Speeding Up Simulation of SystemC Using Model Checking*

Nicolas Blanc[1] and Daniel Kroening[2]

[1] ETH Zurich, Switzerland
[2] Oxford University, Computing Laboratory, UK

Abstract. SystemC is a system-level modeling language that offers a wide range of features to describe concurrent systems. The SystemC standard permits simulators to implement a deterministic thread scheduling policy, which often hides concurrency-related design flaws. We present a novel compiler for SystemC that integrates a formal race analysis based on Model Checking techniques. The key insight to make the formal analysis scalable is to apply the Model Checker only to small partitions of the model. 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.

1 Introduction

Time-to-market requirements have rushed the Electronic Design 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. A key feature is joint modeling of both the hardware and software component of a system using a language that is well-known to engineers. A promising candidate for an industry standard is SystemC.

SystemC offers a wide range of language features such as hierarchical design by means of a hierarchy of modules, arbitrary-width bit-vector types, and concurrency with related synchronization mechanisms. SystemC permits different levels of abstraction, from a very high-level specification with big-step transactions down to the gate level. The execution model of SystemC is driven by *events*, which start or resume processes. In addition to communication via shared variables, processes can exchange information through predefined communication channels such as signals and FIFOs.

* This paper is an extended version of a conference paper that appeared at ICCAD 2008 [1]. 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.

M.V.M. Oliveira and J. Woodcock (Eds.): SBMF 2009, LNCS 5902, pp. 1–16, 2009.

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 yet 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.

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. In general, proving or refuting process independence requires precise static analysis. From a designer perspective, the statically computed dependency relations between the threads provide key insights into potential races.
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 [2], a novel research compiler for SystemC. The static computation of the race conditions relies on a Model Checker. The technique we propose is independent of the specific formal engine. We have performed our experiments using SatAbs [3], a SAT-based Model Checker implementing predicate abstraction, and CBMC, a SAT-based bounded Model Checker. 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.

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 [4]. Flanagan and Freud use a formal type system to detect race-condition patterns in Java [5]. *Eraser* is a dynamic data-race detector for concurrent applications [6]. It uses binary rewriting techniques to monitor shared variables and to find failures of the locking discipline at runtime. Other tools, such as *RacerX* [7] and *Chord* [8], 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.

Model Checkers are frequently applied to the verification of concurrent applications, and SystemC programs are an instance; see [9] for a survey on software Model Checking. Vardi identifies formal verification of SystemC models as a research challenge [10]. 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 [11]. 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 [12]. 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 [13], a Model Checker based on *Predicate Abstraction* [14,15], to verify the sequential model.

Verisoft is a popular tool for the systematic exploration of the state space of concurrent applications [16] 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 [17]. 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 [18]. 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 [19]. The procedure explores alternative schedules until all relevant traces are discovered. Helmstetter et al. present a partial-order reduction technique for SystemC [20]. Their approach relies on dynamic information and is similar to Flanagan and Godefroid's technique [18]. 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 [21]. 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 [22]. 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.

2 Partial-Order Reduction for SystemC

In this section, we provide a brief introduction to the concurrency model of SystemC and describe the challenges of applying partial-order reduction in the context of SystemC.

2.1 An Overview of the Concurrency Model of SystemC

The dominating concurrency model for software permits asynchronous interleavings between threads, that is, running processes are preempted. SystemC is different as it is mainly designed for modeling 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. 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.

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;
  }
};
```

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.

2.2 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 the process *increment* is triggered before *guard*. It is clear that the number of traces grows exponentially with the number of clock cycles. As a result, systematic exploration of all interleavings rapidly becomes unmanageable, and the bad behavior might go unnoticed.

A conventional static analysis can discover that *guard* reads the pressure and that *increment* modifies the pressure, concluding that the processes are indeed

dependent and that all interleavings must be explored. Similarly, a conventional dynamic analysis would always detect a read/write dependency between *guard* and *increment*, forcing the simulator to execute all schedules. However, such analyses fail to detect that *guard* and *increment* are commutative in most cases. Our tool uses a Model Checker to compute the weakest predicate over the pre-state variables that guarantees the absence of races between the processes. In this example, it is easy to see that the execution of *increment* and *guard* is commutative if and only if

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

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

2.3 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 [23,24,25]. 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 [25].

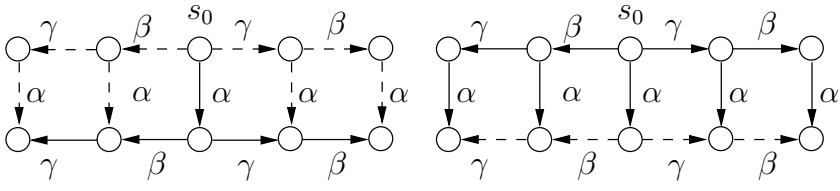
The literature distinguishes between partial-order reduction based on *persistent sets* and reduction based on *sleep sets*. The two approaches are orthogonal and achieve better results when combined. Both techniques compute a subset of the runnable transitions for each visited state and restrict future exploration to transitions in this set.

We denote the set of states and the set of processes of a SystemC model by S and θ, respectively. We denote the set of enabled (runnable) processes (transitions) in a state s by $Enabled(s)$, i.e., *Enabled* is a mapping from S to $\mathscr{P}(\theta)$. Processes are relations between states. We write $s \xrightarrow{\alpha} t$ to denote that the state changes from s to t by executing process α.

Definition 1. *[22] 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:*

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

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.



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

Fig. 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.

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 the guard $\{s\}$ [25].

Definition 2. *[25] 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.*

Similar to [21], Scoot uses a data-flow analysis in order to compute an over-approximating dependency relation.

Definition 3. *[25] Let (S, S_0, θ) be a transition system, and $s_0 \in S$ denote one of its states. A set of transitions $T \subset Enabled(s_0)$ 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.*

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

Figure 1.1 illustrates the effects of the persistent-set technique. In state s_0, the exploration uses the persistent set $T = \{\alpha\}$ to avoid visiting some of the states. In contrast, the sleep-set technique maintains a set of runnable transitions that can be skipped during the exploration (the sleep set). The method is concerned with branching information from the *past*. Figure 1.2 shows a typical exploration using sleep sets. Unlike the previous approach, the sleep-set technique only reduces the number of explored transitions and has no effect on the number of explored states. The exploration backtracks early when the sleep set contains all runnable transitions.

3 Implementation

3.1 Overview of Scoot

Figure 2 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

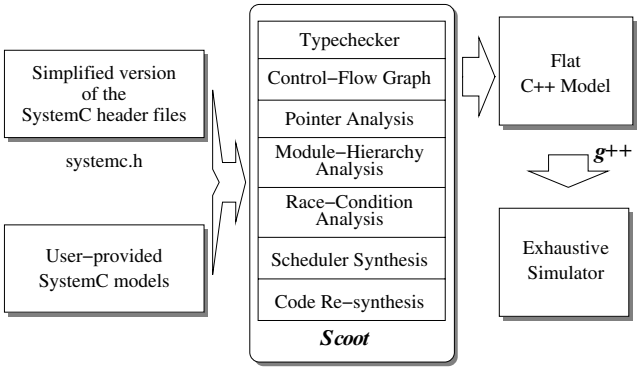


Fig. 2. Overview of Scoot

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 the computation of race conditions for each pair of processes, which is explained in Sec. 3.3. 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, 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 [26].

3.2 A Scheduler with Partial-Order Reduction

Algorithm 1 is Scoot's implementation of the evaluation phase. In contrast to the related work, *evaluation_phase* schedules runnable processes using information *statically* collected to reduce the number of interleavings explored. We are not aware of tools that compute equally strong conditions statically.

The evaluation phase terminates once the set of runnable processes is empty. The algorithm performs partial-order reduction using persistent sets and sleep sets, and is a variation of techniques presented by Godefroid [25]. On line 3, the procedure calls the function *runnable*() to check if the set of runnable processes is empty before proceeding to the next iteration.

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

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

At simulation time, the scheduler calls *get_pers* to compute the set *persistents* of persistent processes. The subsequent part of the algorithm uses the set *sleeps*, declared outside the main loop on line 2, to perform partial-order reduction. On line 5, the set *awakes* consists of the persistent processes *not* in *sleeps*. If the set of awaken processes is empty (line 6), then other traces are covering all subsequent behaviors, and therefore, the simulator stops the execution. Otherwise, the scheduler computes the sleep sets for the next iteration using the map *next_sleeps*, which maps processes to a set of processes (lines 7–14). One line 10, the call to *commutative* returns *true* if the processes p_i and p_j are commutative in the current state. The scheduler reduces the computation of conditional independence to the computation of commutativity conditions by considering that all the processes are always enabled – if $\rho \notin Enabled(s)$, then this is interpreted as $s \xrightarrow{\rho} s$. This way, two processes are independent in the current state if and only if they are commutative in this state. SCOOT relies on Model Checking to compute a conservative condition that guarantees commutativity of the processes in the current state; the details of this pre-computation are presented in the following subsection. In contrast, traditional approaches need to rely on either executing the processes to determine which transitions are independent in the current state, which adds overhead, or on an imprecise data-flow analysis.

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

3.3 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. In general, SystemC processes need not terminate, and thus computing the strongest possible commutativity condition for a given pair of processes p_1 and p_2 is undecidable. We compute a conservative approximation by applying a Model Checker to the harness given as Program 2.

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 then iteratively strengthened

```
   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);
```

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

SCOOT passes the harness to a Model Checker to check the reachability of the last line, which is modeled by means of an assertion. If the Model Checker returns a counterexample, we have a trace π with an initial state satisfying the initial condition ϕ, passing through both processes, and ending in a state that violates the assertion. The path therefore begins in a state in which the two processes are commutative. SCOOT then computes the weakest precondition of $s_{1,2} = s_{2,1}$ alongside that path. Let P_π denote this condition. The executions of $p_1(); p_2()$ and $p_2(); p_1()$ from a state s terminate and yield an equal state if s satisfies P_π. Consequently, P_π is an under-approximation of the commutativity condition for p_1 and p_2. At this point, SCOOT strengthens ϕ using $\neg P_\pi$, yielding ϕ'. This removes the trace π and any trace similar to π that goes through the same control locations. SCOOT iterates this process until the Model Checker stops reporting counterexamples. At this point, the predicate $P = \bigvee_\pi P_\pi$ represents the weakest condition such that the executions of $p_1(); p_2()$ and $p_2(); p_1()$ terminate and that p_1 and p_2 are commutative.

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.

In the following, we elaborate on our integration of the strengthening loop into SatAbs, a Model Checker based on predicate abstraction. Note that our approach is independent of the particular Model Checking engine. The general idea can be extended in different directions. As an example, we can adapt the strengthening loop to operate on infinite traces using a Model Checker for liveness properties such as Terminator [27], or we can replace the Model Checker with a testing engine to discover terminating traces at the cost of code-coverage guarantees.

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 [14,15]. 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. If the counterexample can be simulated on the original program, it is reported to the user. The counterexample is called *spurious* if it does not correspond to a concrete trace. In that case, a refinement procedure adds new predicates in a way that removes the spurious trace. This is automated by *Counterexample Guided Abstraction Refinement* (CEGAR) [28] and promoted by the Model Checker Slam [13]. Predicate abstraction has been applied to SpecC [29] and SystemC [30]. Figure 3 shows the integration of our technique into SatAbs. After strengthening, SatAbs retains the abstract model obtained during previous iterations.

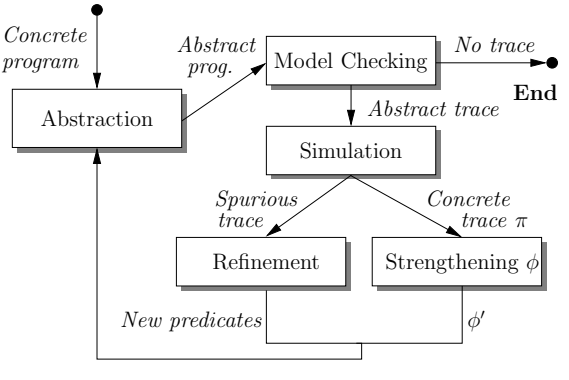


Fig. 3. Iterative computation of the process commutativity condition using predicate abstraction

4 Experimental Evaluation

In this section, we evaluate the benefits of integrating our partial-order reduction into a simulator that examines all schedules exhaustively using a backtracking search. 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 our results on a 3GHz Linux machine. We make the benchmarks and the tool available for experimentation by other researchers at *www.cprover.org/scoot/*.

4.1 The Running Example

We continue our running example (Program 1). Figure 4 depicts the number of explored transitions as a function of the number of simulation steps using persistent and sleep sets *(P+S)* and without partial-order reduction *(No-POR)*. We set *PMAX* to *10*. Our simulator performs a state-less search, that is, the simulator replays transitions to backtrack. Those transitions are counted only once. With this technique, the number of transitions explored during simulation grows quadratically with the number of steps, whereas without partial-order reduction, the curve grows exponentially. As mentioned before, a conventional dynamic analysis would always detect a read/write dependency between the two processes, forcing the simulator to explore all schedules.

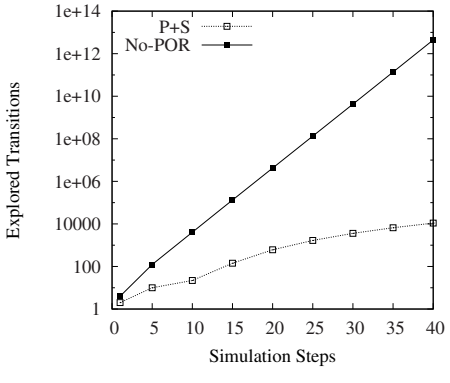


Fig. 4. Number of transitions explored at runtime as a function of the number of simulation steps

4.2 State Machines

We use two different benchmarks to evaluate the benefit of statically computed race conditions. 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 3 contains the skeleton of the benchmark. When triggered, the clients and the server execute

Program 3. Skeleton of Benchmark B1

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

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 5 compares the number of explored transitions, and the total exploration time as a function of the number of simulation steps. We present results without partial-order reduction (*No-POR*) and using a combination of sleep sets and persistent sets (*P+S*). The exploration time is limited to thirty minutes (1800 seconds).

The results indicate that partial-order reduction using statically computed commutativity conditions is able to significantly reduce both the number of explored transitions and the exploration time by about two to three orders of magnitude. With partial-order reduction, the simulator can exhaustively cover all the relevant behaviors up to twelve simulation steps in less than thirty minutes, whereas the naive approach already times out after seven simulation steps.

Our second benchmark (B2) consists of two synchronous state machines communicating via shared variables. The model has three interdependent processes,

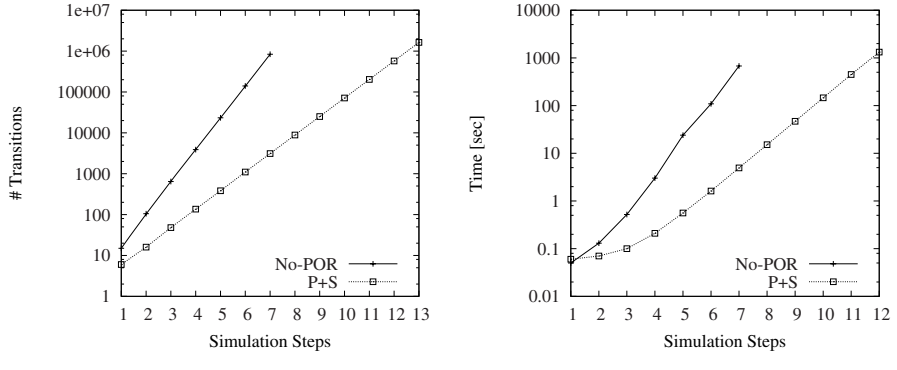


Fig. 5. Performance effect of static partial-order reduction on B1

Table 1. Time to compute the race conditions for each of the process-pairs using SatAbs and CBMC. The timeout is set to ten minutes.

Benchmark	Pair	SatAbs [s]	CBMC [s]
B1	0	< 1	< 1
B1	1	3	< 1
B1	2	3	< 1
B2	0	76	TO
B2	1	19	5
B2	2	19	2

which are sensitive to the clock. The state machines are implemented using case switches. On this benchmark, partial-order reduction reduces the simulation time and the number of explored transition by one order of one magnitude.

For each pair of processes, Table 1 shows the time required for the static analysis running SatAbs and CBMC. The cost for B1 is negligible using both SatAbs and CBMC. The results for B2 indicate that CBMC is faster than SatAbs on the second and third pair of processes but times out on the first one, 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 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.

5 Conclusion

We presented Scoot, a novel compiler for SystemC that integrates static analysis and formal verification techniques in order to improve simulation performance. We 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.

References

1. Blanc, N., Kroening, D.: Race analysis for SystemC using model checking. In: Proceedings of ICCAD 2008, pp. 356–363. IEEE, Los Alamitos (2008)
2. Blanc, N., Kroening, D., Sharygina, N.: Scoot: A tool for the analysis of SystemC models. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 467–470. Springer, Heidelberg (2008)
3. Clarke, E., Kroening, D., Sharygina, N., Yorav, K.: SATABS: SAT-based predicate abstraction for ANSI-C. In: Halbwachs, N., Zuck, L.D. (eds.) TACAS 2005. LNCS, vol. 3440, pp. 570–574. Springer, Heidelberg (2005)
4. Netzer, R.H.B., Miller, B.P.: What are race conditions? Some issues and formalizations. ACM Lett. Program. Lang. Syst. 1, 74–88 (1992)
5. Flanagan, C., Freund, S.N.: Type-based race detection for Java. In: Programming language design and implementation (PLDI), pp. 219–232. ACM, New York (2000)
6. Savage, S., Burrows, M., Nelson, G., Sobalvarro, P., Anderson, T.: Eraser: A dynamic data race detector for multithreaded programs. ACM Trans. Comput. Syst. 15, 391–411 (1997)
7. Engler, D., Ashcraft, K.: RacerX: Effective, static detection of race conditions and deadlocks. In: Operating systems principles (SOSP), pp. 237–252. ACM, New York (2003)
8. Naik, M., Aiken, A., Whaley, J.: Effective static race detection for Java. In: Programming language design and implementation (PLDI), pp. 308–319. ACM, New York (2006)
9. D'Silva, V., Kroening, D., Weissenbacher, G.: A survey of automated techniques for formal software verification. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD) 27, 1165–1178 (2008)
10. Vardi, M.Y.: Formal techniques for SystemC verification. In: Design Automation Conference (DAC), pp. 188–192. ACM, New York (2007)
11. Witkowski, T., Blanc, N., Kroening, D., Weissenbacher, G.: Model checking concurrent Linux device drivers. In: Automated software engineering (ASE), pp. 501–504. ACM, New York (2007)
12. Qadeer, S., Wu, D.: KISS: keep it simple and sequential. SIGPLAN Not. 39, 14–24 (2004)
13. Ball, T., Rajamani, S.K.: The SLAM project: debugging system software via static analysis. In: POPL 2002: Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, pp. 1–3. ACM, New York (2002)
14. Graf, S., Saïdi, H.: Construction of abstract state graphs with PVS. In: Grumberg, O. (ed.) CAV 1997. LNCS, vol. 1254, pp. 72–83. Springer, Heidelberg (1997)
15. Ball, T., Rajamani, S.: Boolean programs: A model and process for software analysis. Technical Report MSR-TR-2000-14, Microsoft Research (2000)
16. Godefroid, P.: Software model checking: The VeriSoft approach. Form. Methods Syst. Des. 26, 77–101 (2005)
17. Sen, A., Ogale, V., Abadir, M.S.: Predictive runtime verification of multi-processor SoCs in SystemC. In: Design Automation Conference (DAC), pp. 948–953. ACM, New York (2008)
18. Flanagan, C., Godefroid, P.: Dynamic partial-order reduction for model checking software. In: Principles of programming languages (POPL), pp. 110–121. ACM, New York (2005)
19. Lamport, L.: Time, clocks, and the ordering of events in a distributed system. Commun. ACM 21, 558–565 (1978)

20. Helmstetter, C., Maraninchi, F., Maillet-Contoz, L., Moy, M.: Automatic generation of schedulings for improving the test coverage of systems-on-a-chip. In: Formal Methods in Computer Aided Design (FMCAD), pp. 171–178. IEEE Computer Society, Los Alamitos (2006)
21. Kundu, S., Ganai, M., Gupta, R.: Partial order reduction for scalable testing of SystemC TLM designs. In: Design Automation Conference (DAC), pp. 936–941. ACM, New York (2008)
22. Wang, C., Yang, Z., Kahlon, V., Gupta, A.: Peephole partial order reduction. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 382–396. Springer, Heidelberg (2008)
23. Peled, D.: All from one, one for all: On model checking using representatives. In: Courcoubetis, C. (ed.) CAV 1993. LNCS, vol. 697, pp. 409–423. Springer, Heidelberg (1993)
24. Peled, D.: Combining partial order reductions with on-the-fly model-checking. In: Dill, D.L. (ed.) CAV 1994. LNCS, vol. 818, pp. 377–390. Springer, Heidelberg (1994)
25. Godefroid, P.: Partial-Order Methods for the Verification of Concurrent Systems. LNCS. Springer, Heidelberg (1996)
26. Blanc, N., Groce, A., Kroening, D.: Verifying C++ with STL containers via predicate abstraction. In: 22nd IEEE International Conference on Automated Software Engineering (ASE), pp. 521–524. IEEE, Los Alamitos (2007)
27. Cook, B., Podelski, A., Rybalchenko, A.: Terminator: Beyond safety. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 415–418. Springer, Heidelberg (2006)
28. Clarke, E.M., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement. In: Emerson, E.A., Sistla, A.P. (eds.) CAV 2000. LNCS, vol. 1855, pp. 154–169. Springer, Heidelberg (2000)
29. Clarke, E., Jain, H., Kroening, D.: Verification of SpecC using predicate abstraction. Form. Methods Syst. Des. 30, 5–28 (2007)
30. Kroening, D., Sharygina, N.: Formal verification of SystemC by automatic hardware/software partitioning. In: Formal Methods and Models for Co-Design (MEMOCODE), pp. 101–110. IEEE Computer Society, Los Alamitos (2005)