An Analytic Evaluation of SystemC Encodings in Promela

D. Campana, A. Cimatti, I. Narasamdya, and M. Roveri

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

Abstract. SystemC is a de-facto standard language for high-level modeling of systems on chip. We investigate the feasibility of explicit state model checking of SystemC programs, proposing several ways to convert SystemC into Promela. We analyze the expressiveness of the various encoding styles, and we experimentally evaluate their impact on the search carried out by SPIN on a significant set of benchmarks. We also compare the results with recent approaches to symbolic verification of SystemC. Our approach never returns false positives, detects assertion violations much faster than recent formal approaches, and has the novel feature of pinpointing non-progressing delta cycles.

1 Introduction

SystemC is a de-facto standard language for writing high-level executable system designs of System-on-Chips (SoCs). SystemC allows for high-speed simulations before synthesizing the RTL hardware description. A SystemC design can be viewed as a multi-threaded program with a specific scheduling policy [19].

Verification of SystemC is crucial in order to pinpoint errors in the design (or in the specification), and to prevent their propagation down to the hardware. Despite the efficiency of simulations, simulation-based verification is not always effective in finding bugs, due to inherently complex behaviors caused by concurrency and thread interleavings. Since each simulation only explores one possible sequence of thread interleavings, simulations can miss bugs that are caused by unexplored thread interleavings.

Formal verification of SystemC has recently gained significant interests with the use of software model checking [5,6,14] and bounded model checking [8,9] techniques, or by reduction to model checking for finite state [17,21] and timed [11] systems.

In this paper, we present and evaluate a family of encodings of SystemC designs in Promela [12]. All the proposed encodings accurately model the full semantics of SystemC. Yet, they feature different characteristics in terms of the properties that can be verified and in terms of the efficiency of the search carried out by SPIN [12]. All the proposed encodings are obtained by combining a generic encoding of the SystemC scheduler, and an encoding of each thread in the SystemC design, and they all allow for checking program assertions. They differ in the way the scheduler-threads synchronization is modeled, and in the use of Promela constructs to control atomicity. As a result, the various encodings can allow for observations of the SystemC design at different granularities. In fact, the simulation of a SystemC design is divided into one or more delta cycles, which in turn are composed of several phases, during which each thread can be activated multiple times. Depending on the specific features of the encoding,

there may be significant variations in the number of stored states and explored transitions during the search. The work in [21] also proposes a Promela encoding of SystemC designs, but the encoding covers only a subset of SystemC semantics.

Detection of the presence of non-progressing delta cycles is very important. Indeed, if a design shows a non-progressing delta cycle, then the design is not synthesizeable. Clearly it is impossible to verify that the SystemC design will eventually exit from every delta cycle using simulation. Our approach is also unique in the ability to check for the absence of non-progressing delta cycles in the SystemC design. This is made possible by the accurate encoding of the scheduler, and is carried out by reduction to checking the absence of non-progress cycles with SPIN.

We carry out a thorough investigation of the features of the various encodings, first analytically, and then experimentally. We have implemented the encodings in a tool called SC2PROMELA. The SPIN model checker is called with different modalities, to check for assertion violations and for the presence of non-progressing delta cycles. We conduct experiments with the encodings on a significant set of benchmarks taken and adapted from [6]. The results of experiments support the analytical conjectures in terms of the features of the search. Moreover, our techniques allow us to discover that some of the benchmarks used in [6] have non-progressing delta cycles.

The proposed approach is oriented to bug finding. In fact, all the encodings proposed in this paper exhibit under-approximations of SystemC designs. That is, we restrict the input values that can be read from the environment. In other words, we perform the verification of the SystemC designs under certain assumptions about the input values.

We also compared the different encodings with ESST [5], a recent and promising technique for SystemC verification. ESST is based on the combination of explicit state model checking techniques to handle the SystemC scheduler with the symbolic lazy abstraction [10] technique to handle the threads. This comparison clearly shows that our Promela encodings are complementary to ESST in that they appear to be very effective in finding bugs that are nested deeply inside a thread. ESST often requires many refinement steps, and turns out to be surprisingly inefficient.

This paper is organized in the following way. Section 2 introduces SystemC and shows a running example that will be used throughout the paper. Section 3 shows our Promela encodings. Section 4 describes analytical comparisons of the encodings. Section 5 describes the experimental evaluation. Section 6 discusses some related work. Finally, Section 7 concludes this paper and outlines some future work.

2 Structure of SystemC Design

SystemC is a C++ library that provides a core language for specifying components of a system design and their interconnections by means of, respectively, modules and channels. The core language also provides threads for modeling the parallel behavior of the design and events for synchronizing the threads. In addition to the core language, SystemC includes a scheduler that runs the threads during a simulation. Thus, a SystemC design can be viewed as a multi-threaded program with a specific scheduler, as shown in Figure 1.

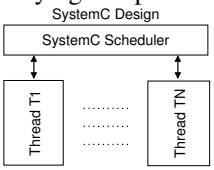


Fig. 1: Static view of a SystemC design.

```
1  SC_MODULE( Producer ) {
2    sc_event e;
3  public:
4    sc_in<int> port_in;
5    sc_out<int> port_out;
6    SC_CTOR( Producer ) {
7      SC_THREAD( write );
8      SC_METHOD( read_ack );
9      sensitive << port_in; dont_initialize();
10   }
11   void write() {
12     wait(SC_ZERO_TIME);
13     while (1) {
14       port_out.write( input() );
15       wait(e);
16     }
17   }
18   void read_ack() {
19     output( port_in.read() );
20     e.notify();
21   }
22 };
```

```
27 SC_MODULE( Consumer ) {
28 public:
29   sc_in<int> port_in;
30   sc_out<int> port_out;
31   SC_CTOR( Consumer ) {
32     SC_METHOD( read_and_ack );
33     sensitive << port_in;
34     dont_initialize();
35   }
36   void read_and_ack() {
37     int ack = process ( port_in.read() );
38     port_out.write(ack);
39   }
40 };
41 int sc_main() {
42   sc_signal<int> p_to_c, c_to_p;
43   Producer *p = new Producer("P");
44   Consumer *c = new Consumer("C");
45   p->port_out(p_to_c); c->port_in(p_to_c);
46   p->port_in(c_to_p); c->port_out(c_to_p);
47   sc_start();
48 }
```

Fig. 2: Example of SystemC design.

An example of a SystemC design is depicted in Figure 2. In the example we have two components, `Producer` and `Consumer`. The constructor of the `Producer` specifies two threads: `write` for writing value to the `Consumer` through the channel bound to its output port `port_out`, and `read_ack` for reading the acknowledgment value sent by the `Consumer`. For simplicity, we treat a process declared by `SC_METHOD` as a thread that suspends itself only if its execution terminates. The constructor also says that the thread `read_ack` is sensitive to the value update of the channel bound to the input port `port_in`, and that it is not runnable at the beginning of the simulation (see call to `dont_initialize()`).

SystemC simulation can be divided into so-called delta cycles. The transition from one delta-cycle into the next one occurs when there are no more runnable threads. In our example the thread `write` first waits until the next delta cycle, by `wait(SC_ZERO_TIME)`, before entering the loop that feeds value for the `Consumer`. Every time it writes a value to the `Consumer`, it suspends itself and waits for the notification of the event `e` by `wait(e)`. The structure and behavior of `Consumer` can be described similarly to that of the `Producer`. The main function `sc_main` instantiates one `Producer` and one `Consumer`, binds their ports to the appropriate signal channels, and then starts the simulation by `sc_start()`.

The SystemC scheduler has several phases[1], as shown in Figure 3(a). In the initialization phase all channels are updated according to their first initialization, and a delta notification is performed to signify the start of a delta cycle. The scheduler then enters the evaluation phase. In this phase the scheduler runs all runnable threads, one at a time, until there are no more runnable threads, while postponing the materializations of channel updates. This phase constitutes one delta cycle.

Having no more runnable threads, the scheduler enters the channel-update phase where it materializes all channel updates. These updates could make some threads runnable, and so the scheduler goes back to the evaluation phase to run these threads. Otherwise, the scheduler accelerates the simulation time to the nearest time point where there exists an event to be notified. This event notification in turn could make some threads runnable, and so the scheduler goes back to the evaluation phase again. Otherwise, the scheduler ends the simulation.

[1] For further details and discussions about the SystemC scheduler we refer the reader to [19].

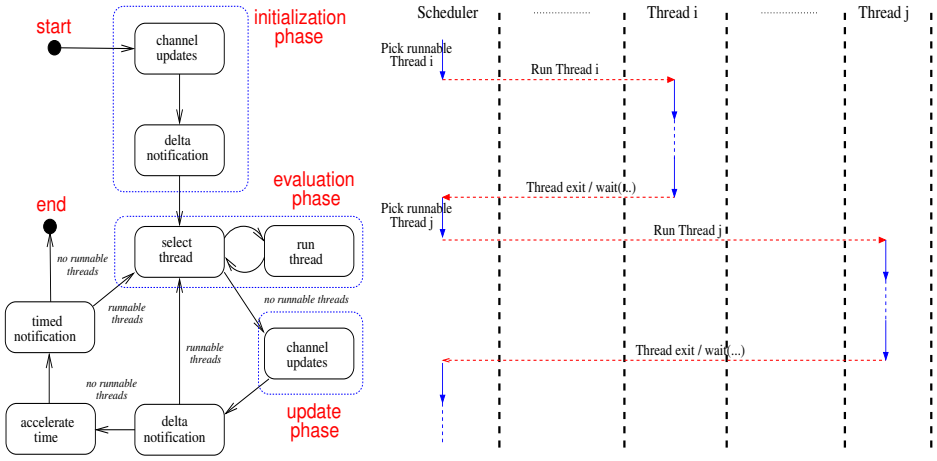


Fig. 3: (a) The SystemC scheduler. (b) The dynamic view of a SystemC design.

The evaluation phase of the scheduler employs a cooperative scheduling with an exclusive run of each thread. That is, as shown in Figure 3(b), when the scheduler gives the control to a thread to run, it never preempts the control from the running thread, but waits until the thread suspends itself and gives the control back to the scheduler.

3 Encoding SystemC in Promela

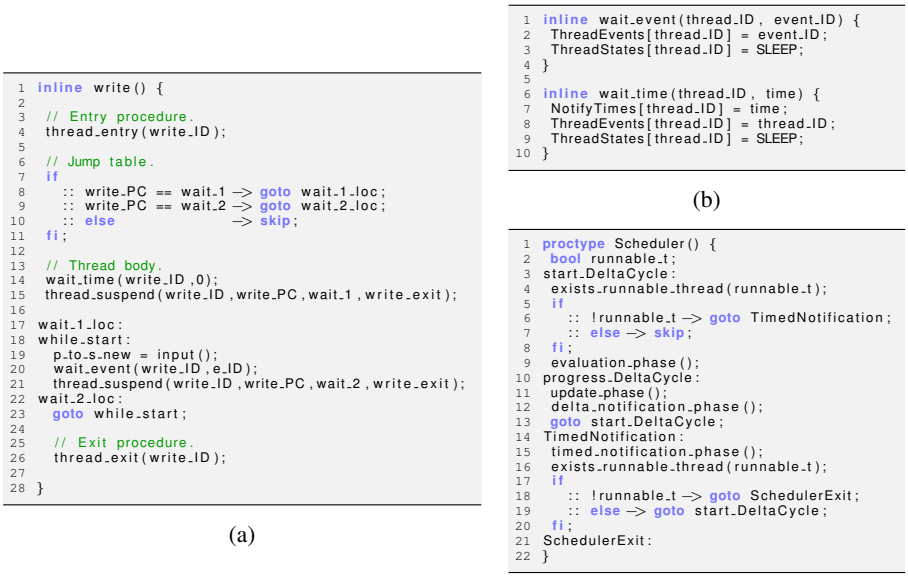
In this work we consider three different Promela encodings of SystemC designs. In all of these encodings we capture the full semantics of SystemC. These encodings differ in the number of Promela processes, the modeling of communication between the threads and the scheduler, and the granularity of atomicity in simulations. These differences can affect the search behavior of the protocol analyzer generated from the resulting Promela models as well as kinds of property that can be verified.

In developing the encodings, we rely on several assumptions. First, we assume that the number of threads and events are known a priori. Thus, we do not handle SystemC designs that contain dynamic creations of objects or threads. Second, we assume that all function calls can statically be inlined. These limitations, however, do not affect the applicability of the proposed techniques since, to the best of our knowledge, most real SystemC design satisfy these assumptions.

For presenting our encodings, we will use the example introduced in the previous section. Although the example only uses a small subset of SystemC features, the example is sufficient to describe our encodings. We believe that other SystemC features that are not used in the example can be encoded in a similar way to those here presented.

3.1 Basic Building Blocks

The three encodings we propose in this paper are built on some basic building blocks that represent the imperative constructs of the body of the SystemC threads and channels, the synchronization primitives, and the scheduler.



```
inline write() {

 // Entry procedure.
 thread_entry(write_ID);

 // Jump table.
 if
   :: write_PC == wait_1 -> goto wait_1_loc;
   :: write_PC == wait_2 -> goto wait_2_loc;
   :: else               -> skip;
 fi;

 // Thread body.
 wait_time(write_ID,0);
 thread_suspend(write_ID,write_PC,wait_1,write_exit);

wait_1_loc:
while_start:
  p_to_s_new = input();
  wait_event(write_ID,e_ID);
  thread_suspend(write_ID,write_PC,wait_2,write_exit);
wait_2_loc:
  goto while_start;

  // Exit procedure.
  thread_exit(write_ID);

}
```

(a)

```
inline wait_event(thread_ID, event_ID) {
 ThreadEvents[thread_ID] = event_ID;
 ThreadStates[thread_ID] = SLEEP;
}

inline wait_time(thread_ID, time) {
 NotifyTimes[thread_ID] = time;
 ThreadEvents[thread_ID] = thread_ID;
 ThreadStates[thread_ID] = SLEEP;
}
```

(b)

```
proctype Scheduler() {
 bool runnable_t;
start_DeltaCycle:
 exists_runnable_thread(runnable_t);
 if
   :: !runnable_t -> goto TimedNotification;
   :: else -> skip;
 fi;
 evaluation_phase();
progress_DeltaCycle:
 update_phase();
 delta_notification_phase();
 goto start_DeltaCycle;
TimedNotification:
 timed_notification_phase();
 exists_runnable_thread(runnable_t);
 if
   :: !runnable_t -> goto SchedulerExit;
   :: else -> goto start_DeltaCycle;
 fi;
SchedulerExit:
}
```

(c)

Fig. 4: Encodings of (a) thread, (b) synchronization primitive, and (c) scheduler.

Encoding threads. We first associate each SystemC thread t with a unique id t_ID. Figure 4(a) shows the encoding of the thread `write` in the previous example. Each thread t is encoded as inline code that starts with an entry procedure encoded in the inline code `thread_entry(`t`_ID)` and ends with an exit procedure encoded in the inline code `thread_exit(`t`_ID)`.

The entry procedure `thread_entry(`t`_ID)` describes the passing of control from the scheduler to the thread. Because the synchronization between the scheduler and the threads are modeled differently in our encodings, so are the expansions of `thread_entry` in the encodings.

The entry procedure is then followed by a "jump table" that determines where the execution of the thread body has to start or, in case of thread suspension, to resume. To be able to resume the thread execution from the middle of the thread body, we have to keep track the program counter. To this end, we associate each thread t with a variable t_PC whose values correspond to code labels.

The translation of the thread body into Promela code is straight-forward due to the large similarities of the imperative constructs between SystemC and Promela. For example, the `while` loop can be translated into Promela `if-fi` and `goto` constructs.

When its execution is suspended, by for example calling wait functions, the thread t has to (1) update t_PC with the location where it will resume its execution, (2) go to the suspension location, which is the exit label t_exit, and (3) return the control back to the scheduler. These three steps are encoded in `thread_suspend(`t`_ID,`t`_PC,rl,sl)`, where `rl` is a value that corresponds to the resume location, and `sl` is the suspension location. We will show later that the exit label t_exit is placed immediately following the inline code that encodes the thread.

On exiting the execution, the thread t has to establish its sensitivity that could dynamically change during its execution, and then it has to give the control back to the scheduler. We encode this exiting procedure in `thread_exit(`t`_ID)`. Similar to `thread_entry`, the expansions of `thread_suspend` and `thread_exit` can be different in our encodings.

Encoding channels. Figure 4(a) also provides a glimpse of how SystemC channels can be encoded into Promela. For example, a SystemC signal s or a port bound to that signal is modeled by a pair (s`_new`, s`_old`) of variables such that every write to s is a write to s`_new` and every read from s is a read from s`_old`. We also associate an event for each signal, and use it to wake up the threads sensitive to that signal.

For each signal, we have the encoding of its update function as an inline Promela code. This code is executed in the channel update phase and accounts for updating the content of s`_old` with the value of s`_new` when they are not equal. When the content of s`_old` gets updated, the inline code will notify the event associated with the signal s. Other kinds of channel can be encoded in a similar way.

Encoding synchronization primitives. Similar to the threads we associate each event e with a unique id e`_ID`. To encode synchronization primitives, the encoding of the scheduler maintains three different arrays: `ThreadStates` indexed by thread id's, `ThreadEvents` indexed by thread id's, and `NotifyTimes` indexed by event id's. `ThreadStates` contains the states of threads. A thread state is either `RUNNABLE` or `SLEEP`. `ThreadEvents` contains the id's of the events whose notifications are waited by the threads. While, `NotifyTimes` indicates the notification times of events.

We encode the call to `wait(`e`)`, for an event e, into inline code `wait_event(`t`_ID,`e`_ID)`, as shown in Figure 4(b), such that t is the calling thread. We note in `ThreadEvents` that the calling thread t is now waiting for an event e, and update the thread's state in `ThreadStates` to `SLEEP`.

To encode `wait_time`, we associate each thread t with an event. We reuse the id of the thread as id for the event associated with it. Then, using the array `NotifyTimes`, we note that the event associated with the thread should be woken up at some time.

The encoding of `notify_event(`e`)` is as follows: we iterate over the array `ThreadStates` to check if a thread t is sleeping or not. If t is sleeping and is waiting for the notification of e, then we update the state of t to `RUNNABLE` and we update the arrays `ThreadEvents` and `NotifyTimes` at index e`_ID` with -1.

Encoding scheduler. The encoding of the scheduler, shown in Figure 4(c), follows the description shown in Figure 3(a). The inline code `evaluation_phase` simply non-deterministically selects a thread whose status is `RUNNABLE` and executes the thread. The inline code `update_phase` updates every channel that needs to be updated by executing the corresponding update function. The inline code `delta_notification_phase` changes the status of a thread from `SLEEP` to `RUNNABLE` if the thread is sensitive to the channel that has just been updated or is waiting for the notification of an event that should be notified at the delta-cycle boundary. The inline

```
1  inline thread_entry(t_ID) {
2    ControlChannel[t_ID] ? CONTROL_TOKEN;
3  }
4
5  inline thread_exit(t_ID) {
6   ThreadState[t_ID] = SLEEP;
7   ThreadEvents[t_ID] = ThreadSensitivity[t_ID];
8   ControlChannel[t_ID] ! CONTROL_TOKEN;
9  }
10
11 inline thread_suspend(t_ID, t_PC, res_loc, susp_loc) {
12  thread_PC = res_loc;
13  ThreadState[t_ID] = SLEEP;
14  ControlChannel[t_ID] ! CONTROL_TOKEN;
15  goto susp_loc;
16 }
17
18 inline thread_suspend_optimize(t_ID) {
19  ThreadState[t_ID] = SLEEP;
20  ControlChannel[t_ID] ! CONTROL_TOKEN;
21  ControlChannel[t_ID] ? CONTROL_TOKEN;
22 }
```

```
1  proctype write_thread() {
2   write_entry:
3    atomic { write(); }
4   write_exit:
5    goto write_entry;
6  }
7
8  inline evaluation_phase() {
9   do
10   :: ThreadStates[write_ID] == RUNNABLE
11        -> ControlChannel[thread_ID] ! CONTROL_TOKEN;
12           ControlChannel[thread_ID] ? CONTROL_TOKEN;
13   :: ThreadStates[read_ack_ID] == RUNNABLE
14        -> ControlChannel[read_ack_ID] ! CONTROL_TOKEN;
15           ControlChannel[read_ack_ID] ? CONTROL_TOKEN;
16   :: else -> break;
17  od;
18 }
```

Fig. 6: Thread-to-process encoding: Promela code.

code `timed_notification_phase` encodes the time acceleration and time notification. The time acceleration is modeled by substracting the notification times in the array `NotifyTimes` by the distance to the nearest time point where an event can be notified. To support the under-approximations, we can set the limit of simulation time.

In our Promela encodings, the schedulers differ only in the inline code `evaluation_phase`, while other parts remain the same.

Encoding non-deterministic inputs. We encode input reading as inline code that selects non-deterministically values from a finite set of random values. This can be thought as verifying the design under certain assumptions about the input values.

3.2 Thread-To-Process Encoding

Our first encoding treats the scheduler and the threads as separate Promela processes, as shown in Figure 5. In particular the encoding of each thread can optionally be enclosed within a Promela `atomic` block. We call our first encoding *thread-to-process*.

In this encoding the synchronization between threads and the scheduler is modeled by passing a `CONTROL_TOKEN` through the Promela rendezvous channels that connect the threads with the scheduler. We maintain the array `ControlChannel` of Promela rendezvous channels such that each rendezvous channel is associated with a thread by means of `thread_ID`. Recall that passing control from the scheduler to a thread is encoded in the inline code `thread_entry`. Such a control passing is modeled as sending (operator !) and receiving (operator ?) the `CONTROL_TOKEN` through the designated rendezvous channel, as shown in Figure 6. Following the semantics of Promela, the receive statement is executable only if the associated rendezvous channel receives the `CONTROL_TOKEN`.

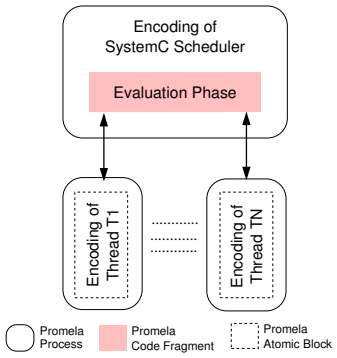
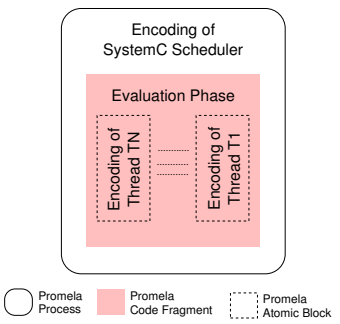


Fig. 5: Thread-to-process encoding.



(a)

```
1  inline write_thread() {
2
3    atomic {
4      write();
5    }
6    write_exit:
7      skip;
8
9  }
10
11 inline evaluation_phase() {
12   do
13     :: ThreadStates[write_ID] == RUNNABLE
14          -> write_thread();
15     :: ThreadStates[read_ack_ID] == RUNNABLE
16          -> read_ack_thread();
17     :: else -> break;
18   od;
19 }
```

(b)

Fig. 7: (a) Thread-to-atomic-block encoding, and (b) threads and scheduler encoding.

Exiting and suspending the thread execution in thread-to-process encoding are modeled as sending the `CONTROL_TOKEN` back to the scheduler, as shown in Figure 6 by the inline code `thread_exit` and `thread_suspend`, respectively. On exiting the thread execution, we set the state of the thread to `SLEEP` and establish the event sensitivity of the thread. Note that, for presentation only, we assume that each thread is sensitive to at most one event. However, in our implementation the elements of the arrays `ThreadEvents` and `ThreadSensitivity` are of composite type that can handle threads with more than one sensitive events.

Similar to exiting the thread execution, on suspending the thread we set the state of the thread to `SLEEP`. Unlike exiting the thread execution, before handing over the control to the scheduler, we first have to set the program counter of the thread, which is `thread_PC`, to point to the location where the thread will resume its execution, which is represented by `resume_loc`. Having passed the control to the scheduler, the thread then waits for the control at the suspension location represented by `suspend_loc`.

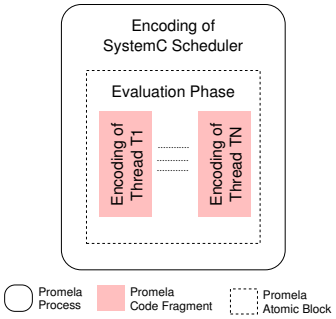
Using the example of thread `write`, Figure 6 shows that the thread itself is encoded as a non-terminating process `write_thread`, such that its body, which is encoded as the inline code `write`, can be enclosed within an atomic block. Note that the body of the thread can only be executed if the thread gets the `CONTROL_TOKEN` from the scheduler.

Figure 6 also shows the encoding of the evaluation phase. To model the cooperativeness of this phase and to allows the exploration of all possible thread interleavings, the scheduler non-deterministically picks a runnable thread and sends the `CONTROL_TOKEN` to the thread. It then waits for the thread to give back the `CONTROL_TOKEN`.

We can optimize the thread-to-process encoding by using the message passing on the rendezvous channels to keep track of the program counter implicitly. This optimization amounts to replacing `thread_suspend` by `thread_suspend_optimize` shown in Figure 6. With this optimization, the program counter t`_PC` for the thread t and the jump table in the thread encoding are no longer needed.

3.3 Thread-To-Atomic-Block Encoding

Another alternative Promela encoding for a SystemC design is to encode the design as a single Promela process containing both the scheduler and the threads, such that each



(a)

```
1 proctype Scheduler() {
2   ...
3   atomic {
4     evaluation_phase();
5   }
6   progress_DeltaCycle:
7   ...
8 }
```

(b)

Fig. 8: (a) One-atomic-block encoding. (b) Encoding of the scheduler.

thread is encoded as an atomic block inside the encoding of the evaluation phase, as shown in Figure 7(a). We call this encoding *thread-to-atomic-block*.

Unlike the thread-to-process encoding, in this encoding we no longer need the rendezvous channels and the control token. The encodings of `thread_entry`, `thread_exit`, and `thread_suspend` in thread-to-atomic-block encoding can be obtained from that of the thread-to-process encoding by replacing the sending and receiving statements with the `skip` statement.

The thread itself is now encoded as inline code instead of a process. The body of the thread encoding is surrounded by an atomic block. Passing control to the thread in the evaluation phase is modeled by executing the inline code of the thread encoding, while giving back control to the scheduler is modeled by simply finishing the execution of the inline code. The encodings of the thread and the scheduler for this thread-to-atomic-block encoding are shown in Figure 7(b).

3.4 One-Atomic-Block Encoding

Yet another alternative encoding can be derived from the thread-to-atomic-block encoding, that is, by enclosing the whole evaluation phase in an atomic block. We call such encoding *one-atomic-block* encoding and depicts its structure in Figure 8(a). Although the modification from the thread-to-atomic-block encoding is small, as we will discuss in the next section, it can change the search behavior dramatically and affect the property that can be verified.

4 Analytical Comparison of Encodings

The Promela encodings of SystemC designs proposed in this paper capture the full semantics of SystemC by modeling all the phases of the SystemC scheduler. However, they differ in terms of (1) the kinds of property that can be verified, and (2) the search behavior during the verification.

4.1 Properties to Verify

A general approach for defining temporal languages for SystemC has been described in [19]. The approach allows for greater flexibility in the temporal resolution granularity

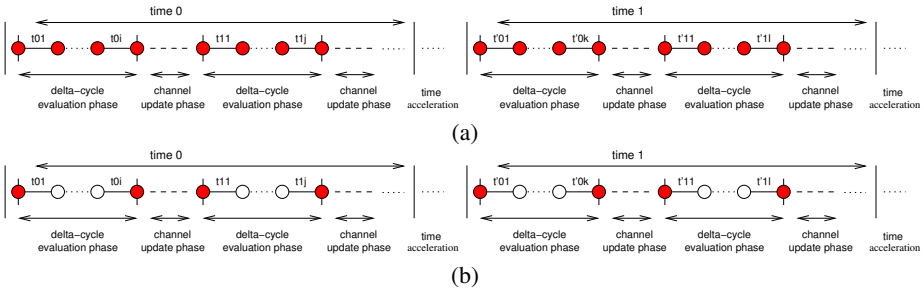


Fig. 9: Sampling rates of (a) thread-to-process and thread-to-atomic-block encodings, and (b) one-atomic-block encoding.

of the specification through the definition of finer grained execution traces following the state machine of SystemC scheduler described in Figure 3(a).

Similar to [19], the encodings presented here define the sampling rates of stored states in the execution traces. In the thread-to-process and the thread-to-atomic-block encodings the states before and after the execution of a thread in the evaluation phase are sampled (see the dark circles in Figure 9(a)). In addition to checking for assertion violations, this kind of sampling allows us to check for consistency properties that should hold before and after the execution of each thread in the evaluation phase.

The one-atomic-block encoding only samples states before and after the evaluation phase itself (see Figure 9(b)). Thus, it can only check for consistency properties at the delta-cycle boundary or at a timed-cycle boundary. But, as well as the thread-to-process and thread-to-atomic-block encodings are capable of, this encoding can be used to check for assertion violations.

To be synthesizable, the SystemC design under verification must not contain non-progressing delta cycles. To this end, in the encoding of the scheduler shown in Figure 4 we place the progress label `progress_DeltaCycle`. In SPIN, during the state space exploration, a label prefixed by `progress` has to be visited infinitely often in an infinite execution; violation of this property yields a non-progressing cycle.

To check for non-progressing delta cycles, the states before and after the execution of each thread in the evaluation phase must be sampled (or stored). Otherwise, a cycle of states in the state space that occurs during the exploration of the evaluation phase might go undetected, and can cause non-termination. Because the one-atomic-block encoding only samples states before and after the evaluation phase, one cannot use the encoding to check for non-progressing delta cycles.

As we have mentioned before, the atomic block surrounding the encoding of the thread body in the thread-to-process and the thread-to-atomic-block encodings is optional. By omitting the `atomic` keyword, these encodings also sample intermediate states during the thread executions. This sampling will allow us to verify temporal properties that speak about states and transitions in the thread body.

4.2 Search Behavior

The Promela encodings that we propose yield different search behavior. The differences are concerned with the number of transitions and states that need to be explored, the size of states, and the optimal applicability of optimizations that SPIN provides.

In the thread-to-process encoding each of the SystemC threads and scheduler is encoded as a separate process. The processes that encode the threads interact with the process that encodes the scheduler by means of rendezvous channels. The more SystemC threads in the design, the more processes and rendezvous channels we need, and thus the larger the size of the states is.

On the other hand, in the thread-to-atomic-block and the one-atomic-block encodings any SystemC design is encoded as a single process and use no channels for synchronizations. So the size of the state should be smaller than that of the thread-to-process encoding.

When there are only small variations of the values of global variables or of the local variables of processes, one can apply the collapse compression method [12] provided by SPIN to reduce the size of state in the thread-to-process encoding. That is, instead of replicating the complete description of all objects that are local to a process in each state, SPIN stores smaller state components separately and assigns small unique index to each of the components. The unique indices are then combined to form the global state. The division of state components usually consists of a descriptor for global data objects and a descriptor for each process. The collapse compression method has some run time overhead, but it should significantly reduce the memory consumed to store the state space. However, although the method is also applicable to the thread-to-atomic-block and one-atomic-block encodings, the reductions of memory consumption will not be as significant as its application to the thread-to-process encoding.

Comparing the thread-to-atomic-block and the one-atomic-block encoding, it is obvious that the one-atomic-block encoding will explore a smaller number of states than that of the thread-to-atomic-block encoding. However, the one-atomic-block encoding can explore more transitions than the thread-to-atomic-block encoding. Consider three threads t_1, t_2, and t_3 such that they are runnable in the evaluation phase and each thread only accesses variables local to the thread. Suppose that the first exploration in the evaluation phase goes with the sequence t_1, t_2, t_3. Now, if in the second exploration we end up with the prefix t_2, t_1, then with the thread-to-atomic-block encoding we can detect that we are visiting the state that we have visited before. That is, the resulting state of performing t_1, t_2 is the same as that of t_2, t_1. Thus, during the second exploration the search does not try to explore t_3. However, since the intermediate states between thread executions are not stored in the one-atomic-block encoding, the encoding has to explore t_3 during the second exploration.

SPIN implements a partial-order reduction (POR) [13] method that allows the verification search to explore only representative subsets of transition interleavings. POR in turn can reduce the number of visited states. However, the application of the POR method provided by SPIN to the proposed Promela encodings is suboptimal (or might not reduce the number of explored transitions at all) because the POR method loses the intrinsic structure of the SystemC designs. A non-interleaved transition in a SystemC design is a code fragment between two synchronization primitives that can make threads suspend themselves, and the static analysis implemented in the POR of SPIN does not recognize such a non-interleaved transition. Thus, the dependence relation between transitions computed by SPIN is rather coarse.

One can think of adding Promela code into the encodings such that the code contains static dependence relation between the SystemC non-interleaved transitions. However, this is not sufficient for checking safety properties because to check such properties we have to make sure that the reduction proviso condition described in [13] is satisfied at the level of SystemC non-interleaved transitions. This condition disallows the existence of a cycle of states in the state space such that there is a transition enabled in one of the state in the cycle, but that transition is never taken into the representative subsets of the explored transitions in any states in the cycle. To satisfy the reduction proviso condition, one needs to be able to access the states stored by the protocol analyzer generated by SPIN from the Promela encodings.

4.3 Other Existing Encodings

There have been several work on translating SystemC designs into Promela code; see [16, 21]. Similar to our thread-to-process encoding, these translations encode each SystemC thread as a Promela process, but, unlike ours, they embed the SystemC scheduler into the encoding of the synchronization primitives. Thus, there is no separate process for the SystemC scheduler.

The focus of these encodings is to verify SystemC designs at the high level of transaction level modeling (TLM). At that level of TLM, the delta cycles are typically not visible and communication between threads are usually through shared global variables instead of SystemC channels. Thus, unlike our encodings where we capture the full semantics of the SystemC scheduler, the existing encodings only considers the evaluation and the timed update phases of the scheduler. Moreover, since the evaluation phase of these encodings is encoded implicitly in the encoding of the synchronization primitives, these encodings cannot be used to detect non-progress delta cycles.

The work in [21] translates each SystemC thread into an automaton, which in turn is translated into a Promela process. The automaton itself is essentially the control-flow graph of the resulting Promela process. The translation from the automaton to the Promela process is not efficient because the resulting Promela process has to keep track of the program counter of the control-flow graph instead of the program counter of suspension points. Moreover, the encodings in [21] assume that time is discrete.

The Promela encoding presented in [16] is considered an improvement of the one in [21]. The aims of the encoding are twofold: (1) to reduce the number of interactions between the encodings of the threads, and (2) to make the application of SPIN optimizations optimal. The encoding manages to achieve (1) by exploiting the blocking statements of Promela, but is unclear about (2).

All the existing work mentioned above does not address the notion of under-approximation due to the restriction of input values.

5 Experimental Evaluation

We have developed a translator, called SC2PROMELA, that takes a SystemC design as an input and performs the Promela encodings proposed in this paper. The translator is a back-end extension of PINAPA [18] that extracts from a SystemC design information

that is useful for performing the encodings. Such information includes structures of module instances, their interconnections, and the abstract syntax tree of the threads.

Recall that the verification against the resulting Promela models are under-approximations, and so are the protocol analyzers (or pan) generated by SPIN from the Promela models. If the SystemC design reads inputs from the environment and the corresponding pan terminates gracefully (no time-out and no memory-out), and an assertion violation or a non-progressing cycle is detected, then we have found a real bug. Otherwise, if neither assertion violations nor non-progressing cycles are detected, and the search does not exceed the depth limit, then the design is safe under the assumption about the input values. In the experiments, we model input reading by selecting non-deterministically a value from a set of ten random values. For the unsafe benchmarks in our experiments, this assumption turns out to be sufficient to find the bugs.

Experimental evaluation setup. We compare the search behavior of our Promela encodings in terms of the number of stored states and the number of transitions, as well as the effectiveness of optimizations provided by SPIN. In the experiments we focus on checking for assertion violations and non-progressing delta cycles. We also compare the Promela encodings against ESST [6] that has been implemented in KRATOS [4]. ESST is not capable of identifying non-progressing delta cycles, and so the comparison with ESST is only about checking for assertion violations.

We run our experiments on a Linux box with Intel-Xeon DC 3GHz processor and 4GB of RAM. We set the time limit to 1000s, the memory limit to 2GB, and the search depth limit to 1,000,000.

Test cases. We use the benchmarks provided in [6] and derive new benchmarks `pipeline_bug`, `token_ring_bug` and `mem_slave_tlm_bug` from the respective safe benchmarks by introducing new assertions that can be violated. The `mem_slave_tlm_bug` family have been modified by introducing a memory overflow, that can only be reached by going through a long iteration in the model.

Results. Table 1 reports for the three encodings the following information: the number of stored states (# States stored), the number of explored transitions (# Transitions), the size of state in byte (# State size), and the run time in second (Time). We mark experiment results with TO and MO, for, respectively, out of time and out of memory. In the Time column we mark the time with * to indicate that the corresponding experiment reaches the search depth limit. For the checks for assertion violations (V), we use U and S for, respectively, unsafe (an assertion is violated) and safe (no assertion violations). For the checks for non-progressing cycles (NP), we considered only the thread-to-process and the thread-to-atomic-block encodings since the one-atomic-block encoding is not suitable for this check. We use NP and (?) to indicate the detection of a non-progressing cycle and an inconclusive results, respectively. We report neither the time needed to translate SystemC designs into Promela models nor the time needed to compile the resulting protocol analyzer since they are negligible.

Table 1 also reports the results of enabling and disabling some SPIN optimizations. In particular it reports the amount of collapse compression in percentage (Compression) and the number of explored transitions when the partial-order reduction is disabled (#

File	V	NP	ESST	thread-to-process					thread-to-atomic-block					one-atomic-block				
				# States stored	# Transitions	State size (byte)	Time (sec)	Compression (%)	# Transition No POR	# States stored	# Transitions	State size (byte)	Time (sec)	Compression (%)	# Transition No POR	# States stored	# Transitions	State size (byte)	Time (sec)	Compression (%)	# Transition No POR
bist_cell	S	(?)	0.50	8	17	204	0.00	-	-	17	8	17	188	0.00	-	17	6	15	188	0.00	-	15
kundu	S	(?)	1.10	1386	3175	268	0.03	-	3175	1386	3175	220	0.02	-	3175	405	2806	220	0.02	-	2806
kundu-bug-1	U	(?)	0.39	18	18	196	0.00	-	18	18	18	164	0.00	-	18	11	11	164	0.00	-	11
kundu-bug-2	U	(?)	1.10	121	181	268	0.00	-	181	121	181	220	0.00	-	181	36	105	220	0.00	-	105
mem_slave_tlm.1	S	(?)	2.79	93	310	364	0.00	-	310	93	310	324	0.01	-	310	6	2014	324	0.09	-	2014
mem_slave_tlm.2	S	(?)	13.60	137	363	364	0.00	-	363	137	363	332	0.00	-	363	6	20014	332	0.90	-	20014
mem_slave_tlm.3	S	(?)	152.19	181	416	372	0.01	-	416	181	416	332	0.01	-	416	6	200014	332	9.25	-	200014
mem_slave_tlm.4	S	(?)	42.80	225	469	372	0.01	-	469	225	469	340	0.00	-	469	6	2000014	340	93.90	-	2000014
mem_slave_tlm.5	S	(?)	773.14	269	522	380	0.01	-	522	269	522	340	0.01	-	522	6	20000014	340	967.00	-	20000014
mem_slave_tlm_bug.1	U	(?)	3.80	16	16	364	0.00	-	16	16	16	324	0.00	-	16	3	3	324	0.00	-	3
mem_slave_tlm_bug.2	U	(?)	20.80	24	24	364	0.00	-	24	24	24	332	0.00	-	24	3	3	332	0.01	-	3
mem_slave_tlm_bug.3	U	(?)	100.99	32	32	372	0.01	-	32	32	32	332	0.01	-	32	3	3	332	0.01	-	3
mem_slave_tlm_bug.4	U	(?)	347.87	40	40	372	0.01	-	40	40	40	340	0.00	-	40	3	3	340	0.01	-	3
mem_slave_tlm_bug.5	U	(?)	TO	48	48	380	0.01	-	48	48	48	340	0.00	-	48	3	3	340	0.01	-	3
pc_sfifo_1	S	(?)	0.30	5640647	7948186	180	* 72.50	51.85	7948186	7096367	9999436	152	* 66.50	55.04	9999436	TO	TO	TO	1000.09	TO	TO
pc_sfifo_2	S	(?)	0.30	10376427	12074486	196	* 91.50	44.51	12074486	12221427	14221346	160	* 85.20	46.89	14221346	7020793	8935572	160	* 63.70	66.93	DL
pipeline	S	(?)	77.09	MO	MO	MO	222.49	MO	MO	MO	MO	MO	MO	151.88	MO	MO	TO	TO	1000.03	TO	TO
pipeline-bug	U	(?)	132.99	76	76	700	0.00	-	76	76	76	616	0.01	-	76	26	26	616	0.01	-	26
token_ring.1	S	NP	0.10	107	216	156	0.00	-	216	107	216	128	0.00	-	216	TO	TO	TO	1000.03	TO	TO
token_ring.2	S	NP	0.10	274	414	228	0.00	-	414	274	414	184	0.01	-	414	TO	TO	TO	1000.07	TO	TO
token_ring.3	S	NP	0.20	640	904	316	0.00	-	904	640	904	248	0.01	-	904	TO	TO	TO	1000.02	TO	TO
token_ring.4	S	NP	0.20	1466	2102	420	0.02	-	2102	1466	2102	336	0.02	-	2102	TO	TO	TO	1000.09	TO	TO
token_ring.5	S	NP	0.30	3336	4964	532	0.06	80.07	4964	3336	4964	440	0.03	96.11	4964	TO	TO	TO	1000.05	TO	TO
token_ring.6	S	NP	0.40	7542	11650	668	0.15	35.76	11650	7542	11650	552	0.10	43.08	11650	TO	TO	TO	1000.08	TO	TO
token_ring.7	S	NP	0.50	16916	26976	820	0.40	24.30	26976	16916	26976	688	0.25	28.86	26976	TO	TO	TO	1000.01	TO	TO
token_ring.8	S	NP	0.70	37618	61566	988	1.02	19.75	61566	37618	61566	840	0.66	20.64	61566	TO	TO	TO	1000.08	TO	TO
token_ring.9	S	NP	1.39	82960	138652	1164	2.60	16.71	138652	82960	138652	1000	1.67	17.00	138652	TO	TO	TO	1000.02	TO	TO
token_ring.10	S	NP	2.40	181550	308666	1364	6.52	14.80	308666	181550	308666	1184	4.17	14.95	308666	TO	TO	TO	1000.01	TO	TO
token_ring.11	S	NP	4.50	394572	680408	1580	16.60	13.38	680408	394572	680408	1376	10.50	13.48	680408	TO	TO	TO	1000.09	TO	TO
token_ring.12	S	NP	4.10	852330	1487350	1812	39.20	12.66	1487350	852330	1487350	1592	26.00	12.30	1487350	TO	TO	TO	1000.04	TO	TO
token_ring.13	S	NP	7.50	1831304	3228180	2052	94.80	11.69	3228180	1831304	3228180	1824	63.30	11.36	3228180	TO	TO	TO	1000.10	TO	TO
token_ring_bug.1	U	NP	0.00	10	10	156	0.00	-	10	10	10	128	0.00	-	10	3	3	128	0.00	-	3
token_ring_bug.2	U	NP	0.00	14	14	228	0.00	-	14	14	14	184	0.00	-	14	3	3	184	0.01	-	3
token_ring_bug.3	U	NP	0.10	18	18	316	0.00	-	18	18	18	248	0.00	-	18	3	3	248	0.00	-	3
token_ring_bug.4	U	NP	0.10	22	22	420	0.00	-	22	22	22	336	0.00	-	22	3	3	336	0.00	-	3
token_ring_bug.5	U	NP	0.10	26	26	532	0.00	-	26	26	26	440	0.00	-	26	3	3	440	0.01	-	3
token_ring_bug.6	U	NP	0.20	30	30	668	0.01	-	30	30	30	552	0.00	-	30	3	3	552	0.00	-	3
token_ring_bug.7	U	NP	0.20	34	34	820	0.01	-	34	34	34	688	0.00	-	34	3	3	688	0.00	-	3
token_ring_bug.8	U	NP	0.30	38	38	988	0.00	-	38	38	38	840	0.01	-	38	3	3	840	0.02	-	3
token_ring_bug.9	U	NP	0.60	42	42	1164	0.00	-	42	42	42	1000	0.00	-	42	3	3	1000	0.00	-	3
token_ring_bug.10	U	NP	1.00	53	53	1364	0.00	-	53	53	53	1184	0.00	-	53	6	6	1184	0.00	-	6
token_ring_bug.11	U	NP	2.00	69	70	1580	0.00	-	70	69	70	1376	0.00	-	70	9	10	1376	0.01	-	10
token_ring_bug.12	U	NP	1.80	97	103	1812	0.00	-	103	97	103	1592	0.00	-	103	12	18	1592	0.01	-	18
token_ring_bug.13	U	NP	3.20	153	176	2052	0.01	-	176	153	176	1824	0.00	-	176	15	44	1824	0.02	-	44
toy	S	(?)	1.60	322250	355609	412	* 3.94	40.02	355609	414332	457217	332	* 3.52	49.70	457217	22908	45835	332	* 1.43	-	45835
toy-bug-1	U	(?)	1.60	142869	142870	412	* 5.21	48.50	142870	183119	183123	332	* 5.12	64.12	183123	11418	11422	332	* 4.08	-	11422
toy-bug-2	U	(?)	0.69	144458	144460	412	* 5.46	48.66	144460	185738	185742	332	* 5.34	64.29	185742	11457	11459	332	* 4.15	-	11459
transmitter.1	U	(?)	0.00	8	8	140	0.00	-	8	8	8	100	0.00	-	8	3	3	100	0.00	-	3
transmitter.2	U	(?)	0.00	12	12	204	0.00	-	12	12	12	156	0.00	-	12	3	3	156	0.00	-	3
transmitter.3	U	(?)	0.00	16	16	284	0.00	-	16	16	16	220	0.00	-	16	3	3	220	0.00	-	3
transmitter.4	U	(?)	0.00	20	20	380	0.00	-	20	20	20	300	0.00	-	20	3	3	300	0.00	-	3
transmitter.5	U	(?)	0.00	24	24	500	0.00	-	24	24	24	396	0.00	-	24	3	3	396	0.00	-	3
transmitter.6	U	(?)	0.10	28	28	628	0.00	-	28	28	28	508	0.00	-	28	3	3	508	0.01	-	3
transmitter.7	U	(?)	0.09	32	32	772	0.01	-	32	32	32	636	0.01	-	32	3	3	636	0.00	-	3
transmitter.8	U	(?)	0.09	36	36	932	0.00	-	36	36	36	780	0.00	-	36	3	3	780	0.00	-	3
transmitter.9	U	(?)	0.10	40	40	1116	0.01	-	40	40	40	940	0.00	-	40	3	3	940	0.01	-	3
transmitter.10	U	(?)	0.10	56	57	1308	0.00	-	57	56	57	1124	0.00	-	57	9	10	1124	0.00	-	10
transmitter.11	U	(?)	0.20	66	69	1516	0.00	-	69	66	69	1316	0.00	-	69	9	12	1316	0.01	-	12
transmitter.12	U	(?)	0.39	83	91	1740	0.01	-	91	83	91	1524	0.00	-	91	9	18	1524	0.01	-	18
transmitter.13	U	(?)	0.20	116	137	1988	0.00	-	137	116	137	1748	0.01	-	137	9	42	1748	0.02	-	42

Table 1: Results for the experimental evaluation of the different Promela encodings.

Transitions No POR). In the Compression column we use $-$ to denote the case when the collapse compression is unsuccessful.

As shown on Table 1, our Promela encodings outperforms ESST in verifying the unsafe benchmarks. ESST performs over-approximations that involve expensive sym-

bolic computations. These over-approximations often require many refinements. This comparison has shown that the Promela encodings are efficient for formal bug finding, and thus they can fruitfully complement ESST in the verification of SystemC designs.

Comparisons on the safe benchmarks are irrelevant because ESST performs over-approximations, while the Promela encodings perform under-approximation. One interesting fact is shown by the experiment on the `pipeline` benchmark. On this benchmark our Promela encodings yield either out of memory or out of time. In this benchmark there are a huge number of possible thread interleavings that need to be explored to check for assertion violations. The size of input values also affect the number of thread interleavings. ESST, on the other hand, has been equipped with a partial-order reduction technique that works at the transition atomicity level of SystemC designs [6]. Thus, ESST explores a less number of thread interleavings. Moreover, due to its over-approximations, ESST is not affected by the size of input values.

Table 1 shows that on most of the benchmarks the thread-to-process and the thread-to-atomic-block encodings store the same number of states and explore the same number of transitions. This fact shows that the synchronization mechanism using token passing through rendezvous channels does not interact negatively with the search behavior. However, in accordance with our analysis, the size of state for the thread-to-process encoding is larger than that of the thread-to-atomic-block encoding.

Regarding the one-atomic-block encoding, Table 1 shows that, for the safe benchmarks, the thread-to-atomic-block encoding stores more states but less transitions than the one-atomic-block encoding. These results are in accordance with our analysis in the previous section. For the unsafe benchmarks, like the family of `token_ring_bug`, the one-atomic-block encoding outperforms the thread-to-atomic-block encoding in terms of the number of stored states and the number of explored transitions. In these benchmarks the assertion violation is found in the first delta cycle. In principle, when the assertion violation occurs after several delta cycles, and there is no guarantee that the delta cycles are progressing, then the one-atomic-block encoding might go out of time.

For the `token_ring` safe family, although we restrict the number of input values, the one-atomic-block encoding goes out of time since the delta cycle does not progress. Hence, because there are no intermediate states sampled in the evaluation phase, the protocol analyzer does not know how to stop the search.

The application of the collapse compression on most of the benchmarks is unsuccessful. That is, due to the overhead incurred by the collapse compression, the actual memory usage for the states is larger than if the collapse compression is disabled. Upon close inspection, it turns out that the threads in most of the benchmarks do not have variables that are local to the threads, but use heavily global variables in their computations. These global variables encode the channels and other global data that are used by the threads to communicate with each other. As mentioned in the previous section, the global data object constitutes one portion in the state component. Moreover, in those benchmarks there is a large number of different variations of the global data objects. Consequently, (1) the collapse compression is not effective for the benchmarks, and (2) when the collapse compression is successful, the differences of compression between the thread-to-process and the thread-to-atomic-block are not significant.

In the family of `toy-bug` benchmarks the application of the collapse compression is effective to reduce the memory consumption. In fact in this family of benchmarks the threads use variables that are local to them in their computations, and in particular, in the thread-to-process encoding, there are only small variations of process states. Thus, with the thread-to-process encoding the actual memory usage can be compressed to less than 49% of the original memory usage, while with the thread-to-atomic-block and one-atomic-block encodings the compression is larger than 64%.

Our experiments confirm the analysis in the previous section that the partial-order reduction provided by SPIN is not effective to reduce the number of explored transitions. In fact, for all encodings and for each benchmark, whether or not the reduction is enabled, the numbers of explored transitions are the same.

We have not been able to empirically compare our Promela encodings with the existing ones described in the previous section. There seems to be a regression that breaks the tool used to generate the encodings.[2] The benchmarks used in [16] are essentially the same as the family of `transmitter` benchmarks used in our work. For `transmitter` of size 13, the encoding in [16] explored 8306 states, while our encodings store no more than 116 states and explore no more than 137 transitions.

On Table 1, column NP, the thread-to-process and the thread-to-atomic-block encodings find non-progressing delta cycles in some benchmarks, in particular in the families of `token_ring_bug` benchmarks. Such bugs were not known beforehand and were not addressed in [5].

Experiments on some of the benchmarks reach the search depth limit. In those benchmarks there are counters that range over integer values and are independent of the input values. It turns out that we can perform more under-approximation on those benchmarks by making the counters range over byte values. In so doing, we are able to verify that the benchmark `pc_sfifo_1` has a non-progressing cycle.

All the benchmarks and the scripts for reproducing the performed experiments can be downloaded at: `http://es.fbk.eu/people/roveri/tests/spin2011`.

6 Related Work

There have been several work on to the formal verification of SystemC designs [5, 6, 8, 9, 11, 14, 17, 21]. We have already discussed in Section 4.3 the work in [16, 21], that are the most closely related to our work.

SystemC verification via translation to other languages have been reported in a number of papers. In [5, 9] SystemC designs are translated into sequential C programs that can be checked by existing software-model checkers. It is also shown in [5] that complete verification of such sequential C programs is ineffective due to the size of the encoded scheduler and the need for precise information on the scheduler states.

Bounded model checking (BMC) have been applied for bug hunting in SystemC designs, as described in [9]. Unlike our work, the work in [9] only addresses untimed SystemC designs, and needs to determine the unwinding depth even though all loops in the input SystemC design are bounded. Similar to our work, the use of BMC is another

[2] Personal communication with Matthieu Moy.

kinds of under-approximation. In addition to bug hunting, the work in [9] also proposes a complete technique based on induction.

An encoding of SystemC designs into networks of timed automata is presented in [11]. The resulting timed automata are then verified with UPPAAL [1]. The Lussy tool chain described in [17] translates SystemC designs into sets of parallel automata. The automata themselves are in turn translated into different models, like SMV models and Lustre model. The experimental results in [11,17] demonstrate limited scalability of the proposed techniques, related to the symbolic techniques used to check the automata.

The work in [14] translates SystemC designs into labeled Kripke structures. However, the translation abstracts away the scheduler, that is, the scheduler is encoded implicitly in each of the threads, and the encoding only considers the evaluation phase.

In [5] we have proposed a technique, called ESST, for the verification of SystemC designs. The technique combines explicit model checking technique to deal with the scheduler and symbolic technique, based on the lazy predicate abstraction [10], to deal with the threads. In [6] we apply a partial-order reduction method to ESST. Although ESST shows promising results, the bottleneck caused by slow refinements make it inefficient for a quick bug finding, as confirmed by our experimental evaluation.

Recent work on monitoring SystemC properties is discussed in [20]. The work needs to modify the SystemC scheduler and instrument the design to observe desirable properties during simulations. However, SystemC simulations do not explore all possible schedules, and so the monitoring process can miss some bugs.

Weakly related to our work is the tool Scoot described in [3]. Scoot extracts from a SystemC design a flat C++ model that can be analyzed by SATABS [7]. The SystemC scheduler itself is included in the flat model. Scoot has been used to synthesize a SystemC scheduler that can speed up simulations by performing race analysis [2]. Scoot has also been used to perform run-time verification. Other work on scalable testing of SystemC designs by using partial-order reduction is described in [15].

7 Conclusions and Future Work

We have proposed three Promela encodings of SystemC designs, and have provided a thorough analysis of them in terms of the kinds of property they allow to verify, and in terms of the search behaviors during the verification. The empirical results obtained from our experiments on these encodings support the analytical comparison between the encodings. The results of experimental evaluation also show that our technique is effective and efficient in finding bugs, can detect non-progressing delta cycles, and can complement existing symbolic verification techniques based on lazy abstraction.

As future work, we will investigate the applicability of test generation to relieve the problem of range restrictions. We will address the problem of redundant thread interleavings. In fact, the POR techniques implemented in SPIN turn out to be largely ineffective in this setting.

Input values can also play a role in causing non-progressing delta cycles. As we restrict the range of input values, our technique can miss such non-progressing cycles. We would like to investigate the applicability of automatic techniques for checking non-termination.

Another direction of future work is to combine under-approximation and over-approximation techniques for full verification of SystemC designs. Finally, we would like to support more SystemC (and C++ in general) features, to allow our proposed technique to be tested on a larger set of benchmarks.

References

1. G. Behrmann, A. David, K. G. Larsen, J. Håkansson, P. Pettersson, W. Yi, and M. Hendriks. UPPAAL 4.0. In *QEST*, pages 125–126. IEEE, 2006.
2. N. Blanc and D. Kroening. Race Analysis for SystemC using Model Checking. In *ICCAD*, pages 356–363. IEEE, 2008.
3. N. Blanc, D. Kroening, and N. Sharygina. Scoot: A Tool for the Analysis of SystemC Models. In *TACAS*, volume 4963 of *LNCS*, pages 467–470. Springer, 2008.
4. A. Cimatti, A. Griggio, A. Micheli, I. Narasamdya, and M. Roveri. Kratos: A Software Model Checker for SystemC. In *CAV*, 2011. To appear.
5. A. Cimatti, A. Micheli, I. Narasamdya, and M. Roveri. Verifying SystemC: a Software Model Checking Approach. In *FMCAD*, pages 51–59, 2010.
6. A. Cimatti, I. Narasamdya, and M. Roveri. Boosting Lazy Abstraction for SystemC with Partial Order Reduction. In *TACAS*, LNCS, pages 341–356, 2011.
7. E. M. Clarke, D. Kroening, N. Sharygina, and K. Yorav. SATABS: SAT-Based Predicate Abstraction for ANSI-C. In *TACAS*, volume 3440 of *LNCS*, pages 570–574. Springer, 2005.
8. D. Große and R. Drechsler. CheckSyC: An Efficient Property Checker for RTL SystemC Designs. In *ISCAS (4)*, pages 4167–4170. IEEE, 2005.
9. D. Grosse, H. Le, and R. Drechsler. Proving Transaction and System-level Properties of Untimed SystemC TLM Designs. In *MEMOCODE*, pages 113 –122, 2010.
10. T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy Abstraction. In *POPL*, pages 58–70, 2002.
11. P. Herber, J. Fellmuth, and S. Glesner. Model Checking SystemC Designs using Timed Automata. In *CODES+ISSS*, pages 131–136. ACM, 2008.
12. G. J. Holzmann. Software Model Checking with SPIN. *Adv. in Comp.*, 65:78–109, 2005.
13. G. J. Holzmann and D. Peled. An improvement in formal verification. In *Proceedings of the 7th IFIP WG6.1 International Conference on Formal Description Techniques VII*, pages 197–211, London, UK, UK, 1995. Chapman & Hall, Ltd.
14. D. Kroening and N. Sharygina. Formal Verification of SystemC by Automatic Hardware/-Software Partitioning. In *MEMOCODE*, pages 101–110. IEEE, 2005.
15. S. Kundu, M. K. Ganai, and R. Gupta. Partial Order Reduction for Scalable Testing of SystemC TLM Designs. In *DAC*, pages 936–941. ACM, 2008.
16. K. Marquet, B. Jeannet, and M. Moy. Efficient Encoding of SystemC/TLM in Promela. Technical report, Verimag, 2010. Verimag Research Report no TR-2010-7.
17. M. Moy, F. Maraninchi, and L. Maillet-Contoz. LusSy: A Toolbox for the Analysis of Systems-on-a-Chip at the Transactional Level. In *ACSD*, pages 26–35. IEEE, 2005.
18. M. Moy, F. Maraninchi, and L. Maillet-Contoz. Pinapa: An Extraction Tool for SystemC Descriptions of Systems-on-a-Chip. In *EMSOFT*, pages 317–324. ACM, 2005.
19. D. Tabakov, G. Kamhi, M. Y. Vardi, and E. Singerman. A Temporal Language for SystemC. In *FMCAD*, pages 1–9. IEEE, 2008.
20. D. Tabakov and M. Vardi. Monitoring Temporal SystemC Properties. In *MEMOCODE*, pages 123–132, 2010.
21. C. Traulsen, J. Cornet, M. Moy, and F. Maraninchi. A SystemC/TLM Semantics in Promela and its Possible Applications. In *SPIN*, volume 4595 of *LNCS*, pages 204–222, 2007.