Software Model Checking SystemC

Alessandro Cimatti, Iman Narasamdya, and Marco Roveri
Fondazione Bruno Kessler, Italy.
Via Sommarive 18, I-38050, Trento, Italy
{cimatti,narasamdya,roveri}@fbk.eu

***Abstract*—SystemC is an increasingly used language for writing executable specifications of Systems-on-Chip. The verification of SystemC, however, is a very difficult challenge. Simulation features great scalability, but can miss important defects. On the other hand, formal verification of SystemC is extremely hard because of the presence of threads, and the intricacies of the communication and scheduling mechanisms.**

In this paper, we explore formal verification for SystemC by means of software model checking techniques, that have demonstrated substantial progress in recent years. We propose an accurate model of SystemC, and three complementary encodings of SystemC to finite-state processes, sequential, and threaded programming models.

We implemented the proposed approaches in a tool-chain, and we carried out a thorough experimental evaluation, using several benchmarks taken from the literature on SystemC verification, and experimenting with different state-of-the-art software model checkers. The results clearly show the applicability and efficiency of the proposed approaches. In particular, the results show the effectiveness of the threaded and of the finite-model encodings to prove and disprove properties, respectively.

***Index Terms*—SystemC, sequentialization, software model checking, CEGAR.**

I. Introduction

SystemC [3] is a de-facto standard language for writing executable designs of System-on-Chips. A high-level executable design allows to efficiently simulate the behaviors of the SoC before synthesizing the RTL hardware description. The verification of SystemC designs is of paramount importance, to reveal errors in early stages of the development flow, and to prevent expensive propagation down to the hardware.

Formal verification of SystemC has recently gained significant interests. Compared to simulations, formal verification provides full coverage, and thus may enhance the reliability and the robustness of the design. However, formal verification of SystemC designs poses two main challanges: to handle SystemC rich constructs and to achieve high-performance analysis. The latter is due to the fact that SystemC has a complex computation model, that can be viewed as a multi-threaded program with a specific scheduling policy [4], and complex communication primitives. For this reason, the previous formal approaches [1], [5]–[10] either show poor performances and scalability, or abstract away significant semantic aspects of SystemC.

In this paper, we propose a novel approach to SystemC verification. Our work focuses on achieving high-performance analysis, while reusing existing front-end components to handle the SystemC constructs. The characterizing feature of our work is the precise modeling of the complex features of SystemC, such as the scheduler and the communication primitives. The idea is to leverage recent advances in software model checking (SMC) techniques [11], that have demonstrated significant successes in the analysis of complex software, and have enabled their use in industrial settings [12], [13].

We explore three different SMC approaches to the verification of SystemC designs. The first approach converts a SystemC design into a *finite-state model* specified in PROMELA [14], and uses the SPIN explicit-state model checker to carry out the analysis. The conversion has several variants, each coming with a generic and accurate encoding of the SystemC scheduler, and an encoding of each SystemC thread. Given the infinite nature of SystemC, the modeling into a finite-state model requires to restrict the space of values that can be acquired from the environment, and to bound the search space. As such, this approach provides only an underapproximation, and is only intended for bug finding. This approach, unlike simulation-based verification, provides full coverage since it explores all possible thread interleavings.

The second approach, referred to as *sequentialization*, encodes SystemC designs into sequential programs: SystemC threads are encoded as functions, called by a top-level routine that mimics the SystemC scheduler. The resulting sequential programs can then be analyzed by any state-of-the-art model checker for sequential programs (e.g. [13], [15]–[19]).

In the third approach, called *Explicit-Scheduler/Symbolic-Threads* (ESST) [1], [20], a SystemC design is mapped onto a *shared-variable* multi-threaded program with *cooperative* scheduling and *mutually-exclusive* thread executions. ESST analyzes each thread using lazy predicate abstraction [16], and orchestrates the overall verification by the direct execution of the SystemC scheduler, using techniques similar to explicit-state model checking. Compared to sequentialization, the scheduler is no longer part of the program being analyzed, but is part of the ESST algorithm itself.

The flow described above was implemented in a comprehensive tool chain, where PINAPA [21] is used as a front end. The translations were implemented in several modules encoding SystemC designs into finite state PROMELA models, sequential C programs, and threaded C programs. The tool chain connects to SPIN and to several existing software model

Some of the material presented in this paper appears in preliminary version in [1] and in [2]. This manuscript presents a new tool-chain for the formal verification of SystemC based on one unifying formal framework, a uniform description of the proposed encodings, a more extensive and deeper experimental evaluation, and the new optimizations added to KRATOS to improve its efficiency.

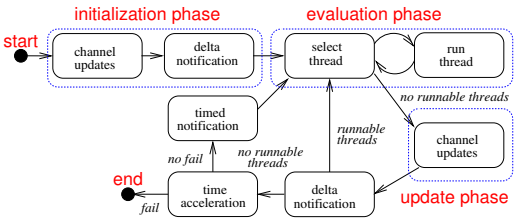


Fig. 1. The SystemC scheduler.

checkers for C [17], [18], [22]–[25]. Substantial work was required to specialize the generic ESST approach of KRATOS to the case of SystemC verification.

We have performed a thorough experimental evaluation of the proposed tool chain, using a significant set of benchmarks. The results show that ESST is effective for proving safety properties, much more than simple sequentialization, and is fruitfully complemented by the encodings into finite-state models for bug finding.

The structure of this paper is as follows. Sec. II provides a background on SystemC. Sec. III provides an overview of the approach, and describes the SMC techniques. Sec. IV, V, and VI, respectively, discuss the encodings into finite-state processes, sequential programs, and multi-threaded programs. Sec. VII discusses related works. Sec. VIII shows our experimental evaluation. Finally, Sec. IX draws the conclusions and outlines some future work.

II. SYSTEMC

SystemC is a C++ library that consists of a core language for modeling the components of a system design and their interconnections, and a simulation kernel (or a scheduler) for fast simulations of the design. The core language models system components by means of modules (or C++ classes) and abstracts communication between modules by means of channels. SystemC provides several primitive channels such as signal, mutex, semaphore, and queue. A module can have one or more thread definitions that model the parallel behavior of the system design. The core language provides general-purpose events as synchronization mechanisms between threads.

The SystemC scheduler runs the threads during simulations. Following the SystemC semantics in [4], the scheduler consists of several phases as shown in Figure 1. In the *initialization phase* all channels are initialized. The scheduler then enters the *evaluation phase* where it executes all runnable threads while postponing the materialization of channel updates performed by the threads. This phase employs a cooperative scheduling policy with mutually-exclusive thread execution. When there are no more runnable threads, the scheduler goes into the *update phase* where it materializes all channel updates postponed during the evaluation phase. An evaluation phase followed by an update phase constitutes a *delta cycle*.

A thread, during its execution, can perform delayed event notifications. That is, the involved events will be notified at some time in the future, including at the *delta notification*. The materializations of channel updates also often require the events associated with the updated channels to be notified at the delta notifications. In turn, all threads that are waiting for the notified events or are sensitive to the channels whose associated events are notified become runnable. If, after the delta notification, there are runnable threads, the scheduler goes back to the evaluation phase to run them. Otherwise, it accelerates the simulation time to the nearest time point where there exist events to be notified. These events are then notified at the *timed notification*. Similar to the delta notification, some waiting threads can become runnable after the timed notifications, and thus the scheduler has to go back to the evaluation phase to run them. If there are no more events to be notified at some future time, denoted in Figure 1 by failure in time acceleration, then the simulation ends.

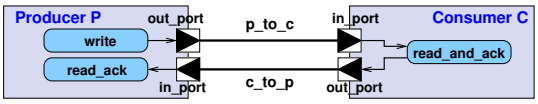


Fig. 2. A `Producer-Consumer` design.

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

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

Fig. 3. SystemC code for design of Figure 2.

An example of a design with two components, `Producer` and `Consumer` is depicted in Figure 2, while its SystemC implementation is reported in Figure 3. The constructor of `Producer` specifies two threads: `write` for writing value to `Consumer` through the channel bound to its output port `out_port`, and `read_ack` for reading the acknowledgment value sent by `Consumer` through the channel bound to its input port `in_port`. 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 `in_port` (`sensitive << in_port`), and is not runnable at the beginning of the simulation (`dont_initialize()`).

In this example, the thread `write` first waits until the next delta cycle (`wait(SC_ZERO_TIME)`) before entering the loop that feeds values to `Consumer`. Every time it writes a value to `Consumer`, it suspends itself waiting for the notification of event `e` (`wait(e)`). The structure and behavior of `Consumer` can be described similarly to that of `Producer`. The main function `sc_main` instantiates one `Producer`, one `Consumer`, binds their ports to the appropriate signal channels, and then starts the simulation (`sc_start()`).

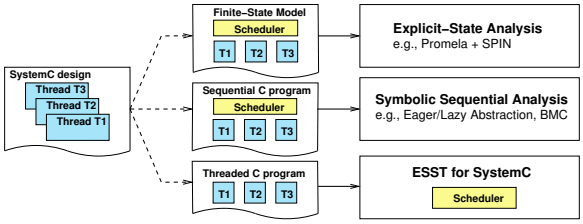


Fig. 4. Approaches to SystemC verification.

III. Overview of the Approach

The proposed verification flow, depicted in Figure 4, takes in input a SystemC design, and generates various descriptions that activate different back-end techniques. In this section we first overview the flow, and then we analyze the features of each back-end technique.

SystemC Translations. Our verification flow relies on the translation of SystemC designs into forms that can be analyzed by existing software model checking techniques. In the first approach, we translate the SystemC design into a finite-state model. The resulting model can then be analyzed using explicit-state model checking techniques. In the second approach, we translate the SystemC design into a sequential C program such that there is a one-to-one correspondence between the threads in the design and the functions in the C program, and thread executions are modeled as function calls. The resulting sequential C program also contains an encoding of the SystemC scheduler. Such a translation enables the use of state-of-the-art symbolic model checking techniques for sequential software, like abstraction-based analyses and bounded model checking. In the third approach, we verify SystemC designs via the ESST technique. We map the threads in the SystemC design onto a set of threads, that are implicitly interpreted by the ESST engine that has been specialized to SystemC. That is, the scheduler of ESST is instantiated to the SystemC scheduler.

Assumptions: Our approaches assume that the number of threads and events are known before the simulation starts, and there is no dynamic creations of objects or threads. We also assume that all function calls in the design can statically be inlined. In our experience, most of SystemC designs satisfy these assumptions. Thus, these assumptions do not severely limit the applicability of our approaches. To enable the use of existing software model checking techniques, we refrain from encoding SystemC constructs to the constructs that most model checkers cannot deal with or cannot handle effectively. For example, most model checkers either do not support arrays or reason about them in an ineffective way. Thus, although encodings into arrays can result in succinct encodings, we would rather encode arrays into multiple variables. Most software model checkers cannot deal with floating-point or real numbers, so we assume that the time delays specified in the SystemC designs can be converted into integers. Additionally, to use the ESST algorithm, we assume that the arguments passed to the synchronization functions are constants.

Explicit State Model Checking of Software. Explicit-state model checking techniques have been widely used in the formal verification of software. These techniques, unlike simulation-based verification, provide full coverage because they exercise all possible input values and, on multi-threaded software, explore all possible thread interleavings.

One of the most successful explicit-state model checkers is SPIN [26]. It is used for verifying the correctness of software models in the form of (non-deterministic) finite-state models written in PROMELA. Besides detecting assertion violations, SPIN can detect non-progressing cycles in PROMELA models.

SPIN provides several optimizations techniques to cope with the state explosion problem. For example, to effectively explore process interleavings, SPIN implements a partial-order reduction method [27]. To reduce memory consumption, SPIN implements statement merging that combines sequences of transitions within the same process into a single transition to avoid the creation of intermediate states. It also implements a state compression method, called collapse compression [14]. This method avoids replicating a complete description of all local components of the system state by storing only smaller state components separately, associating each state component with a unique id, and combining the unique identifiers to form the global state descriptor. Collapse compression has some run time overhead, but should significantly reduce the memory consumed to store the state space.

Explicit-state techniques are unable to deal with the infinite nature of the state space of the threads. Unfortunately, a finite-state abstraction is not easily obtained in general. Thus, these techniques are typically oriented towards bug finding.

Symbolic Model Checking of Sequential Software. Symbolic model checking is the most prominent approach to verifying software due to its ability to counter the infinite nature of the state space of software. Two main paradigms have emerged: counter-example guided abstraction refinement (CEGAR) [15], and bounded model checking (BMC) for software [18].

The CEGAR approaches start with a coarse abstraction of the program, and successively refine the abstraction using the information extracted from spurious counter-examples. The abstraction provides an effective way to cope with the potentially infinite state space, while the abstraction-refinement loop allows to incrementally re-introduce the details required to generate proofs of correctness. The most promising CEGAR-based techniques include the lazy predicate abstraction [16], [22], the lazy abstraction with interpolants [19], and the eager abstraction [13], [17].

The lazy predicate abstraction is based on the on-the-fly construction and analysis of an *abstract reachability tree* (ART). An ART describes the reachable abstract states of the program and is obtained by unwinding the control-flow graph (CFG) of the program. An ART node is a pair (l, φ) where l is a control-flow location, and *region* φ is a formula representing a set of data states. The verification of safety properties is reduced to checking the reachability of some error locations in the CFG. The lazy predicate abstraction expands an ART node by computing the abstract post-conditions of the node's region with respect to the operation of the unwound CFG edge and a finite set of predicates over program variables.

The expansion of an ART node (l, φ) is stopped if φ is unsatisfiable or if the node is *covered* by another node (l, φ') such that φ is subsumed by φ'. During the ART construction, when the expansion reaches an error node (a node with an error location), the path from the root to the reached node is checked for its feasibility. If it is feasible, then the path is a counter-example for the safety property. Otherwise, the path is analyzed to discover new predicates to be used for refining the ART [28]. The ART is then reconstructed by taking into account the newly discovered predicates. An ART is *complete* if no further node expansion is possible, and is *safe* if it additionally has no error node with a satisfiable region. A safe ART is a proof that the program is safe.

The lazy abstraction with interpolants, similarly to the lazy predicate abstraction, is also based on the construction of an ART. But, it avoids the expensive computations of abstract post-conditions by using directly the interpolants generated from the unfeasible paths as regions for the nodes.

Eager abstraction techniques typically extract an abstraction of the sequential program in the form of *Boolean program* [29] by using predicate abstraction techniques. Unlike sequential programs in general, the reachability problem of Boolean programs is decidable [29]. Boolean programs containing no recursive function calls can be directly analyzed using "off-the-shelf" finite-state verification techniques [14], [30], [31]. If the Boolean program is safe, so is the original program. Otherwise, new predicates are discovered and the verification is iterated.

The bounded model checking for sequential programs unwinds the CFG for a fixed number of steps [18]. Given a program, an error location, and an integer k, one constructs a constraint which is satisfiable iff the error location is reachable within k steps. Satisfiability of this constraint can be checked by any constraint solver. Typically, in this context, the bound k refers to the number of CFG unwindings or to the number of loop unrollings. This approach is sound and complete, but only up to the given bound.

***Explicit-Scheduler/Symbolic-Threads* (ESST).** ESST [1] has been introduced as an effective technique for the verification of multi-threaded software with *cooperative* (or *non-preemptive*) scheduling and *mutually-exclusive* thread executions: the scheduler executes at most one thread at a time, and when it gives the control to a thread to run, it waits until the thread suspends itself and gives back the control. ESST analyzes multi-threaded software in the form of *threaded program* that consists of a set of sequential programs (also called threads) that communicate with each other through shared global variables, and communicate with the scheduler (e.g., updating and querying scheduler states) through a set of *primitive functions* provided by ESST.

ESST is a CEGAR-based technique, and combines explicit and symbolic model checking techniques. It analyzes each thread with the lazy predicate abstraction, as if the thread were a sequential program. The whole verification is orchestrated by the direct execution of the scheduler using techniques similar to explicit-state model checking. That is, ESST keeps track of the state of the scheduler explicitly, and includes the scheduler as part of the verification algorithm. To allow the direct execution of the scheduler, ESST needs precise scheduler states, and thus it requires that the arguments passed to the primitive functions are constants. Both the scheduler and the set of primitive functions are left abstract, but the scheduler is required to exhibit a cooperative scheduling policy.

The ESST algorithm is based on the construction and analysis of an *abstract reachability forest* (ARF) that describes the reachable abstract states of the multi-threaded program. An ARF consists of connected ART's, each of which is obtained by unwinding the CFG of the running thread. For a program with N threads, an ARF *node* 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 location and the region of thread i, φ is the region of global variables, and S is the scheduler state. Regions are formulas describing the values of program variables, while the scheduler state is a mapping from scheduler variables to concrete values. The scheduler state, in particular, maintains information about the threads, e.g., the states of the threads. An ARF is constructed by unwinding the CFGs of threads, and by executing the scheduler. Each ART in the ARF is constructed using the lazy predicate abstraction as for the case of sequential programs. In particular, when the operation of the unwound CFG edge involves a call to a primitive function, then ESST has a primitive function executor that takes as inputs the scheduler state and the call to a primitive function, and returns the updated scheduler state obtained from executing the function.

Given a node $(\langle l_1, \varphi_1 \rangle, \ldots, \langle l_i, \varphi_i \rangle, \ldots, \langle l_N, \varphi_N \rangle, \varphi, S)$, such that there are no running threads, ESST then executes the scheduler. The scheduler itself is essentially a function that takes the scheduler state S as an input and outputs a set $\{S'_0, \ldots, S'_M\}$ of scheduler states representing all possible schedules. Each of these states forms an ARF node $(\langle l_1, \varphi_1 \rangle, \ldots, \langle l_i, \varphi_i \rangle, \ldots, \langle l_N, \varphi_N \rangle, \varphi, S'_{\mathbf{j}})$, that becomes the root of new ART of the subsequent running thread. Coverage checks and refinements in ESST are similar to that of the lazy predicate abstraction. In particular, the subsumption checks are done thread-wise and require the scheduler states to match. See [1] for the details of coverage checks and refinements.

In [20] ESST has been enhanced with *partial-order reduction* (POR) [32]–[34] to alleviate the problem of exploring a large number of redundant thread interleavings. POR exploits the commutativity of concurrent transitions that result in the same state when they are executed in different orders.

IV. Encoding into Finite-State Models

In this section we show how to encode SystemC designs into finite-state models. The encoding of a SystemC design is obtained by combining a generic encoding of the scheduler and an encoding of each thread in the design. In particular, we describe three different encodings of SystemC designs in Promela. All of these encoding variants follow the full semantics of SystemC by modeling precisely the SystemC scheduler, but they differ in the way the scheduler-threads synchronizations are modeled, and in the granularities of state sampling. As a result, they feature different characteristics in

terms of the properties that can be verified, but all of them are suitable for verifying program assertions. All the PROMELA encodings exhibit an under-approximation: we restrict the potentially infinite input values to a finite set of values.

For presentation, we will use the example introduced in Section II. Although the example only uses a small subset of SystemC features, the example is sufficient to describe our encodings. Other SystemC features that are not present in the example can be dealt with in a similar way.

A. Basic Building Blocks

1) Encoding Modules, Events, and Channels: We encode module instances by opening up their definitions. Members of module instances, global variables or variables accessible by multiple threads are encoded into a set of global variables in the PROMELA model. Each thread function (functions declared by `SC_THREAD` or `SC_METHOD`) of the instances is encoded either as PROMELA inline code or as PROMELA process, depending on the chosen encoding. For example, a member variable x and a thread function T of a module instance M are encoded, respectively, into a global variable M_x and into inline code or process M_T. In what follows we drop the module instance name prefix when it is clear from the context.

We model an event e by two global variables e_state and e_time. The variable e_state keeps track of the state of e, and ranges over the enumerations {`NOTIFIED`, `NOTIFIED_DELTA`, `NOTIFIED_TIME`, `NONE`}. The value `NOTIFIED` indicates that the event is notified. The values `NOTIFIED_DELTA` and `NOTIFIED_TIME` indicate that the event will be notified at, respectively, the delta notification and the timed notification, while the value `NONE` indicates there is no notification of the event. The variable e_time keeps track of the time delay of the event notification.

Similar to the encoding of module instances, channel instances are encoded by opening up their definitions. We exemplify the encoding of channel instances by the encoding of signal instances. Other kinds of channel can be encoded in a similar way. 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. Following the definition of signal, we associate each signal s with a Boolean variable s_req_up that will be set to `true` after every write to s to indicate a request for an update in the update phase. We also associate s with an event e_s that is used to notify waiting threads that are sensitive to the value update of s.

For each signal s, we have the encoding of its update function as PROMELA inline code s_update, as shown in Figure 5. The code is executed in the update phase, and accounts for updating the content of s_old with the value of s_new when they are not equal, and in such a case the event e_s is set to be notified at the delta notification.

2) Encoding Thread Bodies: The body of each SystemC thread T is encoded as PROMELA inline code T_body. We associate each thread T with two global variables: T_pc keeps track of the program counter of the thread, and T_state keeps track of the state of the thread. The values of T_state range

```
0 #define ITE(C,T,E) { if ::C -> T; ::else -> E; fi }
1
2 inline p_to_c_update() {
3  ITE(p_to_c_new != p_to_c_old,
4      p_to_c_old = p_to_c_new; e_p_to_c = NOTIFIED_DELTA, skip)
5 }
```

Fig. 5. PROMELA encoding of channel update function.

over the enumerations {`RUNNING`, `RUNNABLE`, `WAITING`}, whose meanings are obvious.

To model thread suspensions and resumption, we add code labels in the body such that the labels mark the locations from which the thread has to resume its execution. We then start the encoding of the body with a jump table that directs the execution to the resuming location. The passing of control from the thread T to the scheduler is modeled by the inline code $T\_suspend$. Because the thread-scheduler synchronizations are modeled differently in the encodings, so are the expansions of $T\_suspend$.

When the thread yields the control to the scheduler, it sets the thread state to `WAITING`, sets the program counter to the code label following $T\_suspend$, and executes $T\_suspend$. On receiving the control back, with the help of the jump table, the thread resumes its execution from the location it saved in its latest suspension. Figure 6 shows the encoding of the thread `write` of `Producer p`.

```
0  inline p_write_body() {
1    if // Jump table
2     :: p_write_pc == wait_1 -> goto wait_1_loc;
3     :: p_write_pc == wait_2 -> goto wait_2_loc;
4     :: else -> skip;
5    fi;
6    /* BEGIN  wait(SC_ZERO_TIME) */
7    e_p_write_state = NOTIFIED_DELTA;
8    p_write_state = WAITING;
9    p_write_pc = wait_1;
10   p_write_suspend();
11 wait_1_loc: ;
12   /* END    wait(SC_ZERO_TIME) */
13 while_loop:
14   input(p_to_s_new);
15   p_to_s_req_up = true;
16   /* BEGIN  wait(e) */
17   p_write_state = WAITING;
18   p_write_pc = wait_2;
19   p_write_suspend();
20 wait_2_loc: ;
21   /* END    wait(e) */
22   goto while_loop;
23 }
```

Fig. 6. PROMELA encoding of the body of thread `write`.

3) Encoding Synchronization Functions: Figure 6 also shows the encoding of several SystemC synchronization primitives. Any call to a SystemC wait function, either `wait(`e`)`, for an event e, or `wait(`t`)`, for time t, suspends the thread execution. In particular, to handle a call to `wait(`t`)`, we associate each thread T with an event e_T. The encoding of `wait(`t`)` consists of setting e_T_state to `NOTIFIED_TIME` and e_T_time to t. When t is `SC_ZERO_TIME`, the event will be notified at the delta notification, and so e_T_state is set to `NOTIFIED_DELTA` and e_T_time can be left unchanged because its value is irrelevant.

The consumer code in our example contains a call `e.notify()` that immediately notifies the event `e`. Such an immediate notification is encoded as the inline code `e_notify` in Figure 7. The inline code `notify_threads` in the figure simply makes runnable the threads that are waiting for the event notifications. This code is also executed at the delta notification and timed notification of the scheduler.

```
bool p_write_notified, p_read_notified, c_read_and_ack_notified;

inline is_p_write_notified(notified) {
  ITE(((p_write_pc == wait_1 && e_p_write_state == NOTIFIED) ||
       (p_write_pc == wait_2 && e_state         == NOTIFIED)),
       notified = true, notified = false);
}

inline notify_threads() {
  is_p_write_notified(p_write_notified);
  ITE(p_write_notified, p_write_state = RUNNABLE, skip);
  is_p_read_notified(p_read_notified);
  ITE(p_read_notified, p_read_state = RUNNABLE, skip);
  is_c_read_and_ack_notified(c_read_and_ack_notified);
  ITE(c_read_and_ack_notified, c_read_and_ack_state = RUNNABLE, skip);
}

inline e_notify() { e_state = NOTIFIED;  notify_threads(); e_state = NONE; }
```

Fig. 7. PROMELA encoding of event notifications.

4) Encoding The Scheduler: Figure 8 shows an accurate encoding of the scheduler as described in Figure 1. The inline code `channel_updates` executes the encoding of the update function of each channel instance. The inline code `delta_notification` first sets the states of events to `NOTIFIED` if their current states are `NOTIFIED_DELTA`, then executes `notify_threads` to wake up the threads, and finally sets the states of the notified events to `NONE`. The inline code `exists_runnable_threads` checks for runnable threads, and sets `runnable` to `true` if there exists any runnable thread.

The inline code `time_acceleration` accelerates the time by subtracting the value of e_`time`, for each event e such that e_`state` is `NOTIFIED_TIME`, with the smallest value among them. It also sets e_`state` to `NOTIFIED_DELTA` if the resulting e_`time` becomes 0. Prior to accelerating the time, `time_acceleration` checks if there is an event e whose state is `NOTIFIED_TIME`. If there is no such an event, then it simply sets `accelerated` to `false` and quits; otherwise it accelerates the time and sets `accelerated` to `true`. The inline code `timed_notification` works in the same way as `delta_notification`.

The encodings that we propose have different expansions of the inline code `evaluation_phase`, and we will discuss each of them in the following sub-section.

The label `progress_DeltaCycle` is used to detect a non-progressing delta cycle. That is, during the state space exploration, SPIN checks whether a label prefixed by `progress` is visited infinitely often in an infinite execution; violation of this property yields a non-progressing cycle.

```
proctype Scheduler() {
 bool runnable, accelerated;
 channel_updates();
 delta_notification();
start_DeltaCycle:
 exists_runnable_threads(runnable);
 ITE(runnable,goto TimedNotification,skip);
 evaluation_phase();
 channel_updates();
progress_DeltaCycle:
 delta_notification();
 goto start_DeltaCycle;
TimedNotification:
 time_acceleration(accelerated);
 ITE(accelerated,goto SchedulerExit,skip);
 timed_notification();
 goto start_DeltaCycle;
SchedulerExit:
}
```

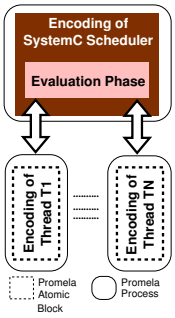
```
inline channel_updates() {
 ITE(p_to_c_req_up,
     p_to_c_update();
     p_to_c_req_up = false,skip);
 ITE(c_to_p_req_up,
     c_to_p_update();
     c_to_p_req_up = false,skip);
}


inline delta_notification() {
  ITE(e_state == NOTIFIED_DELTA,
      e_state = NOTIFIED,skip);
  ...
  notify_threads();
  ITE(e_state == NOTIFIED,
      e_state = NONE,skip);
  ...
}
```

Fig. 8. PROMELA encoding of the SystemC scheduler.

To complete our encoding, we have the following `init` process that initializes the global variables (`init_model` inline function) and runs the scheduler:

```
init { init_model(); run scheduler(); }
```




```
active proctype p_write() {
 p_write_entry:
  atomic { p_write_chan?TK; p_write_body(); }
 p_write_exit:
  goto p_write_entry;
}

inline evaluation_phase() {
 do
  :: p_write_state == RUNNABLE ->
     p_write_state = RUNNING;
     p_write_chan!TK; p_write_chan?TK;
  :: p_read_ack == RUNNABLE ->
     p_read_ack_state = RUNNING;
     p_read_ack_chan!TK; p_read_ack_chan?TK;
  :: c_read_and_ack == RUNNABLE ->
     c_read_and_ack_state = RUNNING;
     c_read_and_ack_chan!TK; c_read_and_ack_chan?TK;
  :: else -> break;
 od;
}
```

Fig. 9. Thread-to-process encoding.

5) Encoding Non-Deterministic Inputs: Input reading is encoded as inline code `input` (see Figure 6) that selects non-deterministically a value from a finite set of values.

B. Encoding Variants

1) Thread-To-Process Encoding: This encoding treats the scheduler and the threads as separate PROMELA processes, as shown in Figure 9.The synchronization between threads and the scheduler is modeled by passing the token `TK` through some rendezvous channels that connect the threads with the scheduler. For a design with N threads, we maintain N rendezvous channels such that each thread T exchanges the token with the scheduler through the channel T_`chan`.

Each thread T is encoded as a non-terminating PROMELA process T_`thread`, as shown in Figure 9. The thread body is enclosed within a PROMELA `atomic` block and it is preceded by receiving (operator `?`) the token `TK` from the designated channel. The receive statement is executable if and only if the rendezvous channel receives the token `TK`.

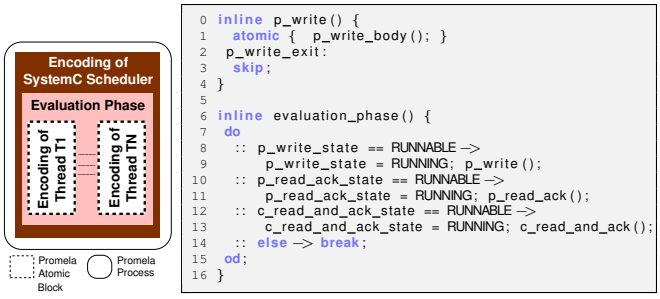
The thread suspension in the thread-to-process encoding is modeled as sending (operator `!`) the token `TK`, followed by receiving it back from the scheduler.

```
inline p_write_suspend() {  p_write_chan!TK; p_write_chan?TK; }
```

The sending and receiving of the token `TK` are sufficient to keep track of the locations in the control flow of the PROMELA code. Thus, this encoding does not require the jump table, the code labels for the encoding of thread suspensions, and the variables introduced to keep track of the program counters of threads.

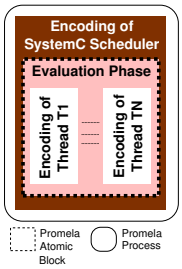
Figure 9 also shows the encoding of the evaluation phase. To model the cooperativeness and to allow the exploration of all possible thread interleavings, the scheduler non-deterministically chooses a runnable thread and sends the token `TK` to the thread (`do-od` construct). It then waits for the thread to give back the token `TK`.

2) Thread-To-Atomic-Block Encoding: This encoding uses a single PROMELA process containing both the scheduler and the threads, such that each SystemC thread is encoded as an atomic block inside the encoding of the evaluation phase, as shown in Figure 10. Passing control to the thread in the evaluation phase is modeled by executing the inline code of the thread, while giving back control to the scheduler is modeled by simply finishing the execution of the inline code.




```
0 inline p_write() {
1   atomic { p_write_body(); }
2  p_write_exit:
3   skip;
4 }
5
6 inline evaluation_phase() {
7  do
8   :: p_write_state == RUNNABLE ->
9       p_write_state = RUNNING; p_write();
10  :: p_read_ack_state == RUNNABLE ->
11      p_read_ack_state = RUNNING; p_read_ack();
12  :: c_read_and_ack_state == RUNNABLE ->
13      c_read_and_ack_state = RUNNING; c_read_and_ack();
14  :: else -> break;
15 od;
16 }
```

Fig. 10. Thread-to-atomic-block encoding.




```
0 proctype scheduler() {
1   ...
2   atomic {
3     evaluation_phase();
4   }
5   ...
6 }
```

Fig. 11. One-atomic-block encoding.

Unlike the thread-to-process encoding, in this encoding we do not need the rendezvous channels and the control token. The inline suspension code, e.g., `p_write_suspend`, of this encoding can be obtained from that of the thread-to-process encoding by replacing the sending and receiving statements with a jump to the exit label, i.e., `p_write_exit`.

3) One-Atomic-Block Encoding: This encoding is derived from the thread-to-atomic-block encoding by enclosing the whole evaluation phase in an `atomic` block, as shown in Figure 11. Although small, this modification can change the search behavior dramatically, and also affects the kinds of property that can be verified.

Unlike the other encodings, the one-atomic-block encoding cannot be used to detect non-progressing delta cycles. Recall that we use the label `progress_DeltaCycle` to allow SPIN to detect such cycles. However, to do so, the encodings must allow SPIN to sample states before and after the execution of each thread in the evaluation phase. Otherwise, a cycle of states in the state space that occurs during the exploration of the evaluation phase might go undetected. As shown in Figure 12 (a) (the dark/red circles), the thread-to-process and thread-to-atomic-block encodings sample the states before and after each thread execution, but, as shown in Figure 12 (b), the one-atomic-block encoding only samples states before and after each evaluation phase.

V. Encoding into Sequential Programs

In this section we describe an encoding of SystemC designs into sequential programs. The proposed encoding consists of a mapping of SystemC threads in the form of functions, and an encoding of the SystemC scheduler that accurately captures the semantics of SystemC. For presentation, we will show how to encode SystemC designs into sequential C programs. Encodings into different target sequential languages can be done in the same way modulo the differences in the language constructs.

Encoding Modules, Events, Channels, and Threads. The encodings of modules, events, channels, and threads in C are the same as those described in Section IV-A, modulo the differences in syntactical constructs. In general, all inline code introduced in Section IV-A can easily be turned into C functions. In particular, each SystemC thread is encoded as a C function, called *thread function*.

Thread suspensions are modeled as returning from the corresponding thread functions. That is, the inline code T`_suspend`, for a thread T, in the Promela encodings is replaced with a return statement. To model context switches, we associate each local variable of a thread with a global variable. When the thread yields the control to the scheduler, the corresponding thread function saves the values of its local variables into their associated global variables. On receiving the control back from the scheduler, it restores the values of its local variables. Note that, due to our assumption that all function calls can always be inlined, having additional global variables is sufficient for supporting context switches.

Figure 13 (left) shows the encoding of the thread `write`. Since the thread has no local variables, there is no saving and restoring of its local variables during thread suspensions. Differently from the finite-state encoding, the resulting sequential program is not an under-approximation since input reading is modeled by the function call `nondet()` that returns a non-deterministic input value.

Encoding the Scheduler. The encoding of the scheduler in C can easily be derived from that of the Promela encodings by turning the inline code into C functions. For example, the inline code `exists_runnable_threads(b)` can be translated into a function `int exists_runnable_threads()` that returns 1 if there is any runnable thread, or 0 otherwise.

The most notable difference from the Promela encodings is the encoding of the evaluation phase. The C language does not provide a non-deterministic construct like `do-od` and `if-fi` in Promela. To exercise all possible thread interleavings, we use the `nondet` function to make non-deterministic choices, as shown in Figure 13 (right). Note that the encoding yields an additional non-existent simulation where the scheduler does not pick any runnable thread. However, such discrepancy in the encoding of the scheduler does not affect the verification of safety properties.

To complete the encoding, we have a `main` function (corresponding to the `init` Promela process) that first calls the `init_models` function to perform the necessary initialization of global variables and then invokes the `scheduler`.

```
0 int main() { init_model(); scheduler(); return 0; }
```

VI. Model Checking SystemC via ESST

In this section we describe how to verify SystemC designs via ESST. First, we have to specialize the ESST to handle the SystemC scheduler and synchronization primitives. Second, we have to encode the SystemC design into a threaded program that ESST can analyze.

Specializing* ESST *to SystemC. ESST is a generic algorithm

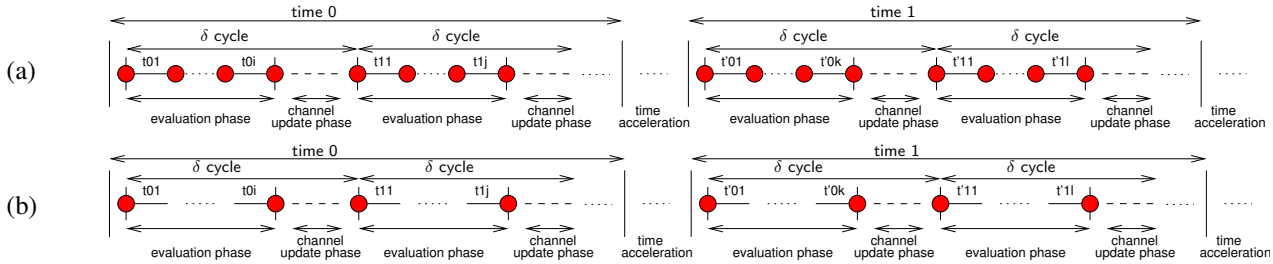


Fig. 12. The granularities of state sampling of (a) thread-to-process and thread-to-atomic-block encodings, and (b) one-atomic-block encoding. The dark/red circles denote the states that are stored during the state exploration of SPIN.

```
0  void p_write() {
1    /**** Jump table ****/
2    if (p_write_pc == wait_1)
3      goto wait_1_loc;
4    else if (p_write_pc == wait_2)
5      goto wait_2_loc;
6
7    /* BEGIN   wait(SC_ZERO_TIME) */
8    e_p_write_state = NOTIFIED_DELTA;
9    p_write_state = WAITING;
10   p_write_pc = wait_1;
11   return;
12 wait_1_loc:
13   /* END     wait(SC_ZERO_TIME) */
14   while (1) {
15     p_to_s_new = nondet();
16     p_to_s_req_up = 1;
17     /* BEGIN   wait(e) */
18     p_write_state = WAITING;
19     p_write_pc = wait_2;
20     return;
21 wait_2_loc:
22     /* END     wait(e) */
23   }
24 }
```

```
0  void evaluation_phase() {
1
2   while (exists_runnable_threads()) {
3
4    if (p_write_state == RUNNABLE
5        && nondet()) {
6      p_write_state = RUNNING;
7      p_write();
8    }
9
10   if (p_read_state == RUNNABLE
11       && nondet())
12     p_read_state = RUNNING;
13     p_read();
14   }
15
16   if (c_read_ack_state == RUNNABLE
17       && nondet())
18     c_read_ack_state = RUNNING;
19     c_read_ack();
20   }
21
22  }
23
24 }
```

Fig. 13. Encodings of (left) thread `write` and of (right) evaluation phase.

for model checking cooperative threads. To verify SystemC designs, we need to specialize ESST to SystemC. Such a specialization amounts to instantiating the ESST scheduler with an implementation of the SystemC scheduler, and defining a set of primitive functions that implement the synchronization functions of SystemC.

We require that, for each SystemC synchronization function, there is a corresponding primitive function that implements it. Furthermore, we introduce an enumeration type `SCEvent` that lists all identifiers associated with event instances in the design. For example, the wait functions `wait(`e`)`, for an event e, and `wait(`t`)`, for time t, correspond, respectively, to the primitive functions `wait_event(`E`)` and `wait_time(`t`)`, where E is the member of `SCEvent` associated to event e.

Encoding into Threaded Programs. The encoding of SystemC designs into threaded programs is simpler than the encoding into sequential programs because the presence of primitive functions and the absence of the scheduler in the encoding. We encode each SystemC threads into a thread in the threaded program. Furthermore, we encode the update function of each SystemC channel instance into a special thread which is executed by the scheduler only during the update phase. The threaded program also contains a `main` thread from which the execution starts. Similar to the `main` function in the sequential program, the `main` thread performs necessary initializations, and then yields the control back to the scheduler to start the simulation.

Each call to a SystemC synchronization function is encoded into a call to the corresponding primitive function.

```
0  void p_write() {
1    wait_time(0);        /* wait(SC_ZERO_TIME) */
2
3    while (1) {
4      p_to_s_new = nondet();
5
6      wait_event(e);     /* wait(e) */
7    }
8  }
```

Fig. 14. The encoding of the thread `write` in the threaded program.

In particular, for synchronization functions that involve event instances, the event arguments are replaced with the identifiers corresponding to the event instances (members of `SCEvent`).

As described in Section III,ESST keeps track of the control location of each thread during its analysis. Thus, unlike the encoding into sequential programs, the threaded program does not need the global variables that keep track of program counters nor the jump table for modeling thread suspension. ESST also keeps track of the region of each thread, and so we do not need additional global variables for storing the values of variables local to a thread when the thread suspends itself. Moreover, in its state, the scheduler keeps track of the state of each thread, of each event, and of the notification time delays. Thus, the threaded program does not need supporting variables for tracking such information.

Figure 14 shows the encoding of the SystemC thread `write` as a thread in the threaded program. The encoding largely resembles the thread code in Figure 3. Notice that, the signal channel is modeled in the same way as in the case of the encoding into the sequential program. The same treatment is also applied to the member variables of module instances.

VII. Related Work

SystemC verification via translation to other languages has been reported in a number of publications. LUSSY [5] is a toolbox that translates SystemC designs into LUSTRE and SMV models. Similar to our translations, the translation performed by LUSSY captures the full semantics of SystemC. However, due to the complexity of the SystemC scheduler, the experimental evaluation reported in [35] shows that the approach implemented in LUSSY does not scale up.

SystemC verification via reduction to CTL model checking over Petri nets has been described in [36]. The experiments in [36] shows that, even for small SystemC designs, the resulting Petri nets can be very large, and thus hard to analyze.

An encoding of a SystemC design into a network of timed automata has been reported in [9]. The resulting network of timed automata is then analyzed using UPPAAL [37]. Similar

to the PROMELA encodings, the resulting timed automata are under-approximations of the original SystemC design.

The work in [38] encodes SystemC designs into the process algebra LOTOS and uses the CADP toolbox [39] for the verification. The work only considers untimed SystemC designs, and the translation into LOTOS does not faithfully models the semantics of SystemC. That is, the work only considers SystemC as a description language for the design under verification, but does not impose the SystemC simulation semantics in analyzing the design.

There have been several works on translating SystemC designs into PROMELA code; see [8], [40]. Similar to our thread-to-process encodings, each SystemC thread is encoded as a PROMELA process, but, unlike ours, they embed the SystemC scheduler into the encoding of the synchronization primitives. The focus of these encodings is to verify SystemC designs at transaction level modeling (TLM). In TLM the delta cycles are typically not visible, and communications between threads are usually through shared global variables instead of SystemC channels. Thus, unlike our encodings that capture the full semantics of the SystemC scheduler, the existing encodings only consider the evaluation and the timed update phases of the scheduler. Moreover, since the evaluation phase is encoded implicitly in the encoding of the synchronization primitives, these encodings are unable to detect non-progress delta cycles. In [8] each SystemC thread is encoded into an automata, which in turn is encoded as a PROMELA process. This encoding is not efficient because the resulting PROMELA process has to keep track of the states of the automaton instead of the program counter of thread suspension points. Moreover, the encodings in [8] assume that time is discrete. The PROMELA encoding presented in [40] is an improvement of the one in [8]. It aims at reducing the number of interactions between the encodings of the threads by exploiting PROMELA blocking statements. We have not been able to empirically compare our PROMELA encodings with those of [40] due to some regressions that breaks the tool used in [40] to generate the encoding. (Personal communication with Matthieu Moy.)

As a concluding remark, we point out that most of the tools for the works cited above are not publicly available, and therefore could not be included in the experimental evaluation reported in next section.

The work in [41] translates SystemC designs into sequential programs and applies induction-based techniques to the sequential programs. Similar to [38], this work only considers untimed SystemC designs.

Formal verification of SystemC designs that abstracts away the scheduler has been reported in [7]. In this work, each thread in the SystemC design is encoded as a labelled transition system (LTS), and the behavior of the whole SystemC design is modeled by the parallel composition of the LTS's of the threads. The scheduler is not explicitly modeled; rather, its encoding is implicitly distributed in the encodings of the threads. This work is limited to the evaluation phase, and does not handle channel updates and time delays.

The work in [10] discusses an approach based on monitoring for the verification of SystemC properties. In [10] the SystemC scheduler and the design are instrumented to observe desirable properties during simulations. However, SystemC simulations do not explore all possible schedules, and so the monitoring process can miss some bugs. Moreover, this approach does not allow for detecting non-progressing delta cycles.

A multi-layer modeling of high-level SystemC designs has been described in [42]. A SystemC design is viewed as having three layers: functional, sequential, and simulation. The first two layers are modeled, respectively, as extended Petri nets, and as predictive synchronization dependence graph, while the simulation layer precisely models the SystemC scheduler. This approach has been used for symbolic simulations and deadlock checking (via BMC), but no methods for full-fledged verification are reported.

Simulation coverage for SystemC designs has been addressed in [43]–[45]. The works in [43], [44] focus on exploring all possible schedules of the design for a given input by using dynamic partial-order reduction. The work in [45] explores subsets of all possible schedules encoded in a partial-order trace that is extracted from a single simulation trace by exploiting concurrency information in the trace.

Formal analysis of SystemC has been applied to other settings than verification. The tool SCOOT [46] has been used to speed up simulations by synthesizing an optimized SystemC scheduler that performs partial-order reduction using information obtained from race analysis [47]. SCOOT has also been used to perform run-time verification by extracting from a SystemC design a flat C++, model that can be analyzed by SATABS [17]. Similar to our sequentialization approach, the SystemC scheduler is included in the flat model.

A general approach for defining temporal languages for SystemC has been described in [4]. The work provides a trace semantics that describes the execution a SystemC design. This semantics has inspired the encodings of the SystemC scheduler in our work, and accounts for the different levels of granularity in our PROMELA encodings (see Figure 12).

The work in [48] provides a rigorous analysis of existing SystemC front-ends, where it emerged that PINAPA [21] and its recent re-implementation PinaVM [49] are the two front-ends covering more SystemC constructs. These two front-ends are at the basis of the works in [5], [8], [35], [40].

VIII. IMPLEMENTATION AND EXPERIMENTS

The Tool Chain. We have implemented the proposed software model checking approaches in a tool chain consisting of two main tools: SYSTEMC2C and KRATOS. The tool chain can be downloaded at `http://es.fbk.eu/tools/kratos`.

SYSTEMC2C performs translations from SystemC designs into sequential and threaded C programs, and is built as a new back-end for PINAPA [21]. PINAPA parses the SystemC design and executes it until the point just before the simulation starts. At that point PINAPA provides SYSTEMC2C with information about all SystemC objects of the design, including module instances, channel instances, and their interconnections, as well as the abstract syntax tree of the threads.

KRATOS [25] is a software model checker for sequential and threaded C programs. KRATOS implements the lazy predicate abstraction and the lazy abstraction with interpolants for analyzing sequential programs, and ESST for analyzing threaded

programs. KRATOS also provides a specialization of ESST to SystemC verification. KRATOS is built on top of an extended version of NUSMV [50], which is tightly integrated with the MATHSAT SMT solver [51].

KRATOS provides a capability of performing the PROMELA encodings, described in Section IV-B, on threaded programs. Thus, to explore the PROMELA encodings, the SystemC design has to be translated first into a threaded program using SYSTEMC2C, and then the resulting threaded program is translated into PROMELA using KRATOS.

Limitations: The PINAPA SystemC front-end has several limitations. It does not recognize all SystemC TLM constructs and does not fully support function pointers. For the experiments we extended PINAPA to handle simple TLM constructs like `sc_export`, which is quite pervasive in SystemC designs. The translator SYSTEMC2C currently does not support rich C++ features like standard template library (STL) data structures. Finally, the KRATOS model checker is not able to handle designs that use complex data types (like e.g. structs), arrays, pointers, dynamic creation of objects, and recursive function. Currently, structs are flattened, each fixed-size array is replaced by multiple scalar variables and by two functions implementing reading from and writing to the array, and pointers are replaced by scalar variables.

Experimental Evaluation. We have carried out an experimental evaluation using a large set of benchmarks taken and adapted from the SystemC distribution [3], from [35], and from [43]. We also derived two new families of unsafe benchmarks, `mem-slave-tlm-bug2` from `mem-slave-tlm` and `token-ring-bug2` from `token-ring`. In these new families the assertion violations can only be found by non-trivial combinations of input values.

We experimented with the PROMELA encodings for bug hunting. That is, we only consider the unsafe benchmarks in the experiment. We model input reading by selecting a value from a set of ten random values. For the considered benchmarks, we found that such a modeling was sufficient to find the bugs. We ran SPIN on all the proposed PROMELA encodings by setting the depth limit search to 1,000,000.

For the sequentialization approach, we compared SATABS-3.0 [17] for the eager abstraction; BLAST-2.7 [22], CPACHECKER revision 5160 [23], and KRATOS [25] for the lazy predicate abstraction; and WOLVERINE-0.5 [24] and KRATOS for the lazy abstraction with interpolants. We also evaluated the BMC technique implemented in CBMC-4.0 [18] by setting the size of loop unwindings to 3 and by considering only the unsafe benchmarks. Finally, we experimented with the ESST algorithm implemented in KRATOS.

We ran our experiments on a Linux box with Intel-Xeon DC 3GHz processors and 4GB of RAM. We set the time limit to 1000s and the memory limit to 2GB. Data to reproduce our experiment results are available at `http://es.fbk.eu/people/roveri/tests/tcad-2012`.

Overall Results. Table I shows the run-time results (in seconds) of the experiments. The column V indicates the status of each benchmark, S for safe and U for unsafe. In the experiments with the PROMELA encodings, the columns T2P, T2AB, and 1AB stand for, respectively, thread-to-process, thread-to-atomic-block, and one-atomic-block encodings. In the experiments with the sequentialization approach, the columns Lazy PA and Lazy AWI stand for, respectively, the lazy predicate abstraction and the lazy abstraction with interpolants. The results T.O. and M.O. indicates that the experiment goes out of time and out of memory, respectively. We mark with - if the result is not available since the experiment was not carried out and mark with Err if the tool produces errors. The best results are highlighted in bold.

Table I shows that, on most unsafe benchmarks, the PROMELA encodings are effective for bug finding. However, due to the under-approximations, the approach can miss some bugs, as witnessed by the experiments on the `token-ring-bug2` benchmarks. In these benchmarks the assertion violations can only be revealed by intricate combinations of input values. For small numbers of token rings, the PROMELA encodings can still find assertion violations. However, for large token rings, the combinations of input values, as well as the large number of thread interleavings, lead to the state explosion problem that makes the PROMELA encodings go out of time or out of memory.

The sequentialization approach can be used to prove and disprove properties. However, as shown on Table I, the approach does not scale up. For example, for the `token-ring.X` families of benchmarks and their buggy versions, the model checkers cannot verify the benchmarks for large `X`. For these families, the BMC approach via CBMC can disprove the properties. However, for the `mem-slave-tlm-bug` and `mem-slave-tlm-bug2` families of unsafe benchmarks, we mark the results with * because, due to insufficient loop unwindings, CBMC reports that the benchmarks are safe. We have tried to incrementally increase the size of loop unwindings to find the bugs, but, in so doing, the constraint fed into CBMC becomes large and hard to solve. As a result, CBMC goes out of time on some benchmarks that it is able to verify with small loop unwindings.

Table I shows that ESST is effective for both proving and disproving properties. Although it is less effective than the PROMELA encodings in disproving properties, these results demonstrate that the PROMELA encodings can fruitfully complement ESST in the verification of SystemC designs.

Results of* PROMELA *Encodings. Table II shows the detailed results of experiments with the PROMELA encodings. The table provides 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 the result with $*$ when SPIN reaches the depth limit. For the checks for non-progressing cycles, reported by the column 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, and also performed experiments on both safe and unsafe benchmarks. We mark with $\surd$ and $-$ for, respectively, a detection of a non-progressing cycle and an inconclusive result. The time needed for encoding the SystemC design into PROMELA, as well as for obtaining the protocol analyzer, is

Benchmark	V	Finite-State Models			Sequentialization							Threaded
		PROMELA +SPIN			Eager	Lazy PA			Lazy AWI		BMC	ESST
		T2P	T2AB	1AB	SATABS	BLAST	CPACHECKER	KRATOS	WOLVERINE	KRATOS	CBMC	
bist-cell	S	-	-	-	31.740	5.300	7.340	0.400	T.O.	**0.300**	-	1.390
kundu-bug-1	U	**0.001**	0.010	**0.001**	33.730	105.850	24.310	25.000	205.370	25.590	1.080	0.590
kundu-bug-2	U	**0.001**	**0.001**	**0.001**	79.160	Err	17.710	0.890	580.990	11.200	2.450	0.500
kundu	S	-	-	-	96.460	Err	35.620	151.490	T.O.	T.O.	-	**1.090**
mem-slave-tlm.1	S	-	-	-	69.150	80.360	120.060	139.790	78.920	40.590	-	**3.500**
mem-slave-tlm.3	S	-	-	-	385.410	745.690	M.O.	T.O.	470.890	596.250	-	**42.690**
mem-slave-tlm.5	S	-	-	-	T.O.	Err	T.O.	T.O.	T.O.	T.O.	-	**280.260**
mem-slave-tlm-bug.1	U	**0.001**	**0.001**	**0.001**	83.140	84.420	42.190	13.800	90.710	27.790	53.750	2.600
mem-slave-tlm-bug.3	U	0.010	**0.001**	**0.001**	719.070	763.640	M.O.	T.O.	505.100	687.150	55.850*	33.390
mem-slave-tlm-bug.5	U	**0.001**	**0.001**	0.010	T.O.	Err	M.O.	T.O.	T.O.	T.O.	56.890*	207.970
mem-slave-tlm-bug2.1	U	**0.001**	**0.001**	**0.001**	75.610	82.070	33.160	2.790	85.830	18.000	54.770	1.400
mem-slave-tlm-bug2.3	U	0.150	**0.130**	0.300	391.900	T.O.	71.680	18.390	T.O.	401.870	56.400*	12.290
mem-slave-tlm-bug2.5	U	21.000	**17.000**	43.300	T.O.	Err	158.580	85.090	T.O.	T.O.	58.960*	40.490
pc-sfifo-1	S	-	-	-	3.490	20.350	16.960	3.300	13.690	3.590	-	**0.300**
pc-sfifo-2	S	-	-	-	4.810	34.650	25.820	8.400	32.430	25.590	-	**0.500**
pipeline-bug	U	**0.001**	**0.001**	**0.001**	737.320	T.O.	54.840	13.600	T.O.	103.290	-	6.400
pipeline	S	-	-	-	T.O.	T.O.	**67.630**	T.O.	T.O.	T.O.	-	81.790
token-ring.1	S	-	-	-	9.970	6.360	11.940	4.300	49.880	3.500	-	**0.100**
token-ring.5	S	-	-	-	814.160	T.O.	M.O.	T.O.	T.O.	T.O.	-	**0.400**
token-ring.9	S	-	-	-	T.O.	T.O.	M.O.	T.O.	T.O.	T.O.	-	**1.100**
token-ring.13	S	-	-	-	T.O.	M.O.	M.O.	T.O.	T.O.	T.O.	290.450	**4.500**
token-ring-bug.1	U	**0.001**	**0.001**	**0.001**	5.460	3.300	14.870	1.500	T.O.	2.590	1.620	**0.001**
token-ring-bug.5	U	**0.001**	**0.001**	**0.001**	748.250	T.O.	M.O.	T.O.	T.O.	T.O.	15.060	0.100
token-ring-bug.9	U	**0.001**	**0.001**	**0.001**	T.O.	T.O.	M.O.	T.O.	T.O.	T.O.	95.460	0.300
token-ring-bug.13	U	**0.001**	**0.001**	**0.001**	T.O.	M.O.	M.O.	T.O.	T.O.	T.O.	288.940	1.790
token-ring-bug2.1	U	0.010	**0.001**	4.100	5.940	2.480	13.980	2.000	T.O.	1.500	1.090	**0.001**
token-ring-bug2.5	U	T.O.	T.O.	T.O.	819.060	T.O.	M.O.	T.O.	T.O.	T.O.	15.370	**0.100**
token-ring-bug2.9	U	M.O.	M.O.	T.O.	T.O.	T.O.	M.O.	T.O.	T.O.	T.O.	97.980	**0.390**
token-ring-bug2.13	U	M.O.	M.O.	T.O.	T.O.	M.O.	M.O.	T.O.	T.O.	T.O.	312.380	**2.700**
toy-bug-1	U	5.550	5.340	4.560	23.570	241.240	45.650	10.200	T.O.	T.O.	1.430	**0.490**
toy-bug-2	U	5.690	5.290	4.560	19.560	144.610	44.810	3.890	T.O.	T.O.	1.410	**0.200**
toy	S	-	-	-	22.150	Err	195.620	T.O.	T.O.	T.O.	-	**1.800**
transmitter.1	U	**0.001**	**0.001**	**0.001**	2.280	1.190	17.060	1.090	T.O.	0.800	0.430	**0.001**
transmitter.5	U	**0.001**	**0.001**	**0.001**	224.070	T.O.	353.480	409.670	T.O.	T.O.	10.080	**0.001**
transmitter.9	U	**0.001**	0.010	**0.001**	T.O.	T.O.	M.O.	T.O.	T.O.	T.O.	74.420	0.100
transmitter.13	U	**0.001**	**0.001**	**0.001**	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.	259.060	0.090

TABLE I: RESULTS OF EXPERIMENTS.

			thread-to-process						thread-to-atomic-block						one-atomic-block					
Benchmark	V	NP	# States stored	# Transitions	State size (byte)	Time (sec)	Compression (%)	# Transitions No POR	# States stored	# Transitions	State size (byte)	Time (sec)	Compression (%)	# Transitions No POR	# States stored	# Transitions	State size (byte)	Time (sec)	Compression (%)	# Transitions No POR
bist-cell.c	S	—	8	8	180	0.001	-	8	8	8	168	0.001	-	8	6	6	168	0.001	-	6
kundu-bug-1.c	U	—	18	18	156	0.001	-	18	18	18	132	0.010	-	18	11	11	132	0.001	-	11
kundu-bug-2.c	U	—	121	136	212	0.001	-	136	121	136	176	0.001	-	136	36	51	176	0.001	-	51
kundu.c	S	—	1386	1591	212	0.020	-	1591	1386	1591	176	0.001	-	1591	405	610	176	0.001	-	610
mem-slave-tlm.1.c	S	—	93	112	268	0.010	-	112	93	112	228	0.001	-	112	6	25	228	0.001	-	25
mem-slave-tlm.3.c	S	—	181	200	276	0.001	-	200	181	200	236	0.010	-	200	6	25	236	0.010	-	25
mem-slave-tlm.5.c	S	—	269	288	284	0.001	-	288	269	288	244	0.010	-	288	6	25	244	0.010	-	25
mem-slave-tlm-bug.1.c	U	—	16	16	268	0.001	-	16	16	16	228	0.001	-	16	3	3	228	0.001	-	3
mem-slave-tlm-bug.3.c	U	—	32	32	276	0.010	-	32	32	32	236	0.001	-	32	3	3	236	0.001	-	3
mem-slave-tlm-bug.5.c	U	—	48	48	284	0.001	-	48	48	48	244	0.001	-	48	3	3	244	0.010	-	3
mem-slave-tlm-bug2.1.c	U	—	62	98	276	0.001	-	98	62	98	240	0.001	-	98	15	51	240	0.001	-	51
mem-slave-tlm-bug2.3.c	U	—	9102	13098	284	0.150	63.42	13098	9102	13098	248	0.130	71.74	13098	1335	5331	248	0.300	-	5331
mem-slave-tlm-bug2.5.c	U	—	1264222	1664218	292	21.000	41.46	1664218	1264222	1664218	256	17.000	44.43	1664218	133335	533331	256	43.300	54.49	533331
pc-sfifo-1.c	S	—	6470247	9117167	164	* 78.200	56.11	9117167	8461247	11922667	140	* 76.200	58.9	11922667	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.
pc-sfifo-2.c	S	—	11955977	13912417	180	* 102.000	47.88	13912417	14472827	16841197	148	* 93.100	50.07	16841197	8249603	10499533	148	* 71.200	71.4	10499533
pipeline-bug.c	U	—	76	76	660	0.001	-	76	76	76	572	0.001	-	76	26	26	572	0.001	-	26
pipeline.c	S	—	M.O.	M.O.	M.O.	M.O.	M.O.	M.O.	M.O.	M.O.	M.O.	M.O.	M.O.	M.O.	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.
token-ring.1.c	S	√	107	207	148	0.001	-	207	107	207	116	0.010	-	207	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.
token-ring.5.c	S	√	3336	4955	524	0.060	82.68	4955	3336	4955	428	0.040	-	4955	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.
token-ring.9.c	S	√	82960	138643	1156	2.610	16.87	138643	82960	138643	988	1.640	17.24	138643	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.
token-ring.13.c	S	√	1831304	3228171	2044	96.600	11.74	3228171	1831304	3228171	1812	63.400	11.44	3228171	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.
token-ring-bug.1.c	U	√	10	10	148	0.001	-	10	10	10	116	0.001	-	10	3	3	116	0.001	-	3
token-ring-bug.5.c	U	√	26	26	524	0.001	-	26	26	26	428	0.001	-	26	3	3	428	0.001	-	3
token-ring-bug.9.c	U	√	42	42	1156	0.001	-	42	42	42	988	0.001	-	42	3	3	988	0.001	-	3
token-ring-bug.13.c	U	√	153	176	2044	0.001	-	176	153	176	1812	0.001	-	176	15	44	1812	0.001	-	44
token-ring-bug2.1.c	U	√	212	551	156	0.010	-	551	212	551	124	0.001	-	551	3	50003	124	* 4.100	-	50003
token-ring-bug2.5.c	U	√	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.
token-ring-bug2.9.c	U	√	M.O.	M.O.	M.O.	M.O.	M.O.	M.O.	M.O.	M.O.	M.O.	M.O.	M.O.	M.O.	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.
token-ring-bug2.13.c	U	√	M.O.	M.O.	M.O.	M.O.	M.O.	M.O.	M.O.	M.O.	M.O.	M.O.	M.O.	M.O.	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.
toy-bug-1.c	U	—	166687	166691	396	* 5.550	51.08	166691	224154	224158	320	* 5.340	65.2	224158	54555	54559	320	* 4.560	-	54559
toy-bug-2.c	U	—	168843	168844	396	* 5.690	50.87	168844	228096	228100	320	* 5.290	65.2	228100	55560	55564	320	* 4.560	-	55564
toy.c	S	—	376677	415656	396	* 4.480	41.79	415656	508822	561472	320	* 4.250	50.98	561472	111114	222249	320	* 2.840	86.33	222249
transmitter.1.c	U	√	8	8	132	0.001	-	8	8	8	92	0.001	-	8	3	3	92	0.001	-	3
transmitter.5.c	U	√	24	24	492	0.001	-	24	24	24	388	0.001	-	24	3	3	388	0.001	-	3
transmitter.9.c	U	√	40	40	1108	0.001	-	40	40	40	932	0.010	-	40	3	3	932	0.001	-	3
transmitter.13.c	U	√	116	137	1980	0.001	-	137	116	137	1740	0.001	-	137	9	42	1740	0.001	-	42

TABLE II: RESULTS OF EXPERIMENTS WITH DIFFERENT PROMELA ENCODINGS.

negligible and is not reported.

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, the size of state for the thread-to-process encoding is larger than that of the thread-to-atomic-block encoding. In the thread-to-process encoding the more threads in the SystemC design, the more processes and rendezvous channels we need, and thus the size of the states is larger. In the other encodings the design is encoded as a single process and use no channels for synchronizations.

Comparing the one-atomic-block and thread-to-atomic-block encodings, it is obvious that the former encoding explores less states than the latter one. However, the one-atomic-block encoding can explore more transitions than the thread-to-atomic-block encoding. Consider three runnable threads t_1, t_2, and t_3 such that each of them only accesses local variables. 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 have the prefix t_2, t_1, then, because the resulting states

of performing t_1, t_2 and t_2, t_1 are the same, the thread-to-atomic-block encoding allows to detect that the state has been visited before, and do not explore t_3. For the one-atomic-block encoding the search continues exploring t_3 because the intermediate states between thread executions are not stored.

Table II shows that, for the unsafe benchmarks, like `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.

We experimented with the collapse compression method of SPIN. For the thread-to-process encoding, when there are only minor variations of the values of global variables or of the local variables of processes, the collapse compression method can reduce the size of state. Although the method is also applicable to the other two encodings, the reductions of memory consumption will not be as significant as for the thread-to-process encoding. The column Compression of Table II reports the application of collapse compression. The numbers in the colum indicate the percentage of actual memory usage relative to the memory usage without compression. (Thus, the smaller the number, the more effective the compression.) On most benchmarks the application of collapse compression is not effective: the overhead of collapse compression enlarges the memory usage. A close inspection showed that the threads in most of the benchmarks do not have variables that are local to the threads, but use heavily shared global variables for channel communications and computations. Moreover, in those benchmarks there is a large number of different variations of the global variables.

On `toy-bug` the application of the collapse compression is effective to reduce the memory consumption. In fact in this family of benchmarks the threads use local variables in their computations, and in particular, in the thread-to-process encoding, there are only small variations of process states, and thus the actual memory usage can be compressed to less than 41% of the original usage. With the thread-to-atomic-block encoding the compression is larger than 65%.

SPIN implements a POR method. However, its application 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. Our experiments, reported in the columns # Transitions and # Transitions No POR, confirm that the POR method of SPIN is not effective to reduce the number of explored transitions.

Unfortunately, adding the PROMELA code for the dependence relation to the resulting encoding is not sufficient. For checking safety properties, one has to ensure that the proviso condition described in [27] is satisfied at the level of SystemC non-interleaved transitions. To guarantee this condition, one needs to be able to access the states stored by the protocol analyzer generated by SPIN from the PROMELA encodings.

Column NP in Table II, shows that the thread-to-process and the thread-to-atomic-block encodings find non-progressing delta cycles in some benchmarks, e.g., `token-ring` and `transmitter`. Further under-approximation on some other benchmarks allows us to detect non-progressing delta cycle before the search reaches the depth limit. For example, by setting the integer counter in `pc-sfifo-1` to range only over byte values, we can detect a non-progressing delta cycle.

Results of Sequentialization. The model checking techniques described in Section III treat both the scheduler and the threads as part of the program under verification. Such general purpose techniques do not exploit the intrinsic structure of the combination of the scheduler and the threads, which is hidden by the encoding into the sequential program. In many cases such a structure, as well as the details of the scheduler, is needed to determine the correctness of the design. For example, a bug in the design only appears after a particular long sequence of thread interleavings and with specific input values. Such a sequence can only be explored if the analyses have detailed information of the scheduler and its interactions with the threads.

The above cases turn out to be problematic for the CEGAR-based techniques, because the initial abstraction of the scheduler is often too aggressive, and thus many refinements are needed to re-introduce necessary details. In particular, the techniques based on the predicate abstraction have to keep track of too many predicates to model the details of the scheduler. Unfortunately, the more predicates we need to keep track of, the more expensive the abstraction computations become. For example, in the experiments with the lazy predicate abstraction of KRATOS on the `token-ring` benchmarks and their buggy versions, we found that most of the predicates involved are concerned with the state of the scheduler, i.e., the states of the threads and events. The more threads in the design, the more predicates that KRATOS keeps track of. Thus, KRATOS does not scale up on these benchmarks. Similar considerations hold for SATABS, BLAST, and CPACHECKER.

The tools based on the lazy abstraction with interpolants solve fewer benchmarks than those based on the lazy predicate abstraction. A close inspection on KRATOS highlights that, on most benchmarks, this technique shows slow convergence. The interpolant generated as region appears to be too weak to make the state covered by other abstract states, and subsequent refinements of the other abstract states involve strong interpolants that prevent them from covering the former one.

CEGAR-based techniques can be effective in cases where the correctness of the design does not depend on the specific interaction between the scheduler and the threads, e.g., it does not depend on the order of thread interleavings. In these cases the scheduler can often be abstracted away.

The sequentialization approach allows for aggressive static optimizations. For example, on the `bist-cell` benchmark, KRATOS can verify the benchmark by relying only on its

optimizing transformations, and without even performing the lazy abstraction analysis. The same result cannot be obtained for the threaded case because the semantics of the threaded program relies on the scheduler and on the primitive functions.

Comparing the tools based on the predicate abstraction, Table I shows that SATABS verifies more sequential benchmarks than other tools. SATABS employs the weakest precondition technique for discovering predicates. On `mem-slave-tlm` and `token-ring`, such a technique finds a smaller number of predicates than those found by CPACHECKER and KRATOS that employ interpolation-based technique for discovering predicates. The results of the lazy predicate abstraction of CPACHECKER and KRATOS are comparable. The different search directions employed by CPACHECKER and KRATOS explain the differences in the experiment results. BLAST shows poor performances compared to other tools.

The BMC technique of CBMC analyzes the sequential program without performing any abstractions, but bounds the number of loop unwindings. Thus, the analysis can only be complete if the program is loop-free or the bounds for the loops are known statically. However, the encoding of the scheduler, in particular the evaluation phase, creates loops whose bounds cannot be inferred statically. Moreover, the longer the sequences of thread interleavings to explore, the higher the bounds are required. Due to the breadth-first nature of BMC, increasing the bounds adds a lot of constraints to the underlying SAT/SMT solvers. These additional constraints degrades the performance of the solvers, and thus the BMC technique itself. Nevertheless, the BMC approach is more effective than other symbolic techniques in cases where bugs can be revealed with a short sequence of thread interleavings.

***Results of* ESST.** The results in Table I demonstrate that ESST is effective both in proving correctness and in finding bugs. For proving correctness, ESST outperforms the sequentialization approach on almost all benchmarks. ESST keeps track of precise details of the scheduler and its interaction with the threads without additional predicates that can burden the predicate abstraction analysis. Thus, compared to the model checking via sequentialization, ESST can greatly reduce the number of predicates to keep track of. The same benefit is obtained by avoiding saving and restoring the values of local variables to the global variables.

For bug finding, Table I shows that ESST verifies more unsafe benchmarks than the PROMELA encodings. Unlike the PROMELA encodings, ESST can find bugs in large `token-ring-bug2` benchmarks. In this family of benchmarks, the assertion violations can only be reached by intricate combinations of input values.

IX. CONCLUSION AND FUTURE WORK

In this paper we have proposed a method for the formal verification of SystemC. The approach is based on the encoding of SystemC programs into finite-state/sequential/threaded software, and leverages the combination of heterogeneous state-of-the-art software model checking techniques. Each encoding features a faithful modeling of the SystemC semantics, and shows different characteristics in terms of the properties that can be verified and the efficiency of the search carried out by the model checker.

We have implemented the approach in a comprehensive tool chain, and performed a thorough experimental evaluation on benchmarks taken and derived from literature on SystemC verification. The results of the experimental evaluation show the applicability of the proposed approaches, the effectiveness of the ESST algorithm for proving the properties, and of the PROMELA encodings for finding bugs and detecting non-progressing delta cycles.

As future work, we will extend our back-end to support richer data (e.g., arrays [52], [53]), and enlarge the set of primitives of interaction between the scheduler and the threads, to better handle all TLM constructs. Then, we will investigate how to extend the ESST approach to deal with parametric primitive functions (such as suspension with duration that is not yet specified, or decided at run time). Finally, we will also investigate how to extend the approaches to verify other kinds of property, e.g., complex temporal property.

REFERENCES

[1] A. Cimatti, A. Micheli, I. Narasamdya, and M. Roveri, "Verifying SystemC: A software model checking approach," in *FMCAD*, R. Bloem and N. Sharygina, Eds. IEEE, 2010, pp. 51–59.

[2] D. Campana, A. Cimatti, I. Narasamdya, and M. Roveri, "An analytic evaluation of SystemC encodings in promela," in *SPIN*, ser. LNCS, A. Groce and M. Musuvathi, Eds., vol. 6823. Springer, 2011, pp. 90–107.

[3] "IEEE 1666: SystemC language Reference Manual," 2005.

[4] D. Tabakov, G. Kamhi, M. Y. Vardi, and E. Singerman, "A Temporal Language for SystemC," in *FMCAD*, A. Cimatti and R. B. Jones, Eds. IEEE, 2008, pp. 1–9.

[5] 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*. IEEE, 2005, pp. 26–35.

[6] D. Große and R. Drechsler, "CheckSyC: an efficient property checker for RTL SystemC designs," in *ISCAS (4)*. IEEE, 2005, pp. 4167–4170.

[7] D. Kroening and N. Sharygina, "Formal verification of SystemC by automatic hardware/software partitioning," in *MEMOCODE*. IEEE, 2005, pp. 101–110.

[8] C. Traulsen, J. Cornet, M. Moy, and F. Maraninchi, "A SystemC/TLM Semantics in Promela and Its Possible Applications," in *SPIN*, ser. LNCS, D. Bosnacki and S. Edelkamp, Eds., vol. 4595. Springer, 2007, pp. 204–222.

[9] P. Herber, J. Fellmuth, and S. Glesner, "Model checking SystemC designs using timed automata," in *CODES+ISSS*, C. H. Gebotys and G. Martin, Eds. ACM, 2008, pp. 131–136.

[10] D. Tabakov and M. Y. Vardi, "Monitoring Temporal SystemC Properties," in *MEMOCODE*. IEEE, 2010, pp. 123–132.

[11] R. Jhala and R. Majumdar, "Software model checking," *ACM Comput. Surv.*, vol. 41, no. 4, 2009.

[12] F. Ivancic, I. Shlyakhter, A. Gupta, and M. K. Ganai, "Model checking c programs using f-soft," in *ICCD*. IEEE Computer Society, 2005, pp. 297–308.

[13] T. Ball, E. Bounimova, R. Kumar, and V. Levin, "Slam2: Static driver verification with under 4% false alarms," in *FMCAD*, R. Bloem and N. Sharygina, Eds. IEEE, 2010, pp. 35–42.

[14] G. J. Holzmann, "Software model checking with SPIN," *Advances in Computers*, vol. 65, pp. 78–109, 2005.

[15] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, "Counterexample-guided abstraction refinement for symbolic model checking," *J. ACM*, vol. 50, no. 5, pp. 752–794, 2003.

[16] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre, "Lazy abstraction," in *POPL*. ACM, 2002, pp. 58–70.

[17] E. M. Clarke, D. Kroening, N. Sharygina, and K. Yorav, "SATABS: SAT-Based Predicate Abstraction for ANSI-C," in *TACAS*, ser. LNCS, N. Halbwachs and L. D. Zuck, Eds., vol. 3440. Springer, 2005, pp. 570–574.

[18] E. M. Clarke, D. Kroening, and F. Lerda, "A Tool for Checking ANSI-C Programs," in *TACAS*, ser. LNCS, K. Jensen and A. Podelski, Eds., vol. 2988. Springer, 2004, pp. 168–176.

[19] K. L. McMillan, "Lazy abstraction with interpolants," in *CAV*, ser. LNCS, T. Ball and R. B. Jones, Eds., vol. 4144. Springer, 2006, pp. 123–136.

[20] A. Cimatti, I. Narasamdya, and M. Roveri, "Boosting Lazy Abstraction for SystemC with Partial Order Reduction," in *TACAS*, ser. LNCS, P. A. Abdulla and K. R. M. Leino, Eds., vol. 6605. Springer, 2011, pp. 341–356.

[21] M. Moy, F. Maraninchi, and L. Maillet-Contoz, "Pinapa: an extraction tool for SystemC descriptions of systems-on-a-chip," in *EMSOFT*, W. Wolf, Ed. ACM, 2005, pp. 317–324.

[22] D. Beyer, T. A. Henzinger, R. Jhala, and R. Majumdar, "The software model checker Blast," *STTT*, vol. 9, no. 5-6, pp. 505–525, 2007.

[23] D. Beyer and M. E. Keremoglu, "CPAchecker: A Tool for Configurable Software Verification," in *CAV*, ser. LNCS, G. Gopalakrishnan and S. Qadeer, Eds., vol. 6806. Springer, 2011, pp. 184–190.

[24] D. Kroening and G. Weissenbacher, "Interpolation-based software verification with wolverine," in *CAV*, ser. LNCS, G. Gopalakrishnan and S. Qadeer, Eds., vol. 6806. Springer, 2011, pp. 573–578.

[25] A. Cimatti, A. Griggio, A. Micheli, I. Narasamdya, and M. Roveri, "Kratos - a software model checker for SystemC," in *CAV*, ser. LNCS, G. Gopalakrishnan and S. Qadeer, Eds., vol. 6806. Springer, 2011, pp. 310–316.

[26] G. J. Holzmann, *The Spin model checker: primer and reference manual*. Add. Wesley, 2003.

[27] G. J. Holzmann and D. A. Peled, "An improvement in formal verification," in *7th IFIP WG6.1 Int. Conf. on Formal Description Techniques VII*, London, UK, UK, 1995, pp. 197–211.

[28] T. A. Henzinger, R. Jhala, R. Majumdar, and K. L. McMillan, "Abstractions from proofs," in *POPL*, N. D. Jones and X. Leroy, Eds. ACM, 2004, pp. 232–244.

[29] T. Ball and S. K. Rajamani, "Bebop: A symbolic model checker for boolean programs," in *SPIN*, ser. LNCS, K. Havelund, J. Penix, and W. Visser, Eds., vol. 1885. Springer, 2000, pp. 113–130.

[30] K. L. McMillan, *Symbolic model checking*. Kluwer, 1993.

[31] M. Sheeran, S. Singh, and G. Stålmarck, "Checking safety properties using induction and a sat-solver," in *FMCAD*, ser. LNCS, W. A. H. Jr. and S. D. Johnson, Eds., vol. 1954. Springer, 2000, pp. 108–125.

[32] P. Godefroid, *Partial-Order Methods for the Verification of Concurrent Systems - An Approach to the State-Explosion Problem*, ser. LNCS. Springer, 1996, vol. 1032.

[33] D. A. Peled, "All from one, one for all: on model checking using representatives," in *CAV*, ser. LNCS, vol. 697. Springer, 1993, pp. 409–423.

[34] A. Valmari, "Stubborn sets for reduced state generation," in *APN 90: Proceedings on Advances in Petri nets 1990*. New York, NY, USA: Springer-Verlag, 1991, pp. 491–515.

[35] M. Moy, "Techniques and tools for the verification of systems-on-a-chip at the transaction level," INPG, Grenoble, Fr, Tech. Rep., Dec 2005.

[36] D. Karlsson, P. Eles, and Z. Peng, "Formal verification of SystemC designs using a petri-net based representation," in *DATE*, G. G. E. Gielen, Ed. EDAA, Leuven, Belgium, 2006, pp. 1228–1233.

[37] G. Behrmann, A. David, K. G. Larsen, J. Håkansson, P. Pettersson, W. Yi, and M. Hendriks, "Uppaal 4.0," in *QEST*. IEEE, 2006, pp. 125–126.

[38] H. Garavel, C. Helmstetter, O. Ponsini, and W. Serwe, "Verification of an industrial SystemC/TLM model using LOTOS and CADP," in *MEMOCODE*. IEEE Computer Society, 2009, pp. 46–55.

[39] H. Garavel, F. Lang, R. Mateescu, and W. Serwe, "Cadp 2010: A toolbox for the construction and analysis of distributed processes," in *TACAS*, ser. LNCS, P. A. Abdulla and K. R. M. Leino, Eds., vol. 6605. Springer, 2011, pp. 372–387.

[40] K. Marquet, B. Jeannet, and M. Moy, "Efficient Encoding of SystemC/TLM in Promela," Verimag, Tech. Rep., 2010, tR-2010-7.

[41] D. Große, H. M. Le, and R. Drechsler, "Proving transaction and system-level properties of untimed SystemC TLM designs," in *MEMOCODE*. IEEE Computer Society, 2010, pp. 113–122.

[42] C.-N. Chou, C.-H. Hsu, Y.-T. Chao, and C.-Y. R. Huang, "Formal Deadlock Checking on High-level SystemC Designs," in *ICCAD*, 2010, pp. 794–799.

[43] S. Kundu, M. K. Ganai, and R. Gupta, "Partial order reduction for scalable testing of SystemC TLM designs," in *DAC*, L. Fix, Ed. ACM, 2008, pp. 936–941.

[44] C. Helmstetter, F. Maraninchi, and L. Maillet-Contoz, "Full simulation coverage for SystemC transaction-level models of systems-on-a-chip," *Formal Methods in System Design*, vol. 35, no. 2, pp. 152–189, 2009.

[45] A. Sen, "Concurrency-oriented verification and coverage of system-level designs," *ACM Trans. Design Autom. Electr. Syst.*, vol. 16, no. 4, p. 37, 2011.

[46] N. Blanc, D. Kroening, and N. Sharygina, "Scoot: A Tool for the Analysis of SystemC Models," in *TACAS*, ser. LNCS, C. Ramakrishnan and J. Rehof, Eds., vol. 4963. Springer, 2008, pp. 467–470.

[47] N. Blanc and D. Kroening, "Race analysis for SystemC using model checking," in *ICCAD*, S. R. Nassif and J. S. Roychowdhury, Eds. IEEE, 2008, pp. 356–363.

[48] K. Marquet, B. Karkare, and M. Moy, "A theoretical and experimental review of SystemC front-ends," in *FDL*, A. Morawiec and J. Hinderscheit, Eds. ECSI, 2010, pp. 124–129.

[49] K. Marquet and M. Moy, "PinaVM: a SystemC front-end based on an executable intermediate representation," in *EMSOFT*, L. P. Carloni and S. Tripakis, Eds. ACM, 2010, pp. 79–88.

[50] A. Cimatti, E. M. Clarke, F. Giunchiglia, and M. Roveri, "NuSMV: A New Symbolic Model Checker," *STTT*, vol. 2, no. 4, pp. 410–425, 2000.

[51] R. Bruttomesso, A. Cimatti, A. Franzén, A. Griggio, and R. Sebastiani, "The MathSAT 4 SMT Solver," in *CAV*, ser. LNCS, A. Gupta and S. Malik, Eds., vol. 5123. Springer, 2008, pp. 299–303.

[52] A. Armando, M. Benerecetti, and J. Mantovani, "Abstraction refinement of linear programs with arrays," in *TACAS*, ser. LNCS, O. Grumberg and M. Huth, Eds., vol. 4424. Springer, 2007, pp. 373–388.

[53] R. Jhala and K. L. McMillan, "Array abstractions from proofs," in *CAV*, ser. LNCS, W. Damm and H. Hermanns, Eds., vol. 4590. Springer, 2007, pp. 193–206.

Alessandro Cimatti is the head the Embedded Systems Unit of the Fondazione Bruno Kessler, Trento, Italy. He received a M.S. degree in Electronic Engineering from the University of Genova in 1988. He joined the Institute for Scientific and Technological Research in Trento, Italy, in 1990. He has been participating in and leading several technology transfer projects in the design and verification of safety critical systems. His research interests include formal verification methods, satisfiability modulo theories, and their application to requirements engineering, automated planning, monitoring, and diagnosis.

Iman Narasamdya is a post-doctoral researcher in the Embedded Systems Unit of the Fondazione Bruno Kessler, Trento, Italy. He received a Ph.D. degree in Computer Science in 2008 from the University of Manchester, U.K. From 2007 to 2008, he was a post-doctoral researcher at the Verimag laboratory, in Grenoble, France. In 2008, he joined the Embedded Systems Unit of the Fondazione Bruno Kessler. His research interests include software analysis and verification, and programming language design and implementation.

Marco Roveri is a senior researcher in the Embedded Systems Unit of the Fondazione Bruno Kessler, Trento, Italy. He received a Ph.D. degree in Computer Science from the University of Milano, Italy in 2002. He joined the Automated Reasoning Division of the Istituto Trentino di Cultura, Trento, Italy, in 1998, first as a consultant, and in 2002 as a full time researcher. In 2008 he was appointed as a senior researcher in the Embedded Systems Unit of the Fondazione Bruno Kessler, Trento, Italy. His research interests include automated formal verification of hardware and software systems, formal requirements validation of embedded systems, and automated model based planning.