14, Microsoft Research, 2000a.

Thomas Ball and Sriram K. Rajamani. BEBOP: A symbolic model checker for Boolean programs. In *SPIN Workshop on Model Checking of Software*, volume 1885 of *LNCS*. Springer, 2000b.

Thomas Ball and Sriram K. Rajamani. The SLAM project: debugging system software via static analysis. In *POPL '02: Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 1–3, New York, NY, USA, 2002. ACM. ISBN 1-58113-450-9.

Thomas Ball, Byron Cook, Vladimir Levin, and Sriram K. Rajamani. SLAM and Static Driver Verifier: Technology transfer of formal methods inside Microsoft. In *IFM*, 2004.

Thomas Ball, Ella Bounimova, Byron Cook, Vladimir Levin, Jakob Lichtenberg, Con McGarvey, Bohus Ondrusek, Sriram K. Rajamani, and Abdullah Ustuner. Thorough static analysis of device drivers. In *European Systems Conference (EuroSys)*, 2006.

Mike G. Bartley, Darren Galpin, and Tim Blackmore. A comparison of three verification techniques: directed testing, pseudo-random testing and property checking. In *DAC '02: Proceedings of the 39th conference on Design automation*, pages 819–823, New York, NY, USA, 2002. ACM. ISBN 1-58113-461-4.

Gérard Basler, Daniel Kroening, and Georg Weissenbacher. SAT-based summarisation for Boolean programs. In *SPIN Workshop on Model Checking of Software*, volume 4595 of *LNCS*, 2007.

Michael Behm, John Ludden, Yossi Lichtenstein, Michal Rimon, and Michael Vinov. Industrial experience with test generation languages for processor verification. *dac*, 00:36–40, 2004.

David Berner and Jean-Pierre Talpin. SystemCXML: An extensible SystemC front end using XML. *In proceedings of the forum on specification and design danguages (FDL).*, 2005.

Armin Biere, Alessandro Cimatti, Edmund M. Clarke, Ofer Strichman, and Yunshan Zhu. Bounded model checking. *Advances in Computers*, 58:118–149, 2003.

K Bierhoff. Iterator specification with typestates. In *SAVCBS '06: Proceedings of the 2006 conference on Specification and verification of component-based systems*, pages 79–82. ACM, 2006.

Nicolas Blanc, Daniel Kroening, and Natasha Sharygina. Scoot: A tool for the analysis of SystemC models. In *TACAS*, Lecture Notes in Computer Science, pages 467–470. Springer, 2008.

Richard Bornat. Proving pointer programs in Hoare logic. In *Mathematics of Program Construction*, pages 102–126, 2000.

Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. Comput.*, 35(8):677–691, 1986. ISSN 0018-9340.

Randal E. Bryant, Shuvendu K Lahiri, and Sanjit A. Seshia. Modeling and verifying systems using a logic of counter arithmetic with lambda expressions and uninterpreted functions. In *Computer Aided Verification (CAV)*, volume 2404 of *LNCS*. Springer, 2002.

Jerry R. Burch, Edmund M. Clarke, Kenneth L. McMillan, David L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. In *Proceedings of LICS*, pages 428–439. IEEE Computer Society, 1990.

S. Chaki, E. Clarke, A. Groce, S. Jha, and H. Veith. Modular verification of software components in C. *IEEE Trans. on Software Engineering*, 30(6):388–402, June 2004.

E. Clarke, D. Kroening, N. Sharygina, and K. Yorav. Predicate abstraction of ANSI–C programs using SAT. *Formal Methods in System Design*, 25:105–127, September–November 2004.

Edmund Clarke, Daniel Kroening, Natasha Sharygina, and Karen Yorav. SATABS: SAT-based predicate abstraction for ANSI-C. In *TACAS*, volume 3440 of *Lecture Notes in Computer Science*. Springer, 2005. ISBN 3-540-25333-5.

Edmund Clarke, Himanshu Jain, and Daniel Kroening. Verification of SpecC using predicate abstraction. *Form. Methods Syst. Des.*, 30(1):5–28, 2007. ISSN 0925-9856.

Edmund M. Clarke, E. Allen Emerson, and A. Prasad Sistla. Automatic verification of finite state concurrent systems using temporal logic specifications: A practical approach. In *POPL*, pages 117–126, 1983.

Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In *CAV '00: Proceedings of the 12th International Conference on Computer Aided Verification*, Lecture Notes in Computer Science, pages 154–169, London, UK, 2000. Springer. ISBN 3-540-67770-4.

D. R. Cok. Specifying Java iterators with JML and Esc/Java2. In *Specification and verification of component-based systems*, pages 71–74, 2006.

Byron Cook, Daniel Kröning, and Natasha Sharygina. Symbolic model checking for asynchronous Boolean programs. In *SPIN Workshop on Model Checking of Software*, volume 3639 of *LNCS*. Springer, 2005.

Byron Cook, Andreas Podelski, and Andrey Rybalchenko. Terminator: Beyond safety. In *CAV*, pages 415–418, 2006.

Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, pages 238–252, 1977.

Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.*, 13(4):451–490, 1991. ISSN 0164-0925.

Srinivas Devadas, Abhijit Ghosh, and Kurt Keutzer. An observability-based code coverage metric for functional simulation. *iccad*, 00:418, 1996. ISSN 1092-3152.

Vijay D'Silva, Daniel Kroening, and Georg Weissenbacher. A survey of automated techniques for formal software verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 27(7):1165–1178, July 2008.

Jr. Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model checking*. MIT Press, Cambridge, MA, USA, 1999. ISBN 0-262-03270-8.

N. Eén and Niklas Sörensson. An extensible SAT-solver. In *Theory and Applications of Satisfiability Testing (SAT)*, pages 502–518, 2003.

E. Allen Emerson and Richard J. Trefler. From asymmetry to full symmetry: New techniques for symmetry reduction in model checking. In *CHARME*, pages 142–156, 1999.

Dawson Engler and Ken Ashcraft. RacerX: Effective, static detection of race conditions and deadlocks. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 237–252, New York, NY, USA, 2003. ACM. ISBN 1-58113-757-5.

M. Ernst, J. Cockrell, W. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. In *International Conference on Software Engineering*, pages 213–224, 1999.

Cormac Flanagan and Stephen N. Freund. Type-based race detection for Java. In *PLDI '00: Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, pages 219–232, New York, NY, USA, 2000. ACM. ISBN 1-58113-199-2.

Cormac Flanagan and Patrice Godefroid. Dynamic partial-order reduction for model checking software. In *POPL '05: Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 110–121, New York, NY, USA, 2005. ACM. ISBN 1-58113-830-X.

R. W. Floyd. Assigning meaning to programs. In J. T. Schwartz, editor, *Mathematical aspects of computer science: Proc. American Mathematics Soc. symposia*, volume 19, pages 19–31, Providence RI, 1967. American Mathematical Society.

Masahiro Fujita and Hiroshi Nakamura. The standard SpecC language. In *ISSS '01: Proceedings of the 14th international symposium on Systems synthesis*, pages 81–86, New York, NY, USA, 2001. ACM. ISBN 1-58113-418-5.

Erich Gamma, Richard Helm, Ralph E. Johnson, and John M. Vlissides. Design patterns: Abstraction and reuse of object-oriented design. In *ECOOP '93: Proceedings of the 7th European Conference on Object-Oriented Programming*, pages 406–431, London, UK, 1993. Springer-Verlag. ISBN 3-540-57120-5.

Patrice Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State-Explosion Problem*. Lecture Notes in Computer Science. Springer, Secaucus, NJ, USA, 1996. ISBN 3540607617.

Patrice Godefroid. Software model checking: The VeriSoft approach. *Form. Methods Syst. Des.*, 26(2):77–101, 2005. ISSN 0925-9856.

Susanne Graf and Hassen Saïdi. Construction of abstract state graphs with PVS. In *CAV '97: Proceedings of the 9th International Conference on Computer Aided Verification*, Lecture Notes in Computer Science, pages 72–83, London, UK, 1997. Springer. ISBN 3-540-63166-6.

D. Gregor and S. Schupp. STLlint: lifting static checking from languages to libraries. *Softw. Pract. Exper.*, 36(3):225–254, 2006. ISSN 0038-0644.

D. Gregor and S. Schupp. Making the usage of STL safe. In *IFIP TC2/WG2.1 Working Conference on Generic Programming*, pages 127–140, 2003.

Arie Gurfinkel, Ou Wei, and Marsha Chechik. Yasm: A software model-checker for verification and refutation. In *Computer Aided Verification (CAV)*, volume 4144 of *LNCS*. Springer, 2006.

Faisal Haque and Jonathan Michelson. *Art of Verification with VERA*. Verification Central, 2001. ISBN 097119940X.

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. ISBN 0-7695-2707-8.

T. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Software verification with Blast. In *Proceedings of the Tenth International Workshop on Model Checking of Software (SPIN)*, 2003a.

Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Gregoire Sutre. Lazy abstraction. In *Principles of Programming Languages (POPL)*. ACM Press, 2002.

Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Shaz Qadeer. Thread-modular abstraction refinement. In *Computer Aided Verification (CAV)*, volume 2725 of *LNCS*. Springer, 2003b.

C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969. ISSN 0001-0782.

C. A. R. Hoare and N. Wirth. An axiomatic definition of the programming language PASCAL. *Acta Informatica*, 2, 1973.

Samin S. Ishtiaq and Peter W. O'Hearn. BI as an assertion language for mutable data structures. In *Principles of Programming Languages*, pages 14–26, 2001.

ISO/IEC. ISO/IEC 14882:2003 (E). *Programming languages - C++*, 2003.

B. Jacobs, F Piessens, and W. Schulte. Vc generation for functional behavior and non-interference of iterators. In *Specification and verification of component-based systems*, pages 67–70, 2006.

Simon L. Peyton Jones, Cordy Hall, Kevin Hammond, Jones Cordy, Hall Kevin, Will Partain, and Phil Wadler. The Glasgow Haskell compiler: a technical overview, 1992.

James C. King. Symbolic execution and program testing. volume 19, pages 385–394, New York, NY, USA, 1976. ACM.

K.L. McMillan. The SMV system. Technical Report CMU-CS-92-131, Carnegie Mellon University, 1992.

Nikolaos Kostaras and H. T. Vergos. Syce: An integrated environment for system design in systemc. In *RSP '05: Proceedings of the 16th IEEE International Workshop on Rapid System Prototyping*, pages 258–260, Washington, DC, USA, 2005. IEEE Computer Society. ISBN 0-7695-2361-7.

N. R. Krishnaswami. Reasoning about iterators with separation logic. In *Specification and verification of component-based systems*, pages 83–86, 2006.

D. Kroening and N. Sharygina. Formal verification of SystemC by automatic hardware/software partitioning. In *MEMOCODE '05: Proceedings of the 2nd ACM/IEEE International Conference on Formal Methods and Models for Co-Design*, pages 101–110, Washington, DC, USA, 2005. IEEE Computer Society. ISBN 0-7803-9227-2.

D. Kroening, E. Clarke, and F. Lerda. A tool for checking ANSI-C programs. In *TACAS*, LNCS, pages 168–176. Springer, 2004.

Daniel Kroening and Ofer Strichman. Efficient computation of recurrence diameters. In *VMCAI 2003: Proceedings of the 4th International Conference on Verification, Model Checking, and Abstract Interpretation*, pages 298–309, London, UK, 2003. Springer-Verlag. ISBN 3-540-00348-7.

Daniel Kroening, Edmund M. Clarke, and Karen Yorav. Behavioral consistency of C and Verilog programs using bounded model checking. In *DAC*, pages 368–371, May 2003.

Sudipta Kundu, Malay Ganai, and Rajesh Gupta. Partial order reduction for scalable testing of SystemC TLM designs. In *DAC '08: Proceedings of the 45th annual conference on Design automation*, pages 936–941, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-115-6.

Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978. ISSN 0001-0782.

B. Liskov and S. Zilles. Programming with abstract data types. In *ACM SIGPLAN Symposium on very high level languages*, pages 50–59. ACM, 1974.

Kenneth L. McMillan. Applications of Craig interpolants in model checking. In *TACAS*, pages 1–12, 2005.

Kenneth L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.

Alice Miller, Alastair F. Donaldson, and Muffy Calder. Symmetry in temporal logic model checking. *ACM Comput. Surv.*, 38(3), 2006.

R. Milner. *A Calculus of Communicating Systems*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1982. ISBN 0387102353.

M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an Efficient SAT Solver. In *Proceedings of the 38th Design Automation Conference (DAC'01)*, pages 530–535, 2001.

Matthieu Moy, Florence Maraninchi, and Laurent Maillet-Contoz. Pinapa: an extraction tool for systemc descriptions of systems-on-a-chip. In *EMSOFT '05: Proceedings of the 5th ACM international conference on Embedded software*, pages 317–324, New York, NY, USA, 2005. ACM. ISBN 1-59593-091-4.

M. Musuvathi, D. Park, A. Chou, D. Engler, and D. Dill. CMC: a pragmatic approach to model checking real code. In *Symposium on Operating System Design and Implementation*, 2002.

Mayur Naik, Alex Aiken, and John Whaley. Effective static race detection for Java. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, pages 308–319, New York, NY, USA, 2006. ACM. ISBN 1-59593-320-4.

Robert H. B. Netzer and Barton P. Miller. What are race conditions? Some issues and formalizations. *ACM Lett. Program. Lang. Syst.*, 1(1):74–88, 1992. ISSN 1057-4514.

Doron Peled. All from one, one for all: On model checking using representatives. In *CAV '93: Proceedings of the 5th International Conference on Computer Aided Verification*, Lecture Notes in Computer Science, pages 409–423, London, UK, 1993. Springer. ISBN 3-540-56922-7.

Doron Peled. Combining partial order reductions with on-the-fly model-checking. In *CAV '94: Proceedings of the 6th International Conference on Computer Aided Verification*, pages 377–390, London, UK, 1994. Springer. ISBN 3-540-58179-0.

Daniel Gracia Pérez, Gilles Mouchard, and Olivier Temam. A new optimized implemention of the SystemC engine using acyclic scheduling. In *DATE*, pages 552–557. IEEE, 2004. ISBN 0-7695-2085-5-1.

Hendrik Post and Wolfgang Küchlin. Automatic data environment construction for static device drivers analysis. In *Specification and Verification of Component-based Systems (SAVCBS)*. ACM Press, 2006.

Shaz Qadeer and Dinghao Wu. KISS: keep it simple and sequential. *SIGPLAN Not.*, 39(6):14–24, 2004. ISSN 0362-1340.

G. Ramalingam. Context-sensitive synchronization-sensitive analysis is undecidable. *ACM Transactions on Programming Languages and Systems*, 22(2):416–430, 2000.

J. Reynolds. Separation logic: A logic for shared mutable data structures. In *Logic In Computer Science*, pages 55–74. IEEE, 2002.

James A. Rowson. Hardware/software co-simulation. In *DAC*, pages 439–440, New York, NY, USA, 1994. ACM. ISBN 0-89791-653-0.

J. Ruf, D. Hoffmann, J. Gerlach, T. Kropf, W. Rosenstiehl, and W. Mueller. The simulation semantics of SystemC. In *DATE '01: Proceedings of the conference on Design, automation and test in Europe*, pages 64–70, Piscataway, NJ, USA, 2001. IEEE Press. ISBN 0-7695-0993-2.

Ashraf Salem. Formal semantics of synchronous SystemC. In *DATE '03: Proceedings of the conference on Design, Automation and Test in Europe*, page 10376, Washington, DC, USA, 2003. IEEE Computer Society. ISBN 0-7695-1870-2.

Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Trans. Comput. Syst.*, 15(4):391–411, 1997. ISSN 0734-2071.

Nick Savoiu, Shukla Sandeep, and Gupta Rajesh. Improving SystemC simulation through Petri net reductions. In *MEMOCODE*, pages 131–140, 2005.

Thorsten Schubert and Wolfgang Nebel. The Quiny SystemC front end: self-synthesising designs. In *FDL*, pages 135–143. ECSI, 2006. ISBN 978-3-00-019710-9.

Stefan Schwoon. *Model-Checking Pushdown Systems*. PhD thesis, Technische Universität München, 2002.

Luc Séméria, Renu Mehra, Barry Pangrle, Arjuna Ekanayake, Andrew Seawright, and Daniel Ng. Rtl c-based methodology for designing and verifying a multi-threaded processor. In *DAC*, pages 123–128, New York, NY, USA, 2002. ACM.

Alper Sen, Vinit Ogale, and Magdy S. Abadir. Predictive runtime verification of multi-processor SoCs in SystemC. In *DAC '08: Proceedings of the 45th annual conference on Design automation*, pages 948–953, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-115-6.

Mary Sheeran, Satnam Singh, and Gunnar Stålmarck. Checking safety properties using induction and a sat-solver. In *FMCAD '00: Proceedings of the Third International Conference on Formal Methods in Computer-Aided Design*, pages 108–125, London, UK, 2000. Springer-Verlag. ISBN 3-540-41219-0.

Kenneth Slonneger and Barry Kurtz. Domain theory and fixed-point semantics. In *Formal Syntax and Semantics of Programming Languages: A Laboratory Based Approach*, pages 341–394. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995. ISBN 0201656973.

A. A. Stepanov and M. Lee. The Standard Template Library. Technical Report ANSI X3J16/94-0095, ISO WG21/N0482, 1994.

Wilson Synder. Verilator. http://www.veripool.org.

SystemC. Open systemc initiative (OSCI). http://www.systemc.org.

SystemVerilog. 1800-2005 IEEE std. for System Verilog. http://www.systemverilog.org.

Serdar Tasiran and Kurt Keutzer. Coverage metrics for functional validation of hardware designs. *IEEE Design and Test of Computers*, 18(4):36–45, 2001. ISSN 0740-7475.

Veldhuizen Todd. Using C++ template metaprograms. pages 459–473, 1996.

Moshe Y. Vardi. Formal techniques for SystemC verification. In *DAC '07: Proceedings of the 44th annual conference on Design automation*, pages 188–192, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-627-1.

Verilog. 13364-2001 IEEE std. for Verilog hardware description language.

W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda. Model checking programs. *Automated Software Engineering*, 10(2):203–232, 2003.

C. Wang and D. Musser. Dynamic verification of C++ generic algorithms. *IEEE Transactions on Software Engineering*, 23(5):314–323, 1997.

Chao Wang, Zijiang Yang, Vineet Kahlon, and Aarti Gupta. Peephole partial order reduction. In *TACAS*, Lecture Notes in Computer Science, pages 382–396. Springer, 2008.

B. W. Weide. Savcbs 2006 Challenge: Specification of iterators. In *Specification and verification of component-based systems*, pages 75–78, 2006.

Thomas Witkowski. Formal verification of Linux device drivers. Master's thesis, Dresden University of Technology, 2007.

Thomas Witkowski, Nicolas Blanc, Daniel Kroening, and Georg Weissenbacher. Model checking concurrent Linux device drivers. In *ASE '07: Proceedings of the 22nd IEEE/ACM international conference on Automated software engineering*, pages 501–504, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-882-4.

Yichen Xie and Alexander Aiken. Saturn: A SAT-based tool for bug detection. In *Computer Aided Verification (CAV)*, volume 3576 of *LNCS*. Springer, 2005.