Race Analysis for SystemC using Model Checking

NICOLAS BLANC
ETH Zurich, Switzerland
and
DANIEL KROENING
Oxford University, Computing Laboratory, UK

SystemC is a system-level modeling language that offers a wide range of features to describe concurrent systems at different levels of abstraction. The SystemC standard permits simulators to implement a deterministic scheduling policy, which often hides concurrency-related design flaws. We present a novel compiler for SystemC that integrates a very precise formal race analysis by means of Model Checking. Our compiler produces a simulator that uses the outcome of the analysis to perform partial order reduction. The key insight to make the Model Checking engine scale is to apply it only to tiny fractions of the SystemC model. We show that the outcome of the analysis is not only valuable to eliminate redundant context switches at runtime, but can also be used to diagnose race conditions statically. In particular, our analysis is able to reveal races that can remain undetected during simulation and is able to formally prove the absence of races.

Categories and Subject Descriptors: D.2.4 [**Software Engineering**]: Software/Program Verification; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs

General Terms: Algorithms, Verification

Additional Key Words and Phrases: SystemC, simulation, partial-order reduction, Model Checking, formal analysis

1. INTRODUCTION

Time-to-market requirements have pushed the Electronic Design Automation (EDA) industry towards design paradigms that permit 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 C-like system design languages has emerged. A key feature of these languages is joint modeling of both the hardware and software component of a system using a language that is well-known to engineers. A promising candidate for an industry standard is SystemC [IEEE Std 2005].

This paper is an extended version of a conference paper that appeared at ICCAD 2008 [Blanc and Kroening 2008]. This research is supported by ETH research grant TH-21/05-1, by the Semiconductor Research Corporation (SRC) under contract no. 2006-TJ-1539, the EU FP7 STREP MOGENTES (project ID ICT-216679), and by the EPSRC project EP/G026254/1.
Permission to make digital/hard copy of all or part of this material without fee for personal or classroom use provided that the copies are not made or distributed for profit or commercial advantage, the ACM copyright/server notice, the title of the publication, and its date appear, and notice is given that copying is by permission of the ACM, Inc. To copy otherwise, to republish, to post on servers, or to redistribute to lists requires prior specific permission and/or a fee.
© 2010 ACM 1084-4309/2010/0400-0001 $5.00

SystemC offers a wide range of language features such as hierarchical design, arbitrary-width bit-vector types, and concurrency with related synchronization mechanisms. SystemC permits different levels of abstraction, ranging 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.

SystemC programs make use of a C++ template library. SystemC modules are therefore plain C++ classes, which are compiled and then linked to a runtime scheduler, which is part of the library. Simulation of the system is performed by execution of the compiled binary, which is simple and efficient.

The concurrency model of SystemC differs from that of SpecC or Handel-C. The methods of a SystemC module may be designated as *threads* or *processes*. Interleaving between those threads is only 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 [IEEE Std 2005] allows simulators to adopt a deterministic scheduling policy. Consequently, the simulators might not explore a problematic schedule, which often prevents the discovery of concurrency-related design flaws.

When describing synchronous circuits at the register transfer level, system designers can prevent races by restricting inter-process communication to deterministic communication channels, offered by the SystemC library. However, the elimination of races from the high-level model is often not desirable. In practice, system designers use constructs that yield races in order to model nondeterministic choices implicit in the design. In particular, SystemC programs containing standard transaction-level modeling (TLM) are frequently subject to race phenomena. TLM designs usually consist of components 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. The current industrial practice is to manually instrument the SystemC model with statements that attempt to randomize the schedule chosen by the SystemC scheduler.

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 a very precise method based on Model Checking to statically precompute state predicates that predict race conditions, and to use this information subsequently during the simulation run to prune the exploration of irrelevant concurrent behaviors.

The literature suggests the use of light-weight static analysis as a prelude to simulation. For instance, Kundu et al. [2008] propose to compute the sets of variables that are read or written by the processes. Owing to the use of a much more precise analysis technique, we obtain process dependency conditions that are significantly more accurate than read/write sets, and can result in a substantially reduced number of explored interleavings. The remaining interleavings are so few that they can often be simulated exhaustively. We can control the degree of precision of our analysis to trade precision for computational cost in a sound way.

Precise process dependency conditions have applications that go beyond simulation. Dependency conditions point to the exact source of interference between two processes, and therefore provide valuable insights into the design. Our technique can prove or refute process dependencies in cases that are easily missed by the engineer or go undetected during simulation.

We have implemented this technique in Scoot [Blanc et al. 2008], a novel research compiler for SystemC. The static computation of the dependency conditions relies on a Model Checker, but the technique we propose is independent of the specific formal analysis engine. We have performed our experiments using SatAbs [Clarke et al. 2005], a SAT-based Model Checker implementing predicate abstraction, and CBMC [Clarke et al. 2003], a SAT-based bounded Model Checker. Our experimental results indicate that precise dependency conditions can be computed statically at reasonable cost, and often result in a simulation speedup of a factor of ten or better.

Outline. We discuss related work in Section 2. Then we provide an overview of the concurrency model of SystemC and background on partial-order reduction in Sections 3 and 4, respectively. Details of our implementation are reported in Section 5. Experimental results are reported in Section 6. We provide a formalization of the semantics of SystemC by means of fixed-points in Appendix A.1. Using this semantics, we present a condition for soundness of partial-order reduction for SystemC in Appendix A.2.

2. RELATED WORK

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 [2000] 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 [2007] 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 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 2000], 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 *stateless search*. VERISOFT's support for partial-order reduction relies exclusively on dynamic information to achieve the reduction. In a recent paper, Sen et al. [2008] 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. The simulator can then test these conditions at runtime to detect reduction opportunities with little overhead.

Flanagan and Godefroid [2005] describe a stateless 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. [2006] 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 prove the absence of races.

Kundu et al. [2008] 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 [2005] 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. [2008] 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.

3. THE SYSTEMC SCHEDULER

In this section, we first provide an informal review of the different phases of the SystemC scheduling algorithm. A formalization of the relevant aspects of the con-

currency model of SystemC using a fixed-point semantics is in Appendix A.1.

The dominating concurrency model for software permits asynchronous interleavings between threads, that is, the scheduler can preempt processes. SystemC is different as it is designed for modeling both asynchronous and synchronous systems. Its scheduler 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 SystemC 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 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 *wait* function. During the execution of a process, immediate notification can generate additional runnable processes. 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.

4. PARTIAL-ORDER REDUCTION FOR SYSTEMC

4.1 A Motivating Example

Program 1 serves as running example and illustrates the need for a combination of Model Checking and partial-order reduction. 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.

Program 1 A SystemC module with a 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;
  }
};
```

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

$$\mathit{pressure} \neq \mathit{PMAX} - 1 \quad \wedge \quad \mathit{pressure} \neq \mathit{PMAX}$$

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

4.2 Background on Partial-Order Reduction

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 1993; 1994; 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. Scoot uses partial-order reduction to generate a simulator that explores only necessary interleavings. We briefly survey the standard definitions from the literature in this section [Godefroid 1996].

The literature distinguishes between partial-order reduction based on *persistent sets* and reduction based on *sleep sets*. The two approaches 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.

We denote the set of states and the set of processes of the system by S and θ, respectively. As explained above, we denote the set of enabled (runnable) processes (transitions) in a state s by $\mathit{Enabled}(s)$, i.e., $\mathit{Enabled}$ is a mapping from S to $\mathscr{P}(\theta)$.

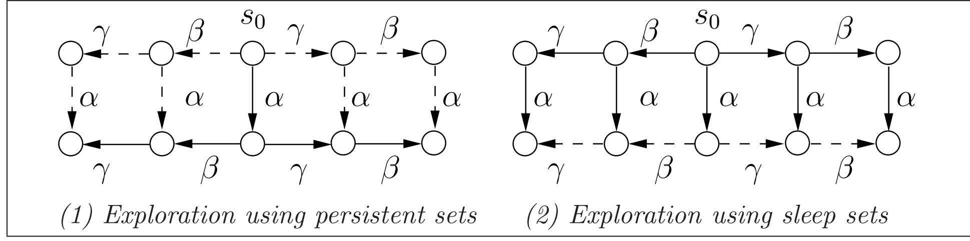

Figure 1: Example of partial-order reduction using persistent sets (1) and sleep sets (2). The reduced state graph contains only the transitions depicted with solid lines.

Definition 4.1. [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 $s \in \phi$ and $t \in S$ the following hold:

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

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. Composition $\alpha \circ \beta$ of relations α and β is defined in the usual way. 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].

Scoot uses heavy-weight techniques such as Model Checking to compute the condition ϕ. In addition, we also compute dependency relations between processes using a (comparatively) light-weight data-flow analysis. It is important to note that these dependency relations guarantee commutativity in *any* reachable state, as formalized by Godefroid [1996]:

Definition 4.2. Let $D \subseteq \theta\times\theta$ be a symmetric and reflexive relation over the transitions of the system. The relation D is a *valid dependency relation* for θ if and only if $(\alpha, \beta) \notin D$ implies that α, β are independent in all reachable states.

The persistent-set reduction technique computes a set of runnable transitions in each visited state and restricts the future exploration to transitions in this set only [Godefroid 1996].

Definition 4.3. Let (S, S_0, θ) be a transition system, and $s_0 \in S$ denote one of its states. A set of transitions $T \subset \mathit{Enabled}(s_0)$ is *persistent* in s_0 if and only if for all $\beta \in T$ and all sub-traces $s_0 \xrightarrow{\alpha_0} s_1 \xrightarrow{\alpha_1} s_2 \ldots s_n \xrightarrow{\alpha_n} s_{n+1}$ using transitions $\alpha_i \notin T$, β and α_i are independent in s_i.

Persistent sets are usually computed using information from a preliminary light-weight static analysis. The effects of the persistent-set technique are illustrated in

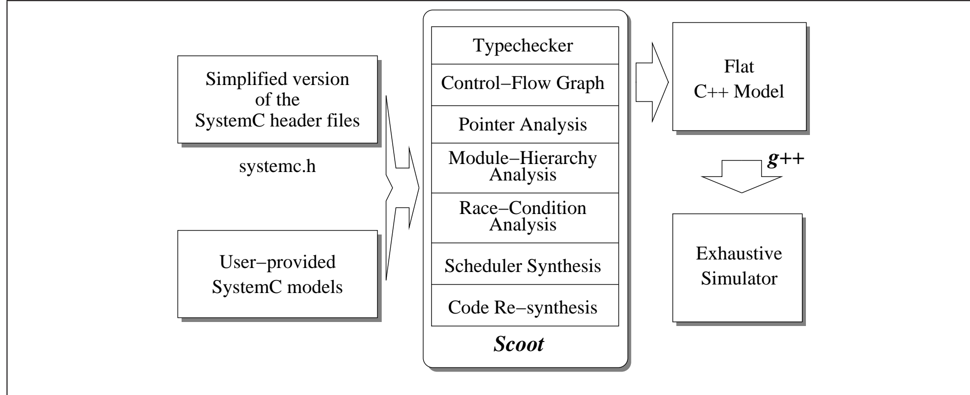

Figure 2: Overview of Scoot

Figure 1.1. 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 1.2 shows a typical exploration using sleep sets. Unlike the persistent-set 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. We formalize a soundness criterion for partial-order reduction on SystemC models in Appendix A.2.

5. A SYSTEMC SIMULATOR WITH PARTIAL-ORDER REDUCTION

5.1 Overview of Scoot

Figure 2 provides a high-level overview of Scoot. We use an in-house C++ front-end to translate the SystemC source files into a control flow graph (CFG). The front-end of Scoot accepts a large subset of C++ including inheritance, overloading, virtual functions, and many forms of templates.

Scoot abstracts implementation details of the SystemC library by using simplified header files that declare only relevant aspects of the API and omit the actual implementation. Subsequently, Scoot uses static analysis techniques to discover the module hierarchy, the sensitivity list of processes, and the port bindings. The next step is to compute the process dependencies – we elaborate on this step in Section 5.3. Finally, Scoot translates the CFG back to a flat C++ program, which no longer requires the SystemC library. We use *g++* to compile the C++ file and to obtain an executable simulator.

We forbid dynamic creation of processes and dynamic modifications of sensitivity lists (*next_trigger* functions). The support for SystemC currently comprises static creation of processes, static sensitivity lists, waiting using sensitivity lists, waiting for a specific event, waiting for a certain amount of time, immediate notification, delta notification, time notification, and communication channels such as *sc_signals*, *sc_fifos*, and *tlm_fifos*. We have a broad support for the general features of C++; e.g., our support for STL container classes is described in [Blanc et al. 2007].

Algorithm 1 Evaluation Phase: the call to `guarded_independent()` returns the independence condition for p_i and p_j. This predicate, which is computed at compile time using Model Checking, is evaluated at runtime by the function `eval()`.

```
1  void evaluation_phase()
     Set sleeps := ∅;
3    while(runnable()≠ ∅) do
       persistents := get_pers();
5      awakes := persistents \ sleeps;
       if(awakes= ∅) then exit(0);
7      Map next_sleeps; // Process -> Set
       for all (Process p_i ∈ awakes) do
9        for all (Process p_j ∈ sleep) do
           if(eval(guarded_independent(p_i,p_j)))
11           next_sleeps[p_i] := next_sleeps[p_i] ∪ {p_j};
         end for
13       sleep := sleep∪ {p_i};
       end for
15     Process p := nondet_select(awakes);
       run(p);
17     sleeps := next_sleeps[p];
     end while
```

Algorithm 2 Computation of persistent sets. The call to `guarded_independent()` returns the independence condition for p_i and p_j, which is computed at compile time using Model Checking. On line 7, the algorithm checks if this predicate is syntactically different from "*true*".

```
1  Set get_pers()
     Set persistents := runnable();
3
     for all(Process p_i ∈ Runnable()) do
5      Bool pers := false;
       for all(Process p_j) do
7        if(guarded_independent(p_i,p_j) ≠ "true") then
           pers := true; break;
9      end for

11     if(pers = false)
         persistents := persistents\{p_i};
13   end for

15   if(persistents = ∅) then
       return select_first(runnable());
17
     return persistents;
```

5.2 A Scheduler with Partial-Order Reduction

Algorithm 1 is SCOOT's implementation of the evaluation phase. Our algorithm performs partial-order reduction using persistent sets and sleep sets, and is a variation of techniques presented by Godefroid [1996]. In contrast to Helmstetter et al. [2006], `evaluation_phase()` schedules runnable processes using information collected *statically* to reduce the number of interleavings explored.

The evaluation phase terminates once the set of runnable processes is empty (line 3). The scheduler calls `get_pers()` to compute the set *persistents* of persistent processes (line 4). 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 of processes that are awake 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 `guarded_independent()` returns the independence guard for the processes p_i and p_j. This predicate, over the visible state of the system, is evaluated at runtime by the function `eval()`.

SCOOT relies on Model Checking to compute a conservative condition that guarantees independence of the processes in the current state. The details of this precomputation 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 or on an imprecise data-flow analysis.

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

Algorithm 2 sketches the function `get_pers()`, which computes the set of persistent processes[1]. On line 2, the set *persistents* of persistent processes is initialized with the set of runnable processes (the enabled transitions). Subsequently, the algorithm removes all independent processes, i.e., processes without dependencies, from *persistents* to defer their execution, thus reducing the nondeterminism of the system. On line 7, the algorithm checks if the independence guard for p_i and p_j that is returned by `guarded_independent()` is different from "*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.

5.3 Computing the Process Commutativity Conditions

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. 1. We reduce the definition of guarded independence (Def. 4.1) to process commutativity by treating processes as always enabled (if a process is not enabled then its execution simply does not modify the state). In general, SystemC processes need not terminate, and thus computing the weakest possible commutativity condition for a given pair of processes p_1 and p_2 is undecid-

[1]In addition, our actual implementation exploits invariant properties of the evaluation phase, e.g., signal values, to further reduce the set of persistent processes.

Program 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. 3.

```
1  assume(ϕ);
2  s0 := current_state;
3  p1(); p2();
4  s1,2 := current_state;
5  current_state := s0;
6  p2(); p1();
7  s2,1 := current_state;
8  assert(s1,2 ≠ s2,1);
```

able. We compute a conservative under-approximation by applying a verification engine to the harness given as Program 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. The initial values of the visible variables are stored in s_0, the pair of processes $p_1();p_2()$ is run, and the resulting state is stored in $s_{1,2}$. The state is restored to s_0, and $p_2();p_1()$ is run. The resulting state is stored in $s_{2,1}$. The assertion compares the states generated by the two process orderings.[2]

Note that SystemC distinguishes between *method processes* and *threads*. A method process is executed without interruption up to completion, whereas a SystemC thread can suspend its execution using wait statements. The construction of the harness presented in Program 2 is straightforward for method processes, as no context switch is taking place. We present a conversion technique to handle threads in a similar way as method processes in Appendix B.

Scoot passes the harness to a verification engine 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. Observe that if s satisfies P_π, then the executions of $p_1();p_2()$ and $p_2();p_1()$ from the state s

(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. Scoot then strengthens ϕ using $\neg P_\pi$, yielding ϕ'. This blocks any trace that goes through the control locations of π. Scoot iterates this process until the verification engine 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.

[2] As an optimization, we restrict the state comparison to the memory regions that are modified

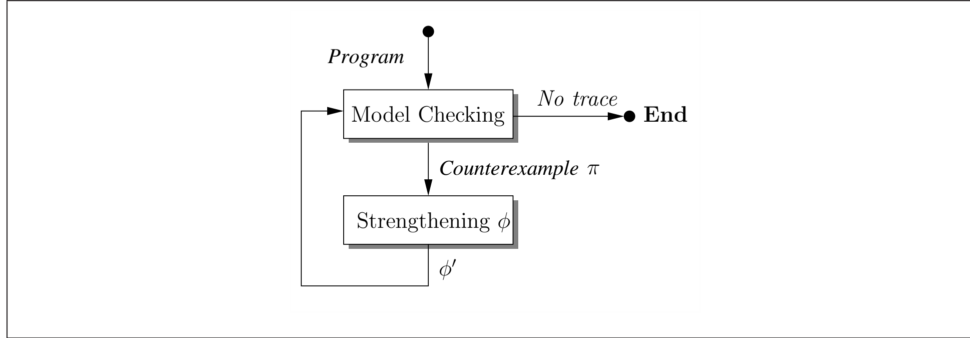

Figure 3: Iterative computation of the process commutativity condition using a verification engine. The loop strengthens ϕ until the verification engine stops reporting counterexamples.

5.4 The Running Example

We illustrate the execution of the strengthening loop presented in Figure 3 on Program 1. In the first iteration, the verification engine verifies Program 3.1, and reports a counterexample π following the lines $1, 2, 3, 5, 6, 7, 8, 9, 11$.

Program 3 Harnesses verified during the first and second strengthening iteration in Figure 3. 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.

Subsequently, we compute the weakest pre-condition of $s_{1,2} = s_{2,1}$ alongside π. We use the following axiom schemata for assignments and assume statements, in

by the processes. These locations are determined by means of a standard data-flow analysis.

backward reasoning style [Hoare 1969; Floyd 1967]:

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

Figure 4 demonstrates the computation of the pre-condition P_π alongside π. The proof starts from line 11 with the condition $s_{1,2} = s_{2,1}$.

1	`assume(true);`	
	$\{pressure + 1 \neq PMAX \wedge pressure \neq PMAX\}$	
2	$\texttt{s}_0$ `:= 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	$\texttt{s}_{1,2}$ `:= pressure;`	$R0$
	$\{s_{1,2} = s_0 + 1 \wedge s_0 + 1 \neq PMAX\}$	
7	`pressure :=` $\texttt{s}_0$`;`	$R0$
	$\{s_{1,2} = pressure + 1 \wedge pressure + 1 \neq PMAX\}$	
8	`pressure := pressure + 1;`	$R0$
	$\{s_{1,2} = pressure \wedge pressure \neq PMAX\}$	
9	`assume(pressure` $\neq$ `PMAX);`	$R1$
	$\{s_{1,2} = pressure\}$	
11	$\texttt{s}_{2,1}$ `:= pressure;`	$R0$
	$\{s_{1,2} = s_{2,1}\}$	

Figure 4: 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 $s_{1,2} = s_{2,1}$.

The algorithm then strengthens the set of initial sates in Program 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 $P \triangleq pressure{+}1 \neq PMAX \wedge pressure \neq PMAX$. Given a state s, this predicate evaluates to *true* if the processes *guard* and *increment* are independent in s. Observe that our technique enumerates only a subset of the feasible paths.

5.5 Implementation of the Strengthening Loop

In the following, we elaborate on our integration of the strengthening loop into two Model Checkers, SatAbs and CBMC. Note that our approach is independent of the particular verification engine. The general idea can be extended in different directions: In Figure 3, we use the verification engine to compute terminating paths. 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, at the cost of precision.

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 2000]. Each predicate in the abstract model is represented by a Boolean variable, while the original variables are removed. The abstract program is created using existential abstraction, which is conservative 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 from the abstract model. This process 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 5 shows the integration of our technique into SATABS. After strengthening, SATABS retains the abstract model for the following iterations.

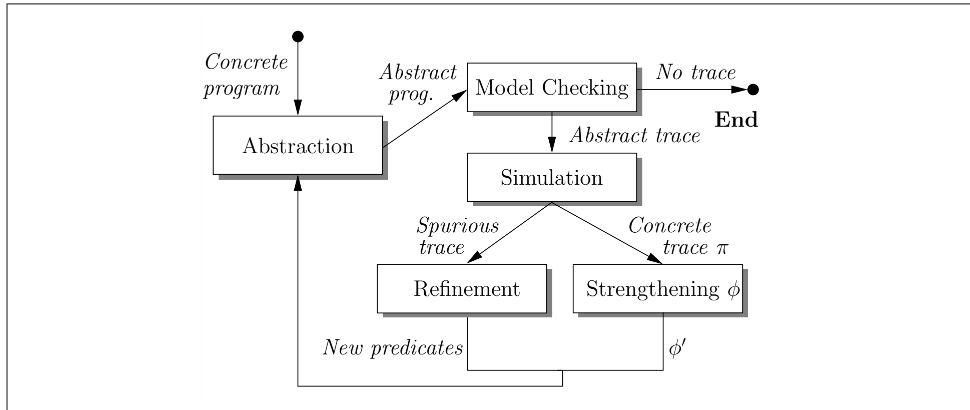

Figure 5: Iterative computation of the process commutativity condition using predicate abstraction.

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 a safety specification [Biere et al. 1999]. 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 satisfying assignment. The method is complete only if k exceeds the completeness threshold [Kroening and Strichman 2003].

We use CBMC as Bounded Model Checker [Clarke et al. 2003]. 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 6 illustrates the integration of our technique into CBMC. First, CBMC unrolls the harness (Program 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 states by adding blocking clauses to the Boolean formula.[3]

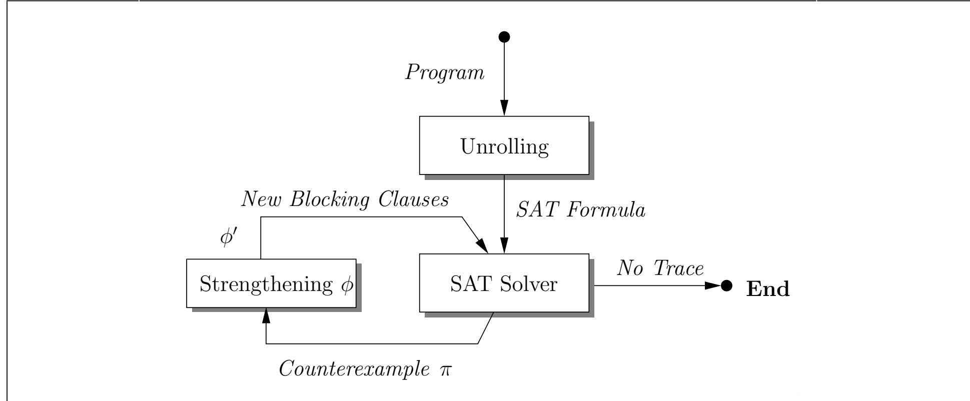

Figure 6: Iterative computation of the process commutativity condition using bounded Model Checking.

6. 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 define the precision of the dependency analysis in terms of the bound that is set on the number of strengthening iterations. "Precision 0" indicates that no iterations are performed, i.e., only the result of the light-weight analysis is used. "Precision 1" means that one strengthening iteration is performed, and so on. "Full Precision" stands for no strengthening limit. 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 quad-core machine equipped with 3 GHz Intel Xeon processors (4 MB cache per core), 8 GB of RAM, and Linux 2.6.20. We make the benchmarks and the tool available for experimentation by other researchers at *www.cprover.org/scoot/*.

[3]The SAT solver is operated in an incremental fashion, which allows it to retain all the clauses learned in previous iterations.

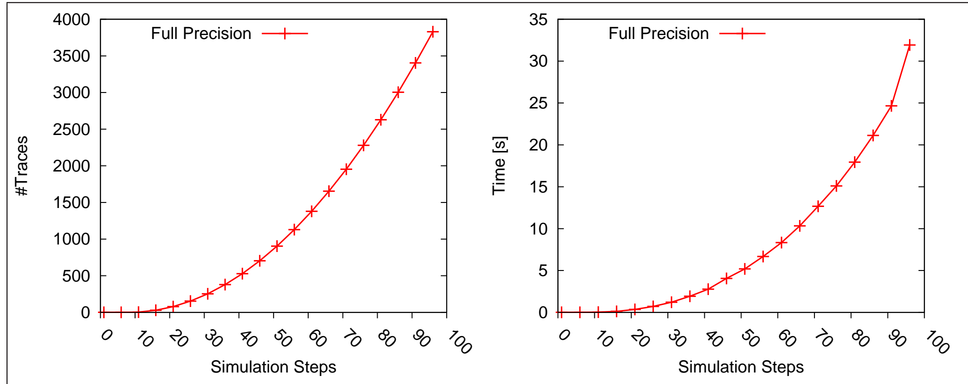

Figure 7: Total time and number of traces explored at runtime as a function of the number of simulation steps, for the running example.

6.1 The Running Example

We continue our running example (Program 1). Figure 7 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 stateless search, that is, backtracking is performed by restoring the initial state and replaying the necessary transitions.

Using our technique, the number of traces explored during simulation grows only quadratically with the number of steps, instead of exponentially. Note that there is always a data dependency between the processes *guard* and *increment* at runtime: Process *guard* always reads *pressure* and process *increment* always writes to *pressure*. Consequently, traditional dynamic partial order reduction techniques always need to fully explore the alternative schedule.

6.2 State Machine Benchmarks

Program 4 Skeleton of Benchmark B1

```
1  bool locked; int op;
   void process_client() {
3   if(!locked){ op=get_pid(); locked=true;}
   }
5  void process_server(){
     switch(state) {
7    ...
     case Idle: {switch(op) {...} break;}
9    case End: {state = Idle; locked = false;}
     }
11 }
```

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

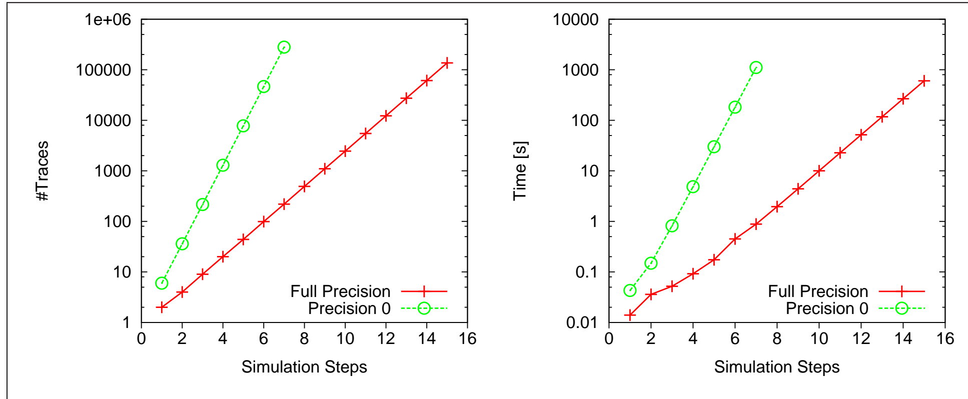

Figure 8: Performance effect of static partial-order reduction on B1

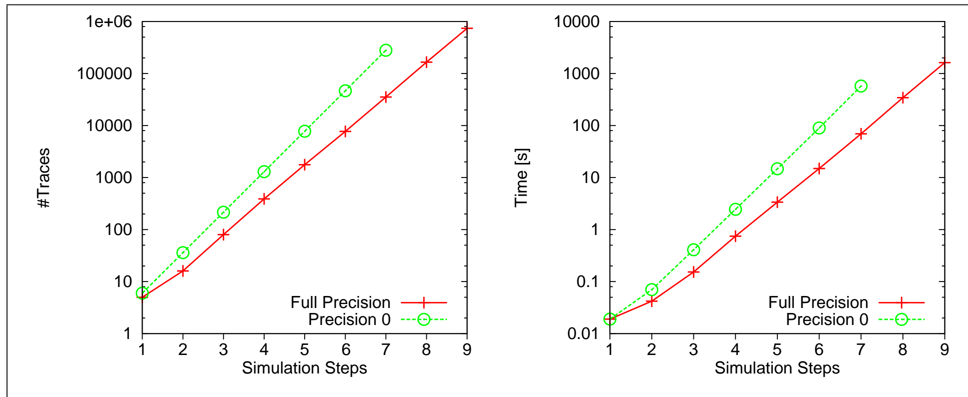

Figure 9: Performance effect of static partial-order reduction on B2

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 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 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 simulation steps.

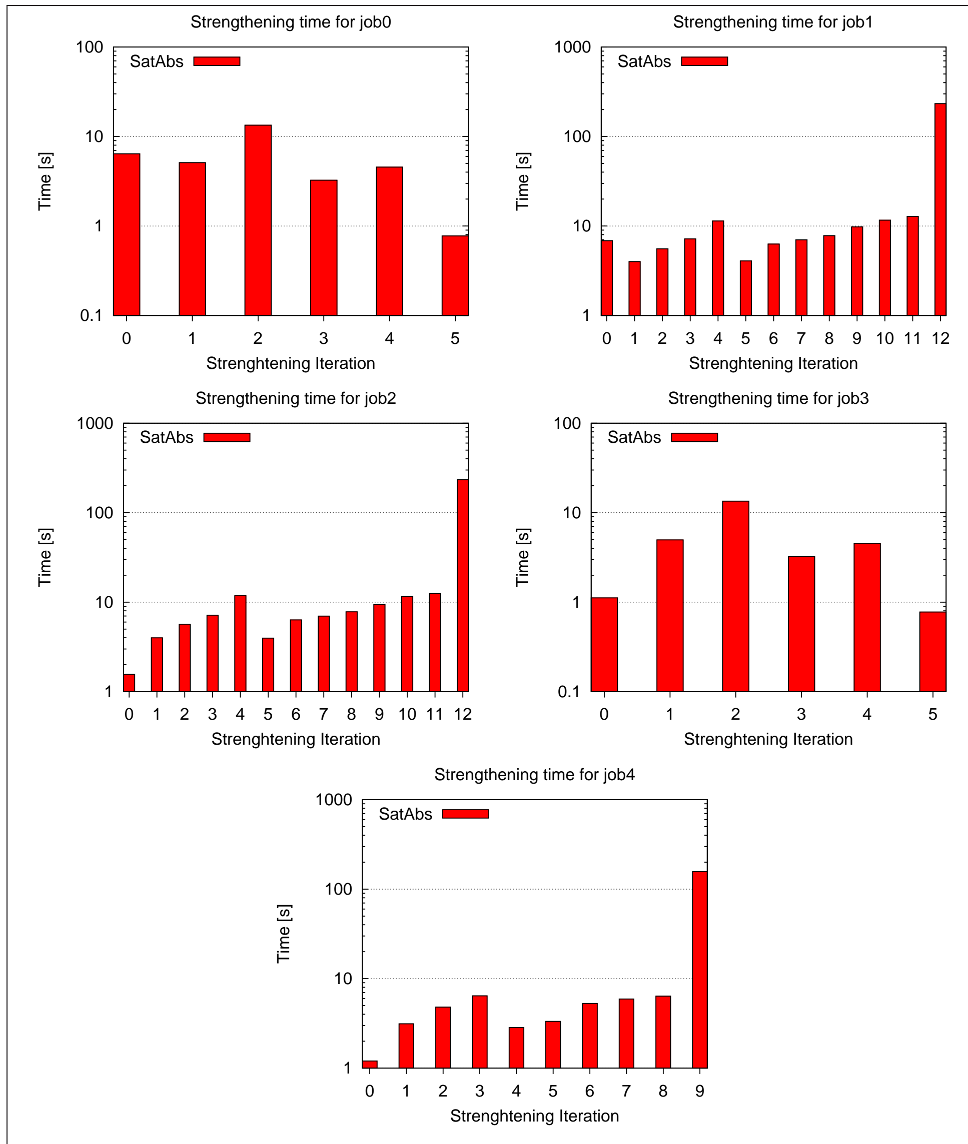

Figure 10: Details of the strengthening process for the analysis of the memory model (Program 5). Each figure corresponds to a distinct verification task and gives the time for the individual strengthening iterations using SatAbs. The last iteration proves the absence of any further counterexamples.

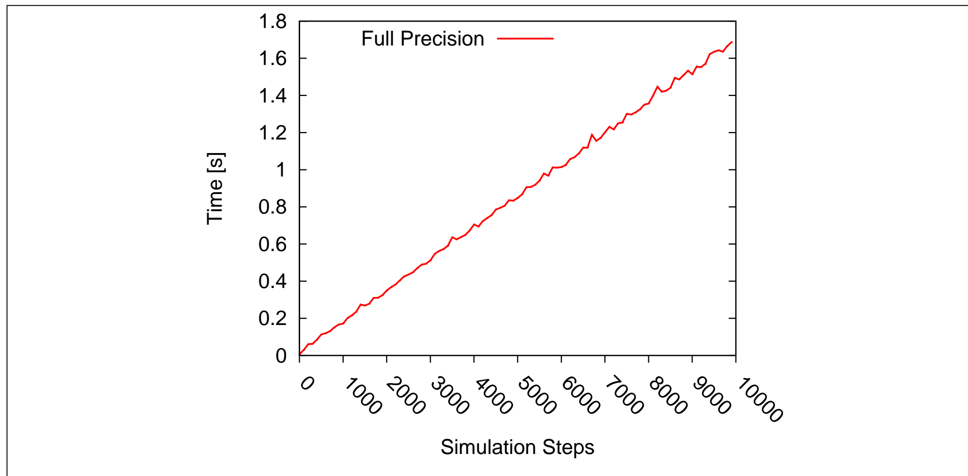

Figure 11: Simulation runtime for the RISC-CPU model. The runtime is linear in the number of steps.

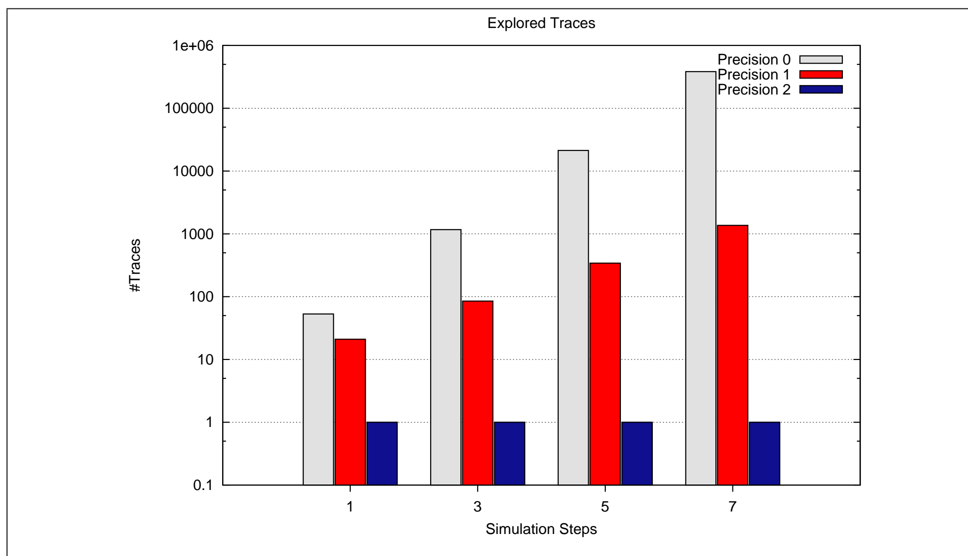

Figure 12: Impact of the precision of the static analysis on the performance of the simulation on the RISC-CPU model. The precision of the analysis is measured in terms of the bound on the number of strengthening iterations. “Precision 0” stands for the light-weight static analysis. The highest precision on this model is six. In this example, precision two is already sufficient to obtain an optimal simulation.

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

Table I. 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.

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 9 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 I 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 results 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.

6.3 An Asynchronous Dual-Port Memory

We present an instance that is difficult for any dependency analysis. Memory modules are frequently modeled 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 processes. Program 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.

Program 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  ...
```

For each pair of processes, Table II 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 instance, 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.

Figure 10 depicts the time for each strengthening iteration using SatAbs. The last strengthening iteration is used for proving the absence of additional counterexamples. Job1 and Job2 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 analyzer must prove that on lines 13 and 18, the guards of the if-statements

Jobs	Proc.	SatAbs			CBMC		
		$P \Leftrightarrow true$	#Strength.	Time [s]	$P \Leftrightarrow true$	#Strength.	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

Table II. 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	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

Table III. 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*.

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*.

6.4 A RISC Processor

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 III 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 loop in one of the processes.[4] 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 11 depicts the simulation performance. Using our precise conditions, the

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

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 12). This means that only shallow exploration is possible. We also quantify the precision obtained by limiting the number of strengthening iterations. 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.

7. CONCLUSION

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 discovered by other dynamic approaches.

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 most precise 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.

Future Work. We have presented a technique to reduce the search space with respect to interleavings explored; similar ideas apply to data nondeterminism as well. Many testing tools employ decision procedures to perform a shallow search with the goal of pruning the set of test-vectors that has to be considered. We plan to explore the application of Model Checkers in this context.

Acknowledgments

The authors are grateful to Doron Bustan (Intel Haifa) for numerous discussions regarding SystemC TLM, and to Vijay D'Silva and Thomas Wahl for suggestions that led to a significant improvement of the paper.

A. CORRECTNESS PROOF

A.1 A Formalization of the SystemC Scheduler

In this section, we provide a formal basis for analyzing SystemC models. The SystemC standard defines the semantics of the concurrency model using informal English. Ruf et al. [2001] give operational semantics for SystemC, Salem [2003] use a denotational semantics, Kroening and Sharygina [2005] use Kripke structures, Savoiu et al. [2005] use Petri-nets. In contrast to the related work, we propose to formalize the concurrency model of SystemC using a fixed-point semantics. This choice of style is motivated by the iterative nature of the scheduling algorithm (we refer the reader to Slonneger and Kurtz [1995] for an introduction). We first recall standard definitions from the literature:

THEOREM A.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 A.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 A.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.

SystemC processes are transitions. Note that the state of the system comprises not only 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 − *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 form $F(f) = f$, 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 A.2 and A.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 $f_0 = \perp$, 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.

We start with the formalization of the evaluation phase. Definition A.4 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 A.4. 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:

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

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, as $\epsilon(\{s\})$ maps to a set of states.

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 A.4 gives rise to Lemma A.5, which establishes the monotonicity of E.

Lemma A.5. *Function E (Def. A.4) 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 $E(g)$. $\square$

From Lemma A.5 and Theorems A.1, A.2 we conclude that the evaluation phase is well-defined if S is finite:

Theorem A.6. *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 A.5), it is sufficient to show that $\mathscr{P}(S) \to \mathscr{P}(S)$ is finite to prove that E is continuous (Theorem A.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 A.2 guarantees the existence of a least fixed point and provides an iterative method to compute it. $\square$

Additionally, we introduce Lemma A.7, 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 $\epsilon_0 = \perp$.

LEMMA A.7. *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. A.4) 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. □

We continue with the formalization of the delta cycle in this style. Definition A.8 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 A.8. 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:

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

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

Definition A.8 gives rise to Lemma A.9, which guarantees the monotonicity of D. Subsequently, Theorem A.10 establishes that the delta phase is well defined if S is finite.

LEMMA A.9. *Function D (Def A.8) is monotone.*

PROOF. Let us we write $\mathcal{A}$ and $\mathcal{B}$ for "$\{{\scriptstyle s\in X | Enabled(s)=\emptyset}\}$" 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 A.10. *The least fixed point δ is well-defined for models with finite memory.*

The proof of this theorem uses the same argument as the proof of Theorem A.6. We conclude the formalization of the concurrency model of SystemC with Definition A.11, which captures the simulation semantics of SystemC.

Definition A.11. 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:

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

A.2 Correctness of Partial-Order Reduction Techniques for SystemC

Partial-order reduction techniques restrict 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.[5] We formalize a condition for the soundness of partial-order reduction as follows.

Definition A.12. Let $\epsilon : \mathscr{P}(S) \rightarrow \mathscr{P}(S)$ denote the evaluation phase (Def. A.4), and let $\hat{\epsilon} : \mathscr{P}(S) \rightarrow \mathscr{P}(S)$ stand for 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 $\epsilon = \hat{\epsilon}$.

Definition A.12 and Lemma A.7 yield the following theorem that provides a method to show correctness of a partial-order reduction technique $\hat{\epsilon}$:

THEOREM A.13. *Let $\hat{\epsilon} : \mathscr{P}(S) \rightarrow \mathscr{P}(S)$ denote a function that models the evaluation phase. The function $\hat{\epsilon}$ 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, $\epsilon(\{s\}) = \hat{\epsilon}(\{s\})$.

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

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 A.14. Let $Persistent : S \rightarrow \mathscr{P}(\theta)$ denote some function that returns a set of persistent processes (Def. 4.3). Additionally, we require that $Persistent(s)$ is empty only if no process is runnable in s. We define $\hat{E}_P : \big(\mathscr{P}(S) \rightarrow \mathscr{P}(S)\big) \longrightarrow \big(\mathscr{P}(S) \rightarrow \mathscr{P}(S)\big)$ as the following higher-order function:

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

The persistent-set technique is the least solution $\hat{\epsilon}_P : \mathscr{P}(S) \rightarrow \mathscr{P}(S)$ 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 $\mathit{Sleep}(s)$ to denote the set of enabled processes in s that can be skipped; that is, $\mathit{Sleep} : S \rightarrow \mathscr{P}(\theta)$. Additionally, let $\mathit{NextSleep} : (S \times \theta) \rightarrow S$ 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 $(\mathit{Sleep} \circ \mathit{NextSleep})(s, \alpha)$ describes the next sleep set, i.e., the sleep set for the states in $\alpha(s)$. Definitions A.15 and A.16 formalize this technique.

Definition A.15. $\mathit{Sleep} \circ \mathit{NextSleep}(s, \alpha)$ is a *sleep set* if and only if $\alpha \in Enabled(s)$ and $\beta \in (\mathit{Sleep} \circ \mathit{NextSleep})(s, \alpha)$ implies that the following holds:

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

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

Definition A.16. Let $\mathit{NextSleep}(s, \rho)$ denote a function that computes sleep sets (Def. A.15). 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:

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

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

B. MODEL CHECKING SYSTEMC 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 SystemC 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.

The construction of the harness presented in Program 2 is straightforward for method processes, as no context switch is taking place. We present a technique to implement the context switches in SystemC threads with goto statements. This approach enables us to handle SystemC threads in the same way as method processes.

For each thread, our conversion technique proceeds as follows:

(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 variable to hold the location of 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 6.1 and Program 6.2 show the code of a thread before and after conversion, respectively. Program 6.1 declares a unique local variable i. After conversion, the variable is declared with static storage duration. Additionally, Program 6.2 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 6.1 correspond to the assignments followed by a goto statement in Program 6.2 (lines 11 and 14).

This conversion approach is applicable only if recursive functions are free of `wait` statements. In practice, SystemC threads rarely execute recursive function calls that contain synchronization routines.

Program 6 Example of conversion of a SystemC thread. Program 6.1 shows the original code of the thread. Program 6.2 shows the code after conversion.

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

(1) Original thread

```
   int i; int pc = 0;
2  void run()
   {
4        if(pc == 1) goto PC1;
         if(pc == 2) goto PC2;
6        if(pc == 3) goto EXIT;

8        i = 0;
         while(true){
10         if(i < 10){
             pc = 1; goto EXIT;
12 PC1:      i = i+1;
           } else {
14           pc = 2; goto EXIT;
   PC2:      i = 0;
16         }
         }
18       pc = 3;
   EXIT: ;
20 }
```

(2) Thread after conversion

REFERENCES

Ball, T. and Rajamani, S. 2000. Boolean programs: A model and process for software analysis. Tech. Rep. MSR-TR-2000-14, Microsoft Research.

Ball, T. and Rajamani, S. K. 2002. 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*. ACM, New York, NY, USA, 1–3.

Biere, A., Cimatti, A., Clarke, E. M., Fujita, M., and Zhu, Y. 1999. Symbolic model checking using SAT procedures instead of BDDs. In *DAC '99: Proceedings of the 36th ACM/IEEE conference on Design automation*. ACM, New York, NY, USA, 317–320.

Blanc, N., Groce, A., and Kroening, D. 2007. Verifying C++ with STL containers via predicate abstraction. In *22nd IEEE International Conference on Automated Software Engineering (ASE)*. IEEE, 521–524.

Blanc, N. and Kroening, D. 2008. Race analysis for SystemC using model checking. In *Proceedings of ICCAD 2008*. IEEE, 356–363.

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

Clarke, E., Jain, H., and Kroening, D. 2007. Verification of SpecC using predicate abstraction. *Form. Methods Syst. Des. 30,* 1, 5–28.

Clarke, E., Kroening, D., Sharygina, N., and Yorav, K. 2005. SATABS: SAT-based predicate abstraction for ANSI-C. In *TACAS*. Lecture Notes in Computer Science, vol. 3440. Springer.

Clarke, E., Kroening, D., and Yorav, K. 2003. Behavioral consistency of C and Verilog programs using bounded model checking. In *DAC*. ACM Press.

Clarke, E. M., Grumberg, O., Jha, S., Lu, Y., and Veith, H. 2000. Counterexample-guided abstraction refinement. In *CAV '00: Proceedings of the 12th International Conference on Computer Aided Verification*. Lecture Notes in Computer Science. Springer, London, UK, 154–169.

Cook, B., Podelski, A., and Rybalchenko, A. 2006. Terminator: Beyond safety. In *CAV*. 415–418.

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

Engler, D. and Ashcraft, K. 2003. RacerX: Effective, static detection of race conditions and deadlocks. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*. ACM, New York, NY, USA, 237–252.

Flanagan, C. and Freund, S. N. 2000. Type-based race detection for Java. In *PLDI '00: Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*. ACM, New York, NY, USA, 219–232.

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

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

Godefroid, P. 1996. *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.

Godefroid, P. 2005. Software model checking: The VeriSoft approach. *Form. Methods Syst. Des. 26,* 2, 77–101.

Graf, S. and Saïdi, H. 1997. 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. Springer, London, UK, 72–83.

Helmstetter, C., Maraninchi, F., Maillet-Contoz, L., and Moy, M. 2006. 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*. IEEE Computer Society, Washington, DC, USA, 171–178.

Hoare, C. A. R. 1969. An axiomatic basis for computer programming. *Commun. ACM 12,* 10, 576–580.

IEEE Std 2005. SystemC language reference manual. IEEE Standard 1666-2005.

Kroening, D. and Sharygina, N. 2005. 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*. IEEE Computer Society, Washington, DC, USA, 101–110.

Kroening, D. and Strichman, O. 2003. Efficient computation of recurrence diameters. In *VMCAI 2003: Proceedings of the 4th International Conference on Verification, Model Checking, and Abstract Interpretation*. Springer-Verlag, London, UK, 298–309.

Kundu, S., Ganai, M., and Gupta, R. 2008. Partial order reduction for scalable testing of SystemC TLM designs. In *DAC '08: Proceedings of the 45th annual conference on Design automation*. ACM, New York, NY, USA, 936–941.

Lamport, L. 1978. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM 21,* 7, 558–565.

Naik, M., Aiken, A., and Whaley, J. 2006. Effective static race detection for Java. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*. ACM, New York, NY, USA, 308–319.

Netzer, R. H. B. and Miller, B. P. 1992. What are race conditions? Some issues and formalizations. *ACM Lett. Program. Lang. Syst. 1,* 1, 74–88.

Peled, D. 1993. 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. Springer, London, UK, 409–423.

Peled, D. 1994. Combining partial order reductions with on-the-fly model-checking. In *CAV '94: Proceedings of the 6th International Conference on Computer Aided Verification*. Springer, London, UK, 377–390.

Qadeer, S. and Wu, D. 2004. KISS: keep it simple and sequential. *SIGPLAN Not. 39,* 6, 14–24.

Ruf, J., Hoffmann, D., Gerlach, J., Kropf, T., Rosenstiehl, W., and Mueller, W. 2001. The simulation semantics of SystemC. In *DATE '01: Proceedings of the conference on Design, automation and test in Europe*. IEEE Press, Piscataway, NJ, USA, 64–70.

Salem, A. 2003. Formal semantics of synchronous SystemC. In *DATE '03: Proceedings of the conference on Design, Automation and Test in Europe*. IEEE Computer Society, Washington, DC, USA, 10376.

Savage, S., Burrows, M., Nelson, G., Sobalvarro, P., and Anderson, T. 1997. Eraser: A dynamic data race detector for multithreaded programs. *ACM Trans. Comput. Syst. 15,* 4, 391–411.

Savoiu, N., Sandeep, S., and Rajesh, G. 2005. Improving SystemC simulation through Petri net reductions. In *MEMOCODE*. 131–140.

Sen, A., Ogale, V., and Abadir, M. S. 2008. Predictive runtime verification of multi-processor SoCs in SystemC. In *DAC '08: Proceedings of the 45th annual conference on Design automation*. ACM, New York, NY, USA, 948–953.

Slonneger, K. and Kurtz, B. 1995. Domain theory and fixed-point semantics. In *Formal Syntax and Semantics of Programming Languages: A Laboratory Based Approach*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 341–394.

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

Wang, C., Yang, Z., Kahlon, V., and Gupta, A. 2008. Peephole partial order reduction. In *TACAS*. Lecture Notes in Computer Science. Springer, 382–396.

Witkowski, T., Blanc, N., Kroening, D., and Weissenbacher, G. 2007. Model checking concurrent Linux device drivers. In *ASE '07: Proceedings of the 22nd IEEE/ACM international conference on Automated software engineering*. ACM, New York, NY, USA, 501–504.