Race Analysis for SystemC using Model Checking

Nicolas Blanc
ETH Zurich, Switzerland
nicolas.blanc@inf.ethz.ch

Daniel Kroening
Oxford University, Computing Laboratory, UK
Daniel.Kroening@comlab.ox.ac.uk

***Abstract*—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 formal *and* scalable race analysis. This analysis combines both classic static analysis and Model Checking techniques. The outcome of the analysis is not only valuable to diagnose the effect of race conditions, but can also be used to improve simulation performance dramatically. Our compiler produces a simulator that uses the race analysis information at runtime to perform partial-order reduction, thereby eliminating context switches that do not affect the result of the simulation. Experimental results show simulation speedups of one order of magnitude and better.**

I. Introduction

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 C-like system design languages has emerged. They promise joint modeling of both the hardware and software component of a system using a language that is well-known to engineers.

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 of transactions down to the gate level. The execution model of SystemC is driven by events that start or resume processes. In addition to communication via shared variables, processes can exchange information through predefined communication channels such as signals and FIFOs.

Technically, SystemC programs rely on a C++ template 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. Methods of a module 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.

This research is supported by ETH research grant TH-21/05-1 and by the Semiconductor Research Corporation (SRC) under contract no. 2006-TJ-1539.

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

1) Using the statically computed dependency relations between the threads, we can generate a static scheduler, replacing the dynamic scheduler shipped with the SystemC library. This accelerates simulation using a single, deterministic schedule.
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 [1], a novel research compiler for SystemC. The static computation of the race conditions relies on the Model Checking engine of SATABS [2], a SAT-based model checker implementing predicate abstraction. 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.

Outline: We discuss the related work and the basics of partial-order reduction in Sec. II and Sec. III. The use of a

Model Checker to obtain dependency information is motivated by means of an example in Sec. IV. We formalize the relevant semantics of the SystemC scheduler in Sec. V. Experimental results are reported in Sec. VII.

II. 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, thus corrupting the consistency of the system [3]. Flanagan and Freund use a formal type system to detect race-condition patterns in Java [4]. *Eraser* [5] is a dynamic data-race detector for concurrent applications. It uses binary rewriting techniques to monitor shared variables and to find failures of the locking discipline at runtime. Other tools, such as *RacerX* [6] and *Chord* [7], rely on classic pointer-analysis techniques to statically detect data races. Data races can also occur in SystemC if processes call synchronization routines while holding shared resources. SystemC offers semaphores and mutex variables for protecting critical sections.

Model Checkers are frequently applied to the verification of concurrent applications; see [8] for a survey on software Model Checking. Vardi identifies formal verification of SystemC models as a research challenge [9]. 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* [10] is a tool for the verification of Linux device drivers. It places the modules into a concurrent environment and relies on SATABS for the verification. *KISS* [11] is a tool for the static analysis of multi-threaded programs written in C. 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 [12], a Model Checker based on *Predicate Abstraction* [13], [14], to verify the sequential model.

Verisoft [15] is a popular tool for the systematic exploration of the state space of concurrent applications 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 [16]. 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 analysis is very precise, as it is based on a Model Checker, and SCOOT is therefore able to detect opportunities for partial-order reduction with little overhead during simulation.

Flanagan and Godefroid describe a state-less search technique with support for partial-order reduction [17]. 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 [18]. The procedure explores alternative schedules until all relevant traces are discovered. Helmstetter et al. present a partial-order reduction technique for SystemC [19]. Their approach relies on dynamic information and is similar to Flanagan and Godefroid's technique [17]. Their simulator starts with a random execution, and observes visible operations to detect dependency between the processes and to fork the execution. Our technique performs a powerful analysis statically that is able to discover partial-order reduction opportunities not detectable using only dynamic information.

Kundu et al. propose to compute read/write dependencies between SystemC processes using a path-sensitive static analysis [20]. At runtime, their simulator starts with a random execution and detects dependent transitions using static information. The novelty of our approach is to combine conventional 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 [21]. Their idea is to compute a condition (or guard) that holds in the states where two specific transitions are independent. We show how to compute these conditions for SystemC using a Model-Checking approach based on Predicate Abstraction.

III. Background on Partial-Order Reduction

Model Checking is an algorithmic technique for exhaustive exploration of transition systems. 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 [22], [23], [24]. 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 compile a simulator that explores only necessary interleavings. We briefly survey the standard definitions from the literature in this section [24].

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 enabled 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 *enabled* in a state s if there exists a state t such that $s \xrightarrow{\alpha} t$, and we write $\alpha \in Enabled(s)$ to denote this fact. Otherwise, α is *disabled* in s.

Definition 1. *[21] 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. $\alpha \in Enabled(s) \Rightarrow$
   $\beta \in Enabled(s) \Leftrightarrow \beta \in Enabled(\alpha(s))$
2. $\beta \in Enabled(s) \Rightarrow$
   $\alpha \in Enabled(s) \Leftrightarrow \alpha \in Enabled(\beta(s))$
3. $\alpha, \beta \in Enabled(s) \Rightarrow \alpha(\beta(s)) = \beta(\alpha(s))$

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. SCOOT uses Model Checking to compute the condition ϕ. Transitions α and β are *independent in s* if and only if α, β are guarded independent with respect to $\{s\}$ [24].

Definition 2. *[24] 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.*

Similar to [20], SCOOT uses a data-flow analysis in order to compute an over-approximating dependency relation.

Definition 3. *[24] Let $\mathcal{T} = (S, S_0, \rightarrow)$ be a transition system, and $s_0 \in S$ denote one of its states. A set of transitions $T \subset Enabled(s_0)$ is* persistent *in s_0 if and only if for all $\beta \in T$ and all sub-traces $s_0 \xrightarrow{\alpha_0} s_1 \xrightarrow{\alpha_1} s_2 ... s_n \xrightarrow{\alpha_n} s_{n+1}$ obtained from transitions $\alpha_i \notin T$, β and α_i are independent in s_i.*

Definition 3 is, thus, concerned about what can happen in the *future*. The persistent-set technique computes a persistent set of enabled transitions in each visited state and restricts the exploration to transitions in this set only. Model Checkers typically compute persistent sets using information from static analysis.

In contrast, the sleep-set technique maintains a set of enabled transitions that can be skipped during the exploration (the sleep set). The method is concerned with branching information from the *past*. 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 enabled transitions.

IV. Introductory example

Program 1 serves as running example and illustrates the need for a Model Checking approach. The module *m* declares two processes *guard* and *increment*. Process *guard* watches the value of shared variable *pressure*, which shall not exceed 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 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;
  }
};
```

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. However, such analysis fails 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, the execution of *increment* and *guard* is commutative if and only if

$$pressure \neq PMAX - 1 \land pressure \neq PMAX$$

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

V. Modeling the SystemC Scheduler

In this section, we present a formalization of the SystemC concurrency model in terms of fix-point computations over the reachable states of the model.

Partial-order reduction has been studied mainly in the context of asynchronous concurrent programs, in which running processes are preempted. SystemC is different as it is designed for simulation of synchronous models. 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 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. 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.

Formally, let S denote the set of states of a SystemC model. A process ρ is a function $S \longrightarrow 2^S$. Note that the execution of the process may not terminate, or may abort with an error. We assume the existence of a failure state $\perp \in S$ such that $\perp \in \rho(s)$ if the execution of the process can diverge when started in state s. We denote the set of runnable processes in s by $Runnable(s)$. The evaluation phase $Ev : 2^S \rightarrow 2^S$ performs a fix-point computation defined by:

$$Ev(S) = \{_{s \in S | Runnable(s) = \emptyset}\} \cup Ev(\bigcup_{s \in S} \bigcup_{\rho \in Runnable(s)} \rho(s))$$

Similarly, we write $Up : 2^S \rightarrow 2^S$ to denote the function that updates the set of states as described by the update phase. The delta cycle performs the fix-point computation defined by:

$$\delta(S) = \delta \circ Up \circ Ev(S)$$

Finally, let $Up_{time} : 2^S \rightarrow 2^S$ denote the function that updates the simulation time and notifies the processes waiting for this event. We model the semantics of the scheduler with the function $Sim(t)$ that computes the set of states at time t. $Sim(0)$ is the set of initial states.

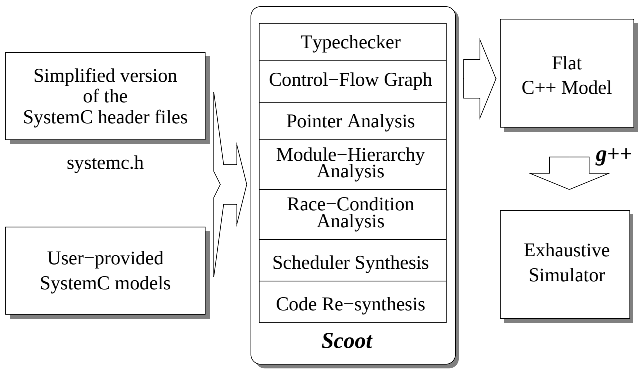
$$Sim(t) = \delta \circ Up_{time} \circ Sim(t-1)$$

VI. Implementation

A. Overview of Scoot

Figure 1 shows an 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 simple forms of templates.

Figure 1 Overview of Scoot



Algorithm 2 Computation of persistent sets

```
Set get_pers(Set runnable)
 Set persistents;
 for all (Process $p_i \in$ runnable) do
  for all (Process $p_j \in$ runnable) do
   if($p_j \geq p_i$) then continue;
   if(commutative($p_i, p_j$)) then
     if($p_i \notin$ persistents) then
       persistents := persistents $\cup \{p_j\}$;
   else
     persistents := persistents $\cup \{p_i, p_j\}$;
 return persistents;
```

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 the computation of race conditions for each pair of processes, which is explained in Sec. VI-C. Scoot then generates the code for the exhaustive simulator. 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, delta notification, time notification, and communication channels such as *sc_signals*, *sc_fifos*, and *tlm_fifos*.

B. A Scheduler with Partial-Order Reduction

Algorithm 1 is Scoot's implementation of the evaluation phase. The soundness of the scheduling algorithm relies on the assumption that processes cannot enable each other during the evaluation phase and therefore, that event notification is

Algorithm 1 Evaluation Phase: the commutativity condition checked by $\text{commutative}(p_i, p_j)$ is a predicate over states computed statically at compile-time.

```
   void evaluation_phase(Set runnable)
2    Set sleeps;
     while (runnable$\neq \emptyset$ ) do
4      if(runnable=$\{p\}$) then begin
         runnable := $\emptyset$; run($p$); return;
6      end;
       independents := get_indep(runnable);
8      for all(Process $p_i \in$ independents) do
         runnable := runnable$\setminus\{p_i\}$; run($p_i$);
10     end for;
       persistents := get_pers(runnable);
12     awakes := persistents \ sleeps;
       if(awakes=$\emptyset$) then exit(0);
14     Map next_sleeps; // Process -> Set
       for all (Process $p_i \in$ awakes) do
16       for all (Process $p_j \in$ sleeps) do
           if(commutative($p_i, p_j$))
18           next_sleeps[$p_i$] :=
               next_sleeps[$p_i$]$\cup\{p_j\}$;
20       end for
         sleeps := sleeps$\cup\{p_i\}$;
22     end for
       Process $p$ := nondet_select(awakes);
24     runnable := runnable$\setminus\{p\}$; run($p$);
       sleeps := next_sleeps[$p$];
26   end while
```

restricted to time notification and delta notification.[1] This restriction is benign in the context of system-level modeling and gives rise to the following theorem:

Theorem 1. *Suppose that for a process α that is runnable in a state $s \in S$, the execution of the process α does not affect the set of runnable processes:*

$$Runnable(\alpha(s)) = Runnable(s)\setminus\{\alpha\}$$

Any two processes $\alpha, \beta \in Runnable(s)$ that obey this restriction are independent in s if they are commutative in s.

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 *runnable* is empty. The algorithm performs partial-order reduction using persistent sets and sleep sets and is a variation of techniques presented in [24]. In line 4, if *runnable* contains one process only, then the scheduler triggers its execution and returns.

Otherwise, the procedure retrieves the set of independent processes. SCOOT statically computes a dependency condition for each pair of processes using a location- and field-sensitive pointer analysis. At simulation time, the scheduler calls *get_indep()* (line 7) to search for runnable processes without data dependencies and adds all such processes to *independents*. Subsequently, on lines 8–9, the scheduler runs all $p_i \in$ *independents* in a deterministic order.

On line 11, the algorithm 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 12, the set *awakes* consists of the persistent processes *not* in *sleeps*. If the set of awaken processes is empty (line 13), 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 14–22). One line 17, the call to *commutative* returns *true* if the processes p_i and p_j are independent in the current 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, on lines 23–25, the scheduling algorithm nondeterministically runs a process from *awakes* and computes the sleep set of the next iteration.

Algorithm 2 computes the set of persistent processes and is the implementation of the function *get_pers()*. As mentioned above, the call to *commutative* returns *true* if the processes p_i and p_j are independent in the current state. In case p_i and p_j are dependent, the scheduler adds both processes to *persistents*, which ensures that both schedules (p_i, p_j) and (p_j, p_i) are explored (line 10). Otherwise, p_j is inserted only if p_i is absent from *persistent*. Informally, the scheduler tries to avoid the execution of (p_j, p_i) if (p_i, p_j) is going to be explored.

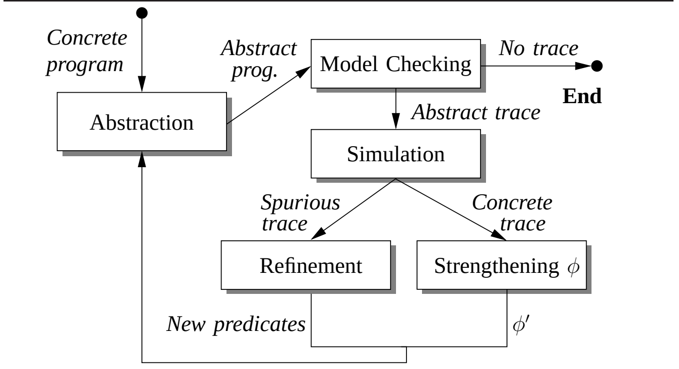
C. Computing the Process Commutativity Conditions

Predicate Abstraction is a Model Checking 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 [13], [14]. 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 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. 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

[1]This rules out immediate notifications. The technique can be extended to support immediate notification by augmenting the computation of the process commutativity condition with support for the set of notified processes, as suggested in [20].

spurious trace. This is automated by *Counterexample Guided Abstraction Refinement* (CEGAR) [25] and promoted by the Model Checker SLAM [12]. Predicate abstraction has been applied to SpecC [26] and SystemC [27].

Figure 2 Iterative computation of the process commutativity condition using predicate abstraction



The commutativity condition for a given pair of processes p_1 and p_2 is checked during simulation by Alg. 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 software model checker 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. 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 $s_{2,1}$.

This harness is passed to the Model Checker SATABS [2]. SATABS checks the reachability of the last line, which is modeled by means of an assertion. If SATABS returns a counterexample, we have a trace π with an initial state satisfying 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. P_π is non-trivial due to the assertion on line 8, which guarantees progress. The executions of $p_1();p_2()$ and $p_2();p_1()$ from a state s terminate and yield an equal state if s satisfies P_π.

Finally, SCOOT strengthens ϕ using $\neg P_\pi$, yielding ϕ'. This removes the trace π. This procedure iterates until all terminating traces are discovered. 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. Note that the procedure is easily integrated into the standard counterexample-guided abstraction refinement loop (Figure 2). Thus, there is no need to restart the abstraction procedure, and the abstract model obtained during the previous iteration is retained.

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. 2.

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

In practice, we observe that the number of facts that SCOOT tracks during the computation of the weakest precondition of $s_{1,2} = s_{2,1}$ may explode. Therefore, instead of comparing the entire state vectors $s_{1,2}$ and $s_{2,1}$, we restrict the comparison to the variables written by the processes. This set is determined by means of a standard data-flow analysis.

We have implemented this procedure in SCOOT using SATABS for model-checking.

VII. EXPERIMENTAL EVALUATION

In this section, we evaluate the benefits of combining partial-order reduction techniques with 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. We obtained the results on a 3GHz Linux machine. The race analysis uses the model-checking engine of SATABS, and the abstract programs are verified using Cadence SMV. We make the benchmarks and the tool available for experimentation by other researchers at *www.cprover.org/scoot/*.

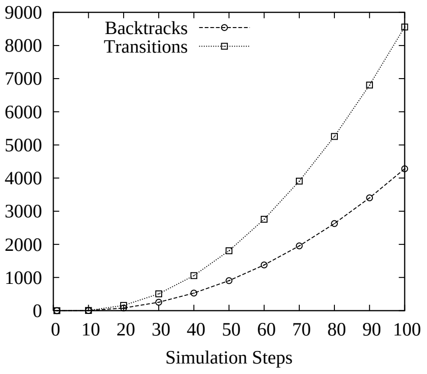
A. The Running Example

We continue our running example (Program 1). Figure 3 depicts the number of backtracks and the number of explored transitions as a function of the number of simulation steps. We set *PMAX* to *10*. The simulator performs a state-less search, that is, backtracking is an expensive operation, as the simulator has to replay the prefix. The partial-order reduction combines persistent sets and sleep sets. In this example, only persistent sets actually achieve a reduction. With this technique, both numbers grow polynomially in the number of steps, whereas without partial-order reduction, the number of backtracks grows exponentially.

B. State Machines

We use two large-scale benchmarks to evaluate the effect of statically computed race conditions. The first benchmark (B1) consists of a synchronous model with three processes. One process plays the role of a server waiting for requests, while the other two compete for access to the service. Program 3

Figure 3 Number of backtracks and transitions on the running example as a function of the number of simulation steps



Program 3 Multiple Clients

```
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;}
   }
}
```

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 4 compares the number of explored transitions, the number of backtracks, and the total exploration time as a function of the number of simulation steps. We present results without partial-order reduction (*No-POR*), using persistent sets (*P*), using sleep sets (*S*), and using their combination (*P+S*). The exploration is limited to 10^7 transitions.

The results indicate that partial-order reduction using statically computed commutativity conditions is able to reduce the number of explored transitions, the number of backtracks, and the exploration time by about three orders of magnitude.

Our second benchmark (B2) consists of two synchronous state machines communicating via shared variables. The model has three processes. The state machines are implemented using case switches. Figure 5 compares the effects of different partial-order-reduction techniques on the simulation times. The reduction is not as impressive as on B1, but still within the range of one to two orders of magnitude.

For each pair of processes, Table I shows the time required for the static analysis running SatAbs and the number of

Benchmark	Pair	SatAbs [s]	# Strengthenings
B1	0	< 1	3
B1	1	23	17
B1	2	21	17
B2	0	1111	65
B2	1	396	24
B2	2	638	23

TABLE I: Runtime and number of iterations required to compute the race conditions for each of the process-pairs

strengthening iterations. The latter is an indicator of the complexity of the control flow. The cost for B1 is negligible; the results for B2 indicate that a precise analysis can be time consuming. However, the computation 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 shown by the experiments, the time required for a full exploration grows exponentially with the number of simulation steps, and therefore, the time spent statically for a precise analysis eventually pays off.

VIII. 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 (hierarchy, port bindings) is computed at compile time by means of a value-set analysis. We use a second value-set analysis to detect independent processes. The next step is to invoke a modified software Model Checker on each pair of 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 detect reduction opportunities that depend on subtle control-flow properties.

The experimental results indicate that our formal race-analysis technique 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.

Acknowledgments

The authors are grateful to Doron Bustan (Intel Haifa) for numerous discussions regarding SystemC TLM, and to

Figure 4 Performance effect of static partial-order reduction on B1

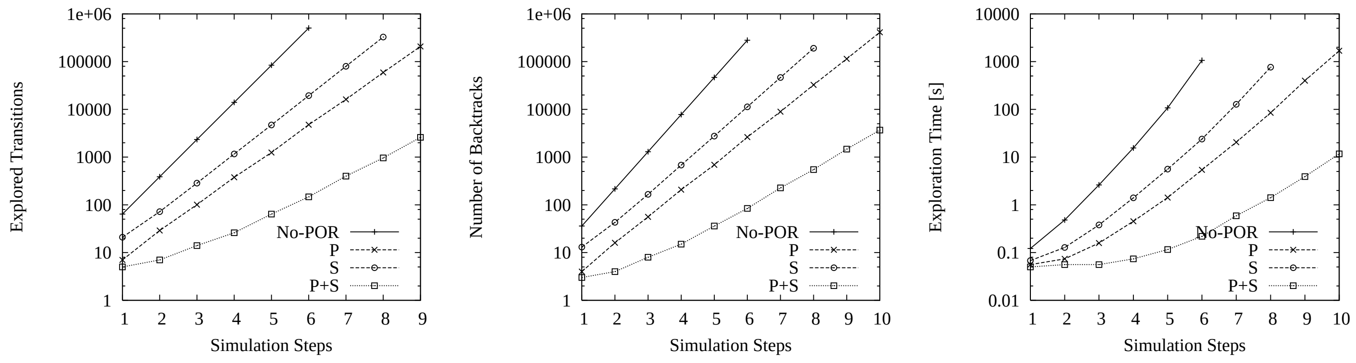
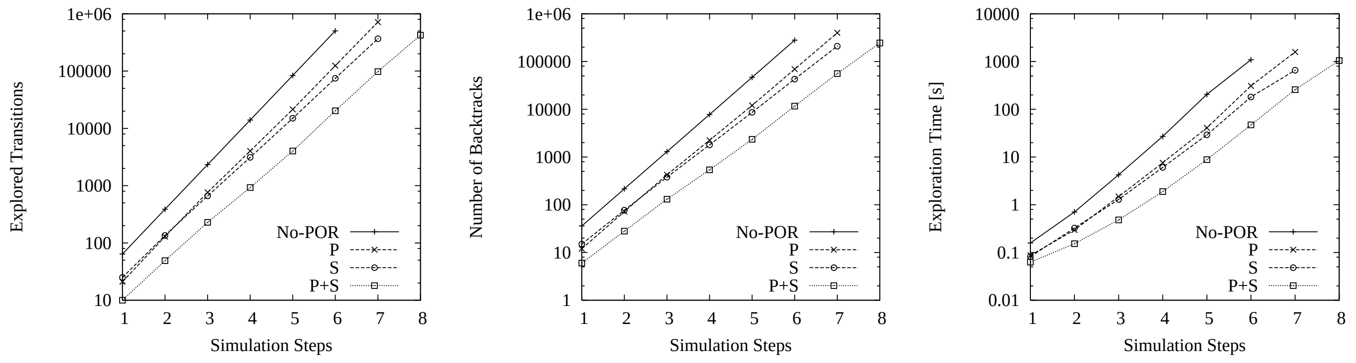


Figure 5 Performance effect of static partial-order reduction on B2



Thomas Wahl for suggestions that led to a significant improvement of the paper.

References

[1] N. Blanc, D. Kroening, and N. Sharygina, “Scoot: A tool for the analysis of SystemC models,” in *TACAS*. Springer, 2008.

[2] E. Clarke, D. Kroening, N. Sharygina, and K. Yorav, “SATABS: SAT-based predicate abstraction for ANSI-C,” in *TACAS*. Springer, 2005.

[3] R. H. B. Netzer and B. P. Miller, “What are race conditions? Some issues and formalizations,” *ACM Lett. Program. Lang. Syst.*, 1992.

[4] C. Flanagan and S. N. Freund, “Type-based race detection for Java,” *SIGPLAN Not.*, 2000.

[5] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson, “Eraser: a dynamic data race detector for multithreaded programs,” *ACM*, 1997.

[6] D. Engler and K. Ashcraft, “RacerX: Effective, static detection of race conditions and deadlocks,” in *SOSP*. ACM, 2003.

[7] M. Naik, A. Aiken, and J. Whaley, “Effective static race detection for Java,” *SIGPLAN Not.*, 2006.

[8] V. D’Silva, D. Kroening, and G. Weissenbacher, “A survey of automated techniques for formal software verification,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, vol. 27, no. 7, pp. 1165–1178, July 2008.

[9] M. Y. Vardi, “Formal techniques for SystemC verification,” in *DAC*, 2007, pp. 188–192.

[10] T. Witkowski, N. Blanc, D. Kroening, and G. Weissenbacher, “Model checking concurrent Linux device drivers,” in *ASE*. ACM, 2007.

[11] S. Qadeer and D. Wu, “KISS: Keep it simple and sequential,” *SIGPLAN Not.*, 2004.

[12] T. Ball and S. K. Rajamani, “The SLAM project: Debugging system software via static analysis,” in *POPL*. ACM, 2002.

[13] S. Graf and H. Saïdi, “Construction of abstract state graphs with PVS,” in *CAV*. Springer, 1997.

[14] T. Ball and S. Rajamani, “Boolean programs: A model and process for software analysis,” Microsoft Research, Tech. Rep., 2000.
[15] P. Godefroid, “Software model checking: The VeriSoft approach,” *Form. Methods Syst. Des.*, 2005.
[16] A. Sen, V. Ogale, and M. Abadir, “Predictive runtime verification of multi-processor SoCs in SystemC,” in *DAC*. IEEE/ACM, 2008.
[17] C. Flanagan and P. Godefroid, “Dynamic partial-order reduction for model checking software,” *SIGPLAN Not.*, 2005.
[18] L. Lamport, “Time, clocks, and the ordering of events in a distributed system,” *Commun. ACM*, 1978.
[19] 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*. IEEE, 2006.
[20] S. Kundu, M. Ganai, and R. Gupta, “Partial order reduction for scalable testing of SystemC TLM designs,” in *DAC*, 2008.
[21] C. Wang, Z. Yang, V. Kahlon, and A. Gupta, “Peephole partial order reduction,” in *TACAS*, 2008.
[22] D. Peled, “Combining partial order reductions with on-the-fly model-checking,” in *CAV*. Springer, 1994.
[23] ——, “All from one, one for all: on model checking using representatives,” in *CAV*. Springer, 1993.
[24] P. Godefroid, *Partial-Order Methods for the Verification of Concurrent Systems – An Approach to the State-Explosion Problem*. Springer, 1996.
[25] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, “Counterexample-guided abstraction refinement,” in *CAV*, 2000.
[26] E. Clarke, H. Jain, and D. Kroening, “Verification of SpecC using predicate abstraction,” *Formal Methods in System Design (FMSD)*, vol. 30, no. 1, pp. 5–28, 2007.
[27] D. Kroening and N. Sharygina, “Formal verification of SystemC by automatic hardware/software partitioning,” in *MEMOCODE*, 2005.