Boosting Lazy Abstraction for SystemC with Partial Order Reduction

Alessandro Cimatti, Iman Narasamdya, and Marco Roveri

Fondazione Bruno Kessler — Irst
{cimatti,narasamdya,roveri}@fbk.eu

Abstract. The SystemC language is a de-facto standard for the description of systems on chip. A promising technique, called ESST, has recently been proposed for the formal verification of SystemC designs. ESST combines *Explicit* state techniques to deal with the SystemC *Scheduler*, with *Symbolic* techniques, based on lazy abstraction, to deal with the *Threads*. Despite its relative effectiveness, this approach suffers from the potential explosion of thread interleavings.
In this paper we propose the adoption of partial order reduction (POR) techniques to alleviate the problem. We extend ESST with two complementary POR techniques (persistent set, and sleep set), and we prove the soundness of the approach in the case of safety properties. The extension is only seemingly trivial: the POR, applied to the scheduler, must be proved not to interfere with the lazy abstraction of the threads.
We implemented the techniques within the software model checker KRATOS, and we carried out an experimental evaluation on benchmarks taken from the SystemC distribution and from the literature. The results showed a significant improvement in terms of the number of visited abstract states and run times.

1 Introduction

SystemC is widely used for the design of systems on chip. Executable models written in SystemC are amenable for high-speed simulation before synthesizing the RTL hardware description. Formal verification of SystemC designs can help to pinpoint errors, preventing their propagation down to the hardware, but can also help to reveal errors in the specifications.

Despite its importance, however, formal verification of SystemC is a very hard challenge. Indeed, a SystemC design is a very complex entity. In addition to rich data, SystemC features a form of multi-threading, where scheduling is cooperative and carried out according to a specific set of rules [20], and the execution of threads is mutually exclusive.

A promising technique, called ESST [7], has recently been proposed for the verification of SystemC designs. ESST combines *Explicit* state techniques to deal with the SystemC *Scheduler*, with *Symbolic* techniques, based on lazy abstraction [2], to deal with the *Threads*. Despite its relative effectiveness, this technique requires the exploration of a large number of thread interleavings, many of which are redundant, with subsequent degradations in the run time performance and high memory consumption.

Partial-order reduction (POR) [11, 18, 22] is a well known model checking technique that tackles the state explosion problem by exploring only representative subset of all possible schedules. In general, POR exploits the commutativity of concurrent transitions that result in the same state when they are executed in different orders. POR

techniques have successfully been integrated in explicit-state software model checkers like SPIN [13] and VERISOFT [10], and also applied in symbolic model checking [15, 23, 1].

In this paper we boost ESST with two complementary POR techniques [11], *persistent set* and *sleep set*. The POR techniques are used in the ESST algorithm to limit the expansion of the transitions in the explicit scheduler, while the nature of the symbolic search of the threads, based on lazy abstraction, remains unchanged. Notice that the application of POR in ESST algorithm is only seemingly trivial, because POR could in principle interact negatively with the lazy abstraction used for the search within the threads. In fact, we prove that the pruning carried out by POR in the abstract space preserves the reachability in the concrete space, which yields the soundness of the approach in the case of safety properties.

We implemented these POR techniques within the KRATOS software model checker. KRATOS implements the ESST algorithm, and is at the core of the tool chain described in [7], which also includes a SystemC front-end derived from PINAPA [17]. We perform an experimental evaluation on the benchmark set used in [7], that includes problems from the SystemC distribution and from the literature. The results show that POR techniques can yield substantial improvements on the performance of the ESST algorithm in terms of the number of visited abstract states and run times.

This paper is structured as follows. In Sec. 2 we briefly introduce SystemC and we briefly describe the ESST algorithm. In Sec. 3 we show by means of an example the possible state explosion problem that may arise. In Sec. 4 we show how to lift POR techniques to the ESST algorithm. In Sec. 5 we revise the related work. Finally, in Sec. 7 we draw some conclusions and we outline future work.

2 Background

The SystemC language. SystemC is a C++ library that consists of (1) a core language that allows one to model a System-on-Chip (SoC) by specifying its components and architecture, and (2) a simulation kernel (or scheduler) that schedules and runs processes (or threads) of components. SoC components are modeled as SystemC modules that communicate through channels (that are bound to the ports specified in the modules).

A module consists of one or more threads that describe the parallel behavior of the SoC design. SystemC provides general-purpose events as a synchronization mechanism between threads. For example, a thread can suspend itself by waiting for an event or by waiting for some specified time. A thread can perform immediate notification of an event or delayed notification.

The SystemC scheduler is a cooperative non-preempting scheduler that runs at most one thread at a time. During a simulation, the status of a thread changes from sleeping, to runnable, and to running. A running thread will only give control back to the scheduler by suspending itself. The scheduler runs all runnable threads, one at a time, in a single delta cycle, while postponing the channel updates made by the threads. When there are no more runnable threads, the scheduler materializes the channel updates, and wakes up all sleeping threads that are sensitive to the updated channels. If, after this step, there are some runnable threads, then the scheduler moves to the next delta cycle.

```
1  SC_MODULE( numgen ) {
2   sc_out<int> o;  // output port.
3   sc_in<bool> ck; // input port for clock.
4
5   // Reads input from environment.
6   void gen() { int x = read_input(); o.write(x); }
7
8   SC_CTOR( numgen ) {
9    // declare "gen" as a method thread.
10   SC_METHOD( gen );
11   dont_initialize();
12   sensitive << ck.pos();
13  }
14 }
15
16 SC_MODULE( stage1 ) {
17  sc_in<int>  i;  // input port.
18  sc_out<int> o;  // output port.
19  sc_in<bool> ck; // input port for clock.
20
21  // Pass value from input port to output port.
22  void pass() { int x = i.read(); o.write(x); }
23
24  SC_CTOR( stage1 ) {
25   // declare "pass" as a method thread
26   SC_METHOD( pass );
27   dont_initialize();
28   sensitive << ck.pos();
29  }
30 }
```

```
31 SC_MODULE( stage2 ) {
32  sc_in<int>   i; // input port.
33  sc_in<bool> ck; // input port for clock.
34
35  // Method for checking value.
36  void check() { int x = i.read(); assert(x == 0); }
37
38  SC_CTOR( stage2 ) {
39   // declare "check" as a method thread
40   SC_METHOD( check );
41   dont_initialize();
42   sensitive << ck.pos();
43  }
44 }
45
46 int sc_main() {
47  sc_signal<int> gen_to_s1, s1_to_s2;
48  sc_signal<bool> ck;
49
50  numgen N; N.o(gen_to_s1); N.ck(ck);
51
52  stage1 S1; S1.i(gen_to_s1); S1.o(s1_to_s2); S1.ck(ck);
53
54  stage2 S2; S2.i(s1_to_s2); S2.ck(ck);
55
56  sc_start(0);
57  for (int i=0; i < 3; ++i) {
58   ck.write(1); sc_start(0); ck.write(0); sc_start(0);
59  }
60 }
```

Fig. 1. Example of SystemC design.

Otherwise, it accelerates the simulation time to the nearest time point where a sleeping thread or an event can be woken up. The scheduler quits when there are no more runnable threads after time acceleration.

An example of SystemC design is shown in Fig. 1. It consists of three modules: `numgen`, `stage1`, and `stage2`. In the `sc_main` function we create an instance for each module, and we connect them such that the instances of `numgen` and `stage1` are connected by the signal `gen_to_s1` and the instances of `stage1` and `stage2` are connected by the signal `s1_to_s2`. The thread `gen` of `numgen` reads an integer value from the environment and sends it to `stage1` through the signal `gen_to_s1`. The thread `pass` of `stage1` simply reads the value from the signal `gen_to_s1` and sends it to `stage2` through the signal `s1_to_s2`. The thread `check` of `stage2` reads the value from `s1_to_s2` and asserts that the read value equals 0. All threads are made sensitive to the positive edge of the clock `ck` (modeled as a boolean signal): they become runnable when the value of the `ck` changes from 0 to 1. The function `dont_initialize` makes the most-recently declared thread sleep initially. The clock cycle is controlled by the loop in the `sc_main` function. The function `sc_start` runs a simulation until there are no more runnable threads. The property that we want to check is that the value read by `numgen` from the environment reaches `stage2` in three clock cycle.

The Explicit Scheduler + Symbolic Threads (ESST) approach. The ESST technique [7] is a counter-example guided abstraction refinement [8] based technique that combines explicit-state technique with lazy predicate abstraction [2]. In the same way as the classical lazy abstraction, the data path of the threads is analyzed by means of predicate abstraction, while the flow of control of each thread and the state of the scheduler are analyzed with explicit-state techniques.

We assume that the SystemC design has been translated into a threaded C program [7] in which each SystemC thread is represented by a C function. Each function corresponding to a thread is represented by a *control-flow automaton* (CFA), which is

a pair (L, G), where L is the set of control locations and $G \subseteq L \times Ops \times$ is the set of edges such that each edge is labelled by an operation from the set Ops of operations.

Threads in a threaded C program communicate with each other by means of shared global variables, and use primitive functions and events as synchronization mechanism. For SystemC we have the following primitive functions: `wait_event(`e`)`,`wait_time(`t`)`, `notify_event(`e`)`, `notify_event_at_time(`e`,`t`)`, and `cancel_event(`e`)` for an event e and a time unit t.

The ESST algorithm is based on the construction and analysis of an *abstract reachability forest* (ARF) that describes the reachable abstract states of the threaded program. An ARF consists of connected *abstract reachability trees* (ART's), each of which is obtained by unwinding the CFA corresponding to the running thread.

For a threaded program with n threads, a *node* in an ARF is a tuple $(\langle l_1, \varphi_1 \rangle, \ldots, \langle l_n, \varphi_n \rangle, \varphi, S)$, where l_i and φ_i are, respectively, the program location and the region of thread i, φ is the region of global variables, and S is the state of the scheduler. Regions are formulas describing the values of variables. Information maintained in the scheduler state S includes the status of threads and events, the events that sleeping threads are waiting for their notifications, and the delays of event notifications.

An ARF is constructed by unwinding the CFA's of threads, and by executing the scheduler. Unwinding a CFA involves computing the *abstract strongest post-condition* $SP^{\pi}(\varphi, op)$ of a region φ with respect to the operation op labelling the unwound CFA edge and the precision π. In the ESST approach, the precision π can contain a set of predicates that are tracked for the global region and for the thread regions.

Unwinding the CFA by executing primitive functions is performed by the function SEXEC that takes as inputs a scheduler state and a call to a primitive function p, and returns the updated scheduler state obtained from executing p. For example, the state $S' = \text{SEXEC}(S, \texttt{wait\_event(e)})$ is obtained from the state S by changing the status of the running thread to sleep, and noting that the now sleeping thread is waiting for the event e.

We implement the scheduler by the function SCHED that, given a scheduler state where all threads are sleeping, outputs a set of scheduler states representing all possible schedules such that each of the output scheduler states has exactly one running thread.

We expand a node $(\langle l_1, \varphi_1 \rangle, \ldots, \langle l_n, \varphi_n \rangle, \varphi, S)$ by means of the following rules [7]:

E1. If there is a running thread i in S such that the thread performs an operation op and (l_i, op, l_i') is an edge of the CFA of thread i, then
   - If op is *not* a call to primitive function, then the successor node is $(\langle l_1, \varphi_1' \rangle, \ldots, \langle l_i', \varphi_i' \rangle, \ldots, \langle l_n, \varphi_n' \rangle, \varphi', S)$, where $\varphi_i' = SP^{\pi}(\varphi_i \wedge \varphi, op)$, $\varphi_j' = SP^{\pi}(\varphi_j \wedge \varphi, \text{HAVOC}(op)))$ for $j \neq i$, and $\varphi' = SP^{\pi}(\varphi, op)$. The function HAVOC collects all global variables possibly updated by op, and builds a new operation where these variables are assigned with fresh variables.
   - If op is a primitive function, then the successor node is $(\langle l_1, \varphi_1 \rangle, \ldots, \langle l_i', \varphi_i \rangle, \ldots, \langle l_n, \varphi_n \rangle, \varphi, \text{SEXEC}(S, op))$.

E2. If there is no running thread in S, then, for each $S' \in \text{SCHED}(S)$, we create a successor node $(\langle l_1, \varphi_1 \rangle, \ldots, \langle l_n, \varphi_n \rangle, \varphi, S')$, such that the node becomes the root node of a new ART that is then added to the ARF. Such a connection between two nodes is called *ARF connector*.

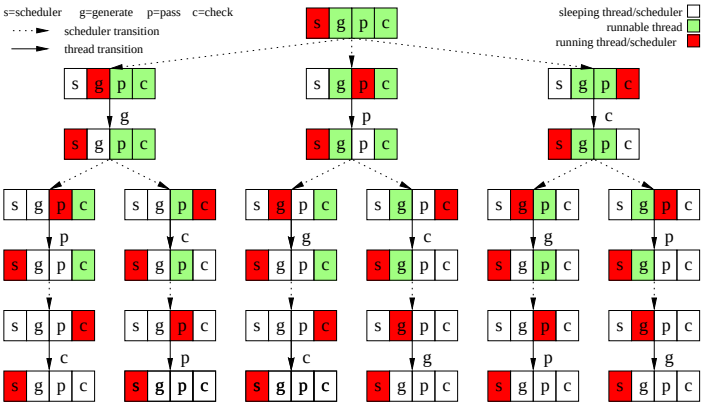

Fig. 2. The 6 possible interleavings for threads of the example in Fig. 1.

In the construction of an ARF, one stops expanding a node if the node is covered by other nodes or if the conjunction of all its thread regions and the global region is unsatisfiable. We say that a node $(\langle l_1, \varphi_1\rangle, \ldots, \langle l_n, \varphi_n\rangle, \varphi, S)$ is *covered* by a node $(\langle l'_1, \varphi'_1\rangle, \ldots, \langle l'_n, \varphi'_n\rangle, \varphi', S')$ if (1) $l_i = l'_i$ for $i = 1, \ldots, n$, (2) $S = S'$, and (3) $\varphi \Rightarrow \varphi'$ and $\bigwedge_{i=1,\ldots,n}(\varphi_i \Rightarrow \varphi'_i)$ are valid. An ARF is *complete* if it is closed under the expansion of the above rules. An ARF is *safe* if it is complete and, for every node $(\langle l_1, \varphi_1\rangle, \ldots, \langle l_n, \varphi_n\rangle, \varphi, S)$ in the ARF such that $\varphi \wedge \bigwedge_{i=1,\ldots,n} \varphi_i$ is satisfiable, none of the locations $l_1, \ldots, l_n$ are error locations.

The construction of an ARF starts with a single ART representing reachable states of the main function. In the root node of that ART all regions are initialized with $True$, all thread locations are set to the entries of the corresponding threads and the only running thread in the scheduler state is the main function. The main function then suspends itself by calling a primitive function that starts the simulation. We expand the ARF using the rules E1 and E2 until either the ARF is complete or we reach a node where one of the thread's location is an error location. In the latter case we build a counterexample consisting of paths in the trees of the ARF and check if the counterexample is feasible. If it is feasible, then we have found a real counterexample witnessing that the program is unsafe. Otherwise, we use the spurious counterexample to discover predicates to refine the ARF. We refer to [7] for further details.

3 The problem of multiple interleavings

The ESST algorithm often has to explore a large number of possible schedules. However, some of them might be redundant because the order of interleavings of some threads is irrelevant.

Fig. 2 depicts, for the example in Fig. 1, the ARF constructed by the ESST algorithm in every delta cycle. This figure clearly shows that there are 6=3! possible schedules. These threads communicate with each other using signals. A signal s in SystemC can be viewed as a pair (s_{old}, s_{new}) of variables such that writing to s is modeled by writing to s_{new} while reading from s is modeled by reading from s_{old}. Thus, these threads access disjoint sets of variables: `gen` writes to $\texttt{gen\_to\_s1}_{new}$, `pass` reads from

$\texttt{gen\_to\_s1}_{old}$ and writes to $\texttt{s1\_to\_s2}_{new}$, and `check` reads from $\texttt{s1\_to\_s2}_{old}$. Therefore, when all the threads become runnable, the order of running them is irrelevant. Consequently, instead of exploring all 3! possible schedules, it is sufficient to explore only one of them.

Partial order reduction techniques (POR) [11, 18, 22] can be used to avoid exploring redundant schedules. However, we need to ensure that in the construction of the ARF the partial order reduction does not remove all counter-example paths witnessing that the SystemC design is unsafe. In the following we will see how to extend the the ESST algorithm to exploit POR in the construction of the ARF guaranteeing that the above condition is satisfied.

4 Reduction Algorithms in ESST

Partial-order reduction (POR) [11, 18, 22] is a model checking technique that is aimed at combating the state explosion by exploring only representative subset of all possible interleavings. In this paper we apply POR to the ESST technique.

4.1 Partial-Order Reduction Techniques

For presentation of POR in this section we follow the standard notions and notations used in other literature [11, 9]. We represent a concurrent program as a transition system $M = (S, S_0, T)$, where S is the finite set of states, $S_0 \subset S$ is the set of initial states, and T is a set of transitions such that for each $\alpha \in T$, we have $\alpha \subset S \times S$. We say that $\alpha(s, s')$ holds and often writes it as $s \xrightarrow{\alpha} s'$ if $(s, s') \in \alpha$. A state s' is a successor of a state s if $s \xrightarrow{\alpha} s'$ for some transition $\alpha \in T$. In the following we will only consider deterministic transitions A transition α is *enabled* in a state s if there is a state s' such that $\alpha(s, s')$ holds. The set of transitions enabled in a state s is denoted by $enabled(s)$. A *path* from a state s in a transition system is a finite or infinite sequence $s_0 \xrightarrow{\alpha_0} s_1 \xrightarrow{\alpha_1} \cdots$ such that $s = s_0$ and $s_i \xrightarrow{\alpha_i} s_{i+1}$ for all i. A path is empty if the sequence consists only of a single state. The length of a finite path is the number of transitions in the path.

Let $M = (S, S_0, T)$ be a transition system, we denote by $Reach(S_0, T) \subseteq S$ the set of states reachable from the states in S_0 by the transitions in T. In this work we are interested in verifying safety properties in the form of program assertion. To this end, we assume that there is a set $T_{err} \subseteq T$ of *error transitions* such that the set

$$E_{M,T_{err}} = \{s \in S \mid \exists s' \in S. \exists \alpha \in T_{err}.\ \alpha(s', s) \text{ holds } \}$$

is the set of *error states* of M with respect to T_{err}. A transition system M is *safe with respect to the set $T_{err} \subseteq T$ of error transitions* iff $Reach(S_0, T) \cap E_{M,T_{err}} = \emptyset$.

Selective search in POR exploits the commutativity of concurrent transitions. The concept of commutativity of concurrent transitions can be formulated by defining an independence relation on pairs of transitions.

DEFINITION 4.1 (INDEPENDENCE RELATION, INDEPENDENT TRANSITIONS) An *independence relation* $I \subseteq T \times T$ is a symmetric, anti-reflexive relation such that for each state $s \in S$ and for each $(\alpha, \beta) \in I$ the following conditions are satisfied: (***Enabledness***) If α is in $enabled(s)$, then β is in $enabled(s)$ iff β is in $enabled(\alpha(s))$. (***Commutativity***) If α and β are in $enabled(s)$, then $\alpha(\beta(s)) = \beta(\alpha(s))$.

We say that two transitions α and β are *independent* of each other if for every state s they satisfy the enabledness and commutativity conditions. We also say that two transitions α and β are *independent in a state* s of each other if they satisfy the enabledness and commutativity conditions in s. □

In the sequel we will use the notion of valid dependence relation to select a representative subset of transitions that need to be explored.

DEFINITION 4.2 (VALID DEPENDENCE RELATION) A *valid dependence relation* $D \subseteq T \times T$ is a symmetric, reflexive relation such that for every $(\alpha, \beta) \notin D$, the transitions α and β are independent of each other. □

The Persistent Set approach. To reduce the number of possible interleavings, in every state visited during the state space exploration one only explores a representative subset of transitions that are enabled in that state. However, to select such a subset we have to avoid possible dependencies that can happen in the future. To this end, we appeal to the notion of persistent set [11].

DEFINITION 4.3 (PERSISTENT SET) A set $P \subseteq T$ of enabled transitions in a state s is *persistent* in s if for every finite nonempty path $s = s_0 \xrightarrow{\alpha_0} s_1 \cdots s_n \xrightarrow{\alpha_n} s_{n+1}$ such that $\alpha_i \notin P$ for all $i = 0, \ldots, n$, we have α_n independent of any transition in P in s_n. □

Note that the persistent set in a state is not unique. To guarantee the existence of successor state, we impose the *successor-state* condition on the persistent set: the persistent set in s is empty iff so is $enabled(s)$. For simplicity, in the sequel whenever we speak about persistent sets, then the sets satisfy the successor-state condition. We say that a state s is *fully expanded* if the persistent set in s equals $enabled(s)$. It is easy to see that, for any transition α not in the persistent set P in a state s, the transition α is disabled in s or independent of any transition in P.

We denote by $Reach_{red}(S_0, T) \subseteq S$ the set of states reachable from the states in S_0 by the transitions in T such that, during the state space exploration, in every visited state we only explore the transitions in the persistent set in that state. It is easy to see that $Reach_{red}(S_0, T) \subseteq Reach(S_0, T)$.

To preserve safety properties of a transition system we need to guarantee that the reduction by means of persistent set does not remove all interleavings that lead to an error state. To this end, we impose the so-called *cycle condition* on $Reach_{red}(S_0, T)$ [9, 18]: a cycle is not allowed if it contains a state in which a transition α is enabled, but α is never included in the persistent set of any state s on the cycle.

THEOREM 4.4 *A transition system $M = (S, S_0, T)$ is safe w.r.t. a set $T_{err} \subseteq T$ of error transitions iff $Reach_{red}(S_0, T)$ that satisfies the cycle condition does not contains any error state from $E_{M, T_{err}}$.* □

The Sleep Set approach. We consider also the *sleep set* POR technique. This technique exploits independencies of enabled transitions in the current state. For example, suppose that in some state s there are two enabled transitions α and β, and they are independent of each other. Suppose further that the search explores α first from s. Then, when the search explores β from s such that $s \xrightarrow{\beta} s'$ for some state s', we associate with s' a sleep set containing only α. From s' the search only explores transitions that

are not in the sleep set of s'. That is, although the transition α is still enabled in s', it will not be explored. Both persistent set and sleep set techniques are orthogonal and complementary, and thus can be applied simultaneously.

Note that the sleep set technique only removes transitions, and not states. Thus Theorem 4.4 still holds when the sleep set technique is applied.

4.2 Applying POR to ESST

Applying POR to the ESST algorithm is not trivial. The ESST algorithm is based on the construction of an ARF that describes the reachable abstract states of the threaded program, while the description of POR in Sec. 4.1 is based on the analysis of reachable concrete states. One then needs to guarantee that the original ARF is safe iff. the reduced ARF, obtained by applying POR in the construction of ARF, is safe. That is, we have to ensure that the selective search performed during the construction of ARF does not remove all non-spurious paths that lead to error locations. In particular, the construction of reduced ARF has to check if the cycle condition is satisfied in its concretization.

To integrate POR techniques into the ESST algorithm we first need to identify transitions in the threaded program. In the above description of POR the execution of a transition is atomic. We introduce the notion of atomic block as the notion of transition in the threaded program. Intuitively an atomic block is a block of operations between calls to primitive functions that can suspend the thread. For simplicity, let us call such primitive functions *wait functions*.

An *atomic block* of a thread is a rooted sub-graph of the CFA satisfying the following conditions: (1) its unique entry is the entry of the CFA or the location that immediately follows a call to a wait function; (2) its exit is the exit of the CFA or the location that immediately follows a call to a wait function; and (3) there is no call to a wait function in any CFA path from the entry to an exit except the one that precedes the exit. Note that an atomic block has a unique entry, but can have multiple exits. We often identify an atomic block by its entry.

For example, consider the thread code of Fig. 3. One atomic block starts from the entry of the thread and ends at the label `lab` and at the exit of the thread. The other atomic block starts from the label `lab` and ends at the label `lab` too and at the exit of the thread. In the sequel we will use the terms transition and atomic block interchangeably.

```
void thread_t() {
  while (...) {
    ...
    wait_event(e);
    lab: ...
  }
}
```

Fig. 3: Fragment of code.

We use static analysis techniques to compute a valid dependence relation. In particular, a pair (α, β) of atomic blocks are in a valid dependence relation if one of the following criteria is satisfied: (1) The atomic block α contains a write to a shared (or global) variable g, and the atomic block β contains a write or a read to g. (2) The atomic block α contains an immediate notification of an event e, and the atomic block β contains a wait for e. (3) The atomic block α contains a delayed notification of an event e, and the atomic block β contains a cancellation of a notification of e.

Persistent sets are computed using a valid dependence relation. Let D be a valid dependence relation. Algorithm 1 computes persistent sets. It is easy to see that the persistent set computed by this algorithm satisfies the successor-state condition.

Algorithm 1 Persistent set.

Input: a set T_E of enabled atomic blocks.
Output: a persistent set P.
1. Let $T_P := \{\alpha\}$, where $\alpha \in T_E$.
2. For each atomic block $\alpha \in T_P$:
   (a) If $\alpha \in T_E$ (α is enabled): Add into T_P every atomic block β such that $(\alpha, \beta) \in D$.
   (b) If $\alpha \notin T_E$ (α is disabled): Add into T_P a set of atomic blocks whose executions guarantee α to become enabled.
3. Repeat step 2 until no more transition can be added into T_P.
4. $P := T_P \cap T_E$.

Algorithm 1 is a variant of the stubborn set algorithm presented in [11], that is, we use a valid dependence relation as the interference relation described in [11].

We apply POR to the ESST algorithm by modifying the ARF node expansion rule E2 (see Section 2) by first computing a persistent set from a set of scheduler states output by the function SCHED; and then by ensuring that the cycle condition is satisfied by the concretization of the constructed ARF.

First, we assume that a valid dependence relation D has been produced by static analysis on the threaded program. Second, we introduce the function PERSISTENT that computes a persistent set of a set of scheduler states. PERSISTENT takes as inputs an ARF node and a set $\mathcal{S}$ of scheduler states, and outputs a subset $\mathcal{S}'$ of $\mathcal{S}$. The input ARF node keeps track of the thread locations, which are used to identify atomic blocks, while the input scheduler states keep track of the status of the threads. From the ARF node and the set S, the function PERSISTENT extracts the set T_E of enabled atomic blocks. PERSISTENT then computes a persistent set P from T_E using Algorithm 1. Finally PERSISTENT constructs back a subset $\mathcal{S}'$ of the input set $\mathcal{S}$ of scheduler states from the persistent set P.

Let $A = (\langle l_1, \varphi_1\rangle, \ldots, \langle l_n, \varphi_n\rangle, \varphi, S)$ be an ARF node that is going to be expanded. We replace the rule E2 in the following way: instead of creating a new ART for each state $S' \in \text{SCHED}(S)$, we create a new ART whose root is the node $(\langle l_1, \varphi_1\rangle, \ldots, \langle l_n, \varphi_n\rangle, \varphi, S')$ for each state $S' \in \text{PERSISTENT}(A, \text{SCHED}(S))$ (rule E2').

To guarantee the preservation of safety properties, following [9], instead of checking that the cycle condition is satisfied we check a stronger condition: at least one state along the cycle is fully expanded.

In the ESST algorithm a *potential* cycle occurs if an ARF node is covered by one of its predecessors in the ARF. Let $A = (\langle l_1, \varphi_1\rangle, \ldots, \langle l_n, \varphi_n\rangle, \varphi, S)$ be an ARF node. We say that the scheduler state S is *running* if there is a running thread in S. Recall our assumption that there is at most one running thread in the scheduler state. We also say that the node A is *running* if its scheduler state S is. Note that during ARF expansion the input of SCHED is always a non-running scheduler state.

A path in an ARF can be represented as a sequence $A_0, \ldots, A_m$ of ARF nodes such that for all i, we have A_{i+1} is a successor of A_i in the same ART or there is an ARF connector from A_i to A_{i+1}. Given an ARF node A of ARF F, we denote by $ARFPath(A, F)$ the ARF path $A_0, \ldots, A_m$ such that (1) A_0 has neither a predecessor ARF node nor an incoming ARF connector, and (2) $A_m = A$. Let Π be an ARF path, we denote by $NonRunning(\Pi)$ the *maximal* sub-sequence of non-running node in Π.

Algorithm 2 ARF expansion algorithm for non-running node.

Input: a non-running ARF node A that contains no error locations.

1. Let $NonRunning(ARFPath(A, F))$ be $A_0, \ldots, A_m$ such that $A = A_m$
2. If there exists $i < m$ such that A_i covers A:
   (a) Let $A_{m-1} = (\langle l'_1, \varphi'_1 \rangle, \ldots, \langle l'_n, \varphi'_n \rangle, \varphi', S')$.
   (b) If $\text{PERSISTENT}(A_{m-1}, \text{SCHED}(S')) \subset \text{SCHED}(S')$:
      – For all $S'' \in \text{SCHED}(S') \setminus \text{PERSISTENT}(A_{m-1}, \text{SCHED}(S'))$:
         • Create a new ART with root node $(\langle l'_1, \varphi'_1 \rangle, \ldots, \langle l'_n, \varphi'_n \rangle, \varphi', S'')$.
3. If A is covered: Mark A as covered.
4. If A is not covered: Expand A by rule E2'.

Algorithm 2 shows how a non running ARF node A (containing no error locations) is expanded in the presence of POR. We fully expand the immediate non-running predecessor node of A when a potential cycle is detected. Otherwise the node is expanded as usual.

Our POR technique differs from that one described in [9], and implemented in SPIN [14]. The technique in [9] tries to select a persistent set that does not create a cycle, and if it does not succeed, then it fully expands the node. In the context of ESST such a technique is expensive. To detect a cycle one has to expand a node by a transition. As explained before, a transition or an atomic block in our case can span over multiple operations in CFA. Thus, a cycle detection often requires expensive computations of abstract strongest post condition.

In addition to coverage check, in the above algorithm one can also check if the detected cycle is spurious or abstract. We only fully expand a node iff the detected cycle is not spurious. As cycles are rare, the benefit of POR can be defeated by the price of generating and solving the constraints that encode the cycle.

In SystemC cycles can occur because there is a chain of wait and immediate notifications of events. For SystemC cycle detection can be optimized by only considering the sequence of non-running ARF node that belongs to the same delta-cycle.

POR based on sleep set can also be applied to ESST by extending the ARF node to include a sleep set. The application of sleep set technique into the ESST algorithm is similar to that of in the case of explicit-state model checking. Due to lack of space we refer to the appendix for a detailed discussion on the application of sleep set to ESST.

4.3 Correctness of Reduction in ESST

The correctness of POR with respect to verifying program assertions of transition systems has been shown by Theorem 4.4. The correctness proof relies on the enabledness and commutativity of independent transitions. However, the proof is applied in concrete state space of the transition system, while the ESST algorithm works in abstract state space represented by an ARF. The following observation shows that two transition that are independent in the concrete state space may not commute in the abstract state space.

For simplicity of presentation, we represent an abstract state by a formula representing a region. Let g_1, g_2 be global variables, and p, q be predicates such that

$p \Leftrightarrow (g_1 < g_2)$ and $q \Leftrightarrow (g1 = g2)$. Let α be the transition g_1 := g_1 - 1 and β be the transition g_2 := g_2 - 1. It is obvious that α and β are independent of each other. However, Fig. 4 shows that the two transition do no commute when we start from an abstract state A_1 such that $A_1 \Leftrightarrow p$. The edges in the figure represent the computation of abstract strongest post condition of the corresponding abstract states and transitions.

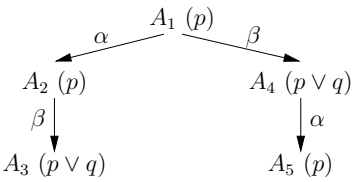

Fig. 4: Independent transitions do not commute in abstract state space.

Even though two independent transitions do not commute in abstract state space, they still commute in the concrete state space over-approximated by the abstract state space, as shown by the lemma below. For a concrete state s and an abstract state A, we write $s \models A$ iff A holds in s.

LEMMA 4.5 Let α and β be transitions that are independent of each other such that for concrete states s_1, s_2, s_3 and abstract state A we have $s_1 \models A$, and both $\alpha(s_1, s_2)$ and $\beta(s_2, s_3)$ holds. Let A' be the abstract successor state of A by applying the abstract strongest post operator to A and β, and A'' be the abstract successor state of A' by applying the abstract strongest post operator to A' and α. Then, there are concrete states s_4 and s_5 such that: (1) $\beta(s_1, s_4)$ holds, (2) $s_4 \models A'$, (3) $\beta(s_4, s_5)$ holds, (4) $s_5 \models A''$, and (5) $s_3 = s_5$. □

The above lemma shows that POR can be applied in abstract state space. The following theorem states the correctness of the reduction in the ESST technique.

THEOREM 4.6 *Let ARF F_1 be obtained by node expansion without* POR *and ARF F_2 be obtained with* POR. *F_1 is safe iff F_2 is safe.* □

5 Related Work

It is discussed in [3] an approach aiming at the generation of a SystemC scheduler from a SystemC design that at run time will use race analysis information to speed up the simulation by reducing the number of possible interleavings. The race condition itself is formulated as a guarded independence relation. The independence relation used in [3] is more precise than the one defined in this paper. In our work two transitions are independent if they are independent in any state. While in [3] two transitions are independent in some state if the guard associated with the pair of transitions is satisfied by the state. The guards of two transitions are computed using model checking techniques. The soundness of the synthesized scheduler relies on the assumption that the SystemC threads cannot enable each other during the evaluation phase. Therefore, immediate event notifications are not allowed. Our work does not have such an assumption. Moreover, unlike our work, the search in [3] is stateless and it is bounded on the number of simulation steps: cycle condition cannot be detected, and thus this approach cannot be used to verify program assertions.

Dynamic POR techniques for scalable testing of SystemC designs are described in [16, 12]. Unlike other works on dynamic POR that collect and analyze run time information to determine dependency of transitions, the work in [16] uses information obtained by static analysis to construct persistent set. For SystemC the criteria that they

use to determine dependency of transitions are similar to the criteria we used in this paper. However, similar to [3], the simulation is bounded and stateless, and thus cannot be used to verify program assertions.

POR has been successfully implemented in explicit-state model checkers, like SPIN [13, 14, 19] and VERISOFT [10]. Despite the inability of explicit-state model checkers to handle non-deterministic inputs, there have been several attempts to encode SystemC designs in PROMELA, the input language of SPIN. Examples of such attempts include the works described in [21, 5]. The aim of those work is to take advantages of search optimizations provided by SPIN, including POR. However, those works cannot fully benefit from the POR implemented in SPIN due to the limitations of the encodings themselves and the limitations of SPIN with respect to the intrinsic structure of SystemC designs. The encoding in [21] is unaware of atomic blocks in the SystemC design. The PROMELA encoding in [5] groups each atomic block in the SystemC design as an indivisible transition using SPIN atomic facility. Such an encoding can reduce the number of visited states, and thus relieves the state explosion problem. It is also shown in that paper that the application of POR reduces the number of visited states. However, the reductions that SPIN can achieve is not at the level of the atomic blocks of threads: SPIN still explores redundant interleavings. Moreover, to be able to handle cycle conditions similarly to Algorithm 2, one has to modify SPIN itself.

There have been some works on applying POR to symbolic model checking techniques as shown in [15, 23, 1]. In these works POR during state-space exploration is obtained by statically adding constraints describing the reduction technique into the encoding of the program. The work in [1] applies POR to symbolic BDD-based invariant checking. The work in [23] can be considered as symbolic sleep-set based technique. The work also introduces the notion of guarded independence relation, where a pair of transitions are independent of each other if certain conditions specified in the pair's guards are satisfied. Such an independence relation is used in [3] for race analysis. Our work in this paper can be extended to use guarded independence relation by exploiting the thread and global regions, but we leave it as future work. The work in [15] considers patterns of lock acquisition to refine the notion of independence transition, which subsequently yields better reductions.

6 Experiments

We have implemented the persistent and sleep sets based POR within the tool chain for SystemC verification described in [7]. It consists of a SystemC front-end, and a model checker called KRATOS which implements the ESST approach. The front-end translates SystemC designs into sequential C programs and into threaded C programs. KRATOS can verify the sequential C programs using the classical lazy abstraction algorithm, or it can verify the threaded C programs using the ESST algorithm (extended with the two POR techniques). KRATOS is built on top of an extended version of NUSMV [6] that integrates the MATHSAT SMT solver [4] and provides advanced predicate abstraction techniques by combining BDDs and SMT formulas.

We have performed an experimental evaluation on the same benchmarks used in [7]. We run the experiments on an Intel-Xeon DC 3GHz running Linux, equipped with

		Visited ARF nodes				Run time (.sec)			
Name	V	No-POR	P-POR	S-POR	PS-POR	No-POR	P-POR	S-POR	PS-POR
toy	S	896	856	624	592	1.900	1.900	1.890	1.900
toy-bug-1	U	834	794	562	530	1.800	1.800	1.800	1.700
toy-bug-2	U	619	589	415	391	0.600	0.600	0.600	0.600
token-ring-1	S	60	60	60	60	0.000	0.010	0.010	0.000
token-ring-2	S	161	153	127	119	0.090	0.090	0.100	0.090
token-ring-3	S	417	285	259	195	0.190	0.090	0.100	0.090
token-ring-4	S	1039	480	538	296	0.400	0.200	0.200	0.200
token-ring-5	S	2505	1870	961	742	0.800	0.690	0.390	0.400
token-ring-6	S	5883	1922	2114	556	2.000	0.600	0.800	0.300
token-ring-7	S	13533	4068	4324	948	4.600	1.200	1.500	0.400
token-ring-8	S	30623	7781	8264	1293	12.400	2.390	2.800	0.600
token-ring-9	S	68385	17779	15938	3262	40.190	5.600	5.690	1.300
token-ring-10	S	151075	41517	30192	4531	155.390	16.500	12.600	1.900
token-ring-11	S	330789	78229	59310	9564	595.640	43.590	34.590	3.800
token-ring-12	S	M.O.	119990	121616	11106	M.O.	85.990	107.590	4.490
token-ring-13	S	M.O.	M.O.	230479	108783	M.O.	M.O.	344.360	96.280
transmitter-1	U	32	32	32	32	0.010	0.010	0.010	0.010
transmitter-2	U	83	45	67	45	0.010	0.010	0.010	0.010
transmitter-3	U	209	66	114	66	0.010	0.010	0.010	0.010
transmitter-4	U	509	176	254	98	0.090	0.010	0.010	0.010
transmitter-5	U	1205	88	478	88	0.190	0.010	0.090	0.010
transmitter-6	U	2789	483	918	172	0.400	0.100	0.100	0.010
transmitter-7	U	6341	307	2124	121	1.000	0.000	0.300	0.010
transmitter-8	U	14213	3847	4540	337	2.690	0.700	0.690	0.100
transmitter-9	U	31493	1209	7964	214	8.390	0.200	1.500	0.010
transmitter-10	U	69125	2053	13943	290	32.890	0.400	2.800	0.100
transmitter-11	U	150533	3298	36348	289	130.690	0.690	9.690	0.100
transmitter-12	U	M.O.	9784	50026	640	M.O.	2.200	21.390	0.200
transmitter-13	U	M.O.	4234	108796	334	M.O.	1.000	75.590	0.100
pipeline	S	25347	7135	8568	7135	205.270	54.690	61.000	54.690
kundu	S	1004	1004	1004	1004	1.190	1.200	1.180	1.200
kundu-bug-1	U	221	221	221	221	0.390	0.400	0.390	0.390
kundu-bug-2	U	866	866	866	866	1.090	1.100	1.200	1.200
bistcell	S	305	305	305	305	0.500	0.500	0.490	0.500
pc-sfifo-1	S	152	152	152	152	0.290	0.300	0.300	0.290
pc-sfifo-2	S	197	197	197	197	0.300	0.300	0.290	0.300
mem-slave-1	S	556	556	556	556	3.390	3.400	3.400	3.400
mem-slave-2	S	992	992	992	992	15.000	15.090	15.390	15.190
mem-slave-3	S	1414	1414	1414	1414	181.980	173.080	183.770	193.180
mem-slave-4	-	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.
mem-slave-5	-	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.

Table 1. Results of the experimental evaluation.

4GB RAM. We fixed the time limit to 1000 seconds and the memory limit to 2GB. We experimented without any POR technique (No-POR), enabling only the persistent set reduction (P-POR), enabling only the sleep set reduction (S-POR), and finally enabling both the persistent and the sleep set reductions (PS-POR).

The results of the experimental evaluation are reported in Table 1. The first column lists the name of the benchmarks. The second column shows the status of the verification: S for safe, U for unsafe, - for unknown. Then we report for each experimented technique the number of visited ARF nodes, and the run time. On the table we indicate out of time with T.O., and out of memory with M.O. All the material to reproduce the experiments can be found at `http://es.fbk.eu/people/roveri/tests/tacas2011`.

The table clearly shows that with POR we can verify more benchmarks. Without POR the verifications of `token-ring-`x, and, `transmitter-`x, with $x \in \{12, 13\}$ resulted in out of memory. For some benchmarks, like `bist-cell`, `kundu`, `mem-slave-`x, the POR defined in this work is not applicable. In such benchmarks

atomic blocks of threads access the same global variables, and thus are dependent on each other. When POR is not applicable, we can see from the table that there is no reduction in the number of visited ARF nodes. However, the times spent for the verification are almost identical. It means that the time spent for the computation of persistent and sleep sets when POR is enabled is negligible.

The table also shows that, on the `pipeline`, `token-ring-`x and `transmitter-`x POR results in a significant reduction in the number of visited ARF nodes, and in a reduction of run time. Moreover, the combination of persistent and sleep sets gives the best results in terms of visited nodes and run time.

With POR enabled, we are still not able to verify `mem-slave-4` and `mem-slave-5` given the resource limits. This is because KRATOS employs a precise predicate abstraction for expanding ARF nodes. Such an abstraction is expensive when there are a large number of predicates involved. For verifying `mem-slave-3`, we already discovered 65 predicates associated with the global region and 37 predicates associated with the thread regions, with an average of 5 predicates per location associated with the thread CFA.

POR, in principle, could interact negatively with the ESST algorithm. The construction of ARF in ESST is sensitive to the explored scheduler states and to the tracked predicates. POR prunes some scheduler states that ESST has to explore. However, exploring such scheduler states can yield a smaller ARF than if they are omitted. In particular, for an unsafe benchmark, exploring omitted scheduler states can lead to the shortest counter-example path. Furthermore, exploring the omitted scheduler states could lead to spurious counter-example ARF paths that yield predicates that allow ESST to perform less refinements and construct a smaller ARF.

Regardless of the fact that there is no guarantee that POR always boosts the ESST algorithm, Table 1 shows that POR can be useful in improving the performance of the ESST algorithm. In the future we will investigate the effectiveness of POR in the ESST algorithm by using randomization on deciding the set of omitted scheduler states to improve the quality of the analysis.

7 Conclusion and Future Work

In this paper we have shown how to extend the ESST approach with POR techniques for the verification of SystemC designs. We proved the correctness of the approach for the verification of program assertions, implemented the approach in the KRATOS model checker, and experimentally evaluated the approach. The proposed techniques significantly reduces the number of visited nodes and the verification time, and allows for the verification of designs that could not be verified without POR.

As future work, we will investigate how to extend the ESST approach to deal with symbolic primitive functions. This requires a generalization of the scheduler exploration with a hybrid (explicit-symbolic) approach, and the use of AllSMT techniques to enumerate all possible next states. We will also apply the ESST techniques to the verification of concurrent C programs from other application domains (e.g. robotics, railways), where different scheduling policies have to be taken into account.

References

1. Alur, R., Brayton, R.K., Henzinger, T.A., Qadeer, S., Rajamani, S.K.: Partial-order reduction in symbolic state-space exploration. Formal Methods in System Design 18(2), 97–116 (2001)
2. Beyer, D., Henzinger, T.A., Jhala, R., Majumdar, R.: The software model checker Blast. STTT 9(5-6), 505–525 (2007)
3. Blanc, N., Kroening, D.: Race analysis for SystemC using model checking. In: ICCAD. pp. 356–363. IEEE (2008)
4. Bruttomesso, R., Cimatti, A., Franzén, A., Griggio, A., Sebastiani, R.: The MathSAT 4SMT Solver. In: CAV. LNCS, vol. 5123, pp. 299–303. Springer (2008)
5. Campana, D., Cimatti, A., Narasamdya, I., Roveri, M.: SystemC verification via an encoding in Spin, submitted for publication
6. Cimatti, A., Clarke, E.M., Giunchiglia, F., Roveri, M.: NuSMV: A New Symbolic Model Checker. STTT 2(4), 410–425 (2000)
7. Cimatti, A., Micheli, A., Narasamdya, I., Roveri, M.: Verifying SystemC: a Software Model Checking Approach. In: FMCAD (2010), to appear
8. Clarke, E.M., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement for symbolic model checking. J. ACM 50(5), 752–794 (2003)
9. Clarke, E.M., Grumberg, O., Peled, D.A.: Model Checking. The MIT Press (1999)
10. Godefroid, P.: Software Model Checking: The VeriSoft Approach. F. M. in Sys. Des. 26(2), 77–101 (2005)
11. Godefroid, P.: Partial-Order Methods for the Verification of Concurrent Systems - An Approach to the State-Explosion Problem, LNCS, vol. 1032. Springer (1996)
12. Helmstetter, C., Maraninchi, F., Maillet-Contoz, L., Moy, M.: Automatic generation of schedulings for improving the test coverage of systems-on-a-chip. In: FMCAD. pp. 171–178. IEEE Computer Society (2006)
13. Holzmann, G.J.: Software model checking with SPIN. Advances in Computers 65, 78–109 (2005)
14. Holzmann, G.J., Peled, D.: An improvement in formal verification. In: 7th IFIP WG6.1 Int. Conf. on Formal Description Techniques VII. pp. 197–211. London, UK, UK (1995)
15. Kahlon, V., Gupta, A., Sinha, N.: Symbolic model checking of concurrent programs using partial orders and on-the-fly transactions. In: CAV. LNCS, vol. 4144, pp. 286–299. Springer (2006)
16. Kundu, S., Ganai, M.K., Gupta, R.: Partial order reduction for scalable testing of systemC TLM designs. In: DAC. pp. 936–941. ACM (2008)
17. Moy, M., Maraninchi, F., Maillet-Contoz, L.: Pinapa: an extraction tool for SystemC descriptions of systems-on-a-chip. In: EMSOFT. pp. 317–324. ACM (2005)
18. Peled, D.: All from one, one for all: on model checking using representatives. In: CAV. LNCS, vol. 697, pp. 409–423. Springer (1993)
19. Peled, D.: Combining partial order reductions with on-the-fly model-checking. Formal Methods in System Design 8(1), 39–64 (1996)
20. Tabakov, D., Kamhi, G., Vardi, M.Y., Singerman, E.: A Temporal Language for SystemC. In: FMCAD. pp. 1–9. IEEE (2008)
21. Traulsen, C., Cornet, J., Moy, M., Maraninchi, F.: A SystemC/TLM Semantics in Promela and Its Possible Applications. In: SPIN. LNCS, vol. 4595, pp. 204–222. Springer (2007)
22. Valmari, A.: Stubborn sets for reduced state generation. In: APN 90: Proceedings on Advances in Petri nets 1990. pp. 491–515. Springer-Verlag, New York, NY, USA (1991)
23. Wang, C., Yang, Z., Kahlon, V., Gupta, A.: Peephole partial order reduction. In: TACAS. LNCS, vol. 4963, pp. 382–396. Springer (2008)