Verifying SystemC: a Software Model Checking Approach

Alessandro Cimatti Andrea Micheli Iman Narasamdya Marco Roveri
Fondazione Bruno Kessler – FBK-irst – Embedded System Unit – Italy
{cimatti,amicheli,narasamdya,roveri}@fbk.eu

***Abstract*—SystemC is becoming a de-facto standard for the development of embedded systems. Verification of SystemC designs is critical since it can prevent error propagation down to the hardware. SystemC allows for very efficient simulations before synthesizing the RTL description, but formal verification is still at a preliminary stage. Recent works translate SystemC into the input language of finite-state model checkers, but they abstract away relevant semantic aspects, and show limited scalability.**

In this paper, we approach formal verification of SystemC by reduction to software model checking. We explore two directions. First, we rely on a translation from SystemC to a sequential C program, that contains both the mapping of the SystemC threads in form of C functions, and the coding of relevant semantic aspects (e.g. of the SystemC kernel). In terms of verification, this enables the "off-the-shelf" use of model checking techniques for sequential software, such as lazy abstraction.

Second, we propose an approach that exploits the intrinsic structure of SystemC. In particular, each SystemC thread is translated into a separate sequential program and explored with lazy abstraction, while the overall verification is orchestrated by the direct execution of the SystemC scheduler. The technique can be seen as generalizing lazy abstraction to the case of multi-threaded software with exclusive threads and cooperative scheduling.

The above approaches have been implemented in a new software model checker. An experimental evaluation carried out on several case studies taken from the SystemC distribution and from the literature demonstrate the potential of the approach.

I. Introduction

The development of System-on-Chips (SoCs) is often started by writing an executable model, using high-level languages such as SystemC [1]. Verification of SystemC designs is an important issue, since errors identified in such models can reveal errors in the specification and prevent error propagation down to the hardware.

SystemC is arguably becoming a de-facto standard, since it allows for high-speed simulation before synthesizing the RTL hardware description. However, formal verification of SystemC is still at a preliminary stage. In fact, a SystemC design is a very complex entity, that can be thought of as multi-threaded software, where scheduling is cooperative and carried out according to a specific set of rules [2], and the execution of threads is mutually exclusive.

There have been several works that have tried to apply model checking techniques to complement simulation [3]–[7]. These approaches map the problem of SystemC verification to some kind of model checking problem, but suffer from severe limitations. Some of them disregard significant semantic aspects, e.g., they fail to precisely model the SystemC scheduler or the communication primitives. Others show poor scalability of model checking, because of too many details included in the model.

In this paper we present an alternative approach to the verification of safety properties (in the form of program assertions) of SystemC designs, based on software model checking techniques [8]–[11]. The primary motivation is to investigate the effectiveness of such techniques, that have built-in abstraction capabilities, and have shown significant success in the analysis of sequential software.

We explore two directions. First, we rely on a translation from SystemC to a sequential C program, that contains both the mapping of the SystemC threads in form of C functions, and the coding of relevant semantic aspects (e.g. of the SystemC kernel). In terms of verification, this enables the "off-the-shelf" use of model checking techniques for sequential software.

However, the exploration carried out during software model checking treats in the same way both the code of the threads and the kernel model. This turns out to be a problem, mostly because the abstraction of the kernel is often too aggressive, and many refinements are needed to re-introduce abstracted details.

Thus, we propose an improved approach, that exploits the intrinsic structure of SystemC. In particular, each SystemC thread is translated into a separate sequential program and explored with lazy abstraction, i.e. by constructing an abstract reachability tree as in [8], [12]. The overall verification is orchestrated by the direct execution of the SystemC scheduler, with techniques similar to explicit-state model checking. This technique, in the following referred to as Explicit-Scheduler/Symbolic Threads (ESST) model checking, is not limited to SystemC: it lifts lazy abstraction to the general case of multi-threaded software with exclusive threads and cooperative scheduling.

We have implemented our approaches into a tool chain that includes a SystemC front-end derived from PINAPA [13], and a new software model checker, called SYCMC, using several extensions built on top of NUSMV and MATHSAT [14]–[16]. We have been experimenting the two approaches on a set of benchmarks taken and adapted from the SystemC distribution, and from other works that are concerned with the verification of SystemC designs. First, we have run several software model checkers on the sequential C programs resulting from the translation of SystemC designs. Finally, we have experimented with the new ESST model checking algorithm. The results, although preliminary, are promising. In particular, the ESST algorithm demonstrates dramatic speed ups over the first approach based on the verification of sequentialized C programs.

The structure of this paper is as follows. In Section II we introduce SystemC. In Section III we reduce model checking of SystemC designs to model checking of sequential C. In Section IV we present ESST model checking. In Section V we discuss some related work. We present the results of the experimental evaluation in Section VI. Finally, in Section VII we draw conclusions and outline some future work.

II. Background on SystemC

SystemC is a C++ library that consists of (1) a core language that allows one to model a SoC by specifying its components and architecture, and (2) a simulation kernel (or scheduler) that schedules and runs processes (or threads) of components. SoC components are modeled as SystemC modules (or C++ classes). Channels abstract communication between modules, while ports in a module are used to bind the modules with channels. The SystemC library provides primitive channels such as signal, mutex, semaphore and queue.

A module contains one or more threads describing the parallel behavior of the SoC design. The SystemC library also provides general-purpose events used for the synchronization between threads. A thread can suspend itself by waiting for an event `e`, i.e. by calling `wait(e)`, or by waiting for some specified time, i.e. by calling `wait(t)`, for some time unit `t` $\geq$ 0. A thread can perform immediate notification of an event `e`, by calling `e.notify()`, or delayed notification, by calling `e.notify(t)` for some time unit `t`.

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

```
SC_MODULE(producer) {
 private:
  int d;
  sc_event e;
 public:
  sc_in<int> p_in;
  sc_port<write_if> p_w;
  SC_HAS_PROCESS(producer);

  producer(sc_module_name name) : sc_module(name) {
   SC_THREAD(write);
   SC_METHOD(read); sensitive << p_in; dont_initialize();
  }

  void write() {
   int t;
   wait(SC_ZERO_TIME);
   while (1) {
    t = d;          // Save old value of d.
    p_w->put(d);    // Write d's value to consumer.
    wait(e);
    assert(d == t+1);
   }
  }

  void read() { d = p_in.read(); e.notify(); }
}

SC_MODULE(consumer), public write_if {
 private:
  int data;
  sc_event f, g;
 public:
  sc_out<int> p_out;
  sc_export<write_if> ex_w;
  SC_HAS_PROCESS(consumer);

  consumer(sc_module_name name) : sc_module(name) {
   ex_w(*this);
   SC_THREAD(compute);
   SC_THREAD(write_b);
  }

  void put(int d) { data = d; f.notify(); }

  void compute() {
   while (1) {
    wait(f); ++data; g.notify();
   }
  }

  void write_b() {
   while (1) {
    wait(g); p_out.write(data);
   }
  }
};

int main() {
  sc_signal<int> s;
  // Create producer and consumer instances.
  produce  * p = new producer("P");
  consumer * c = new consumer("C");
  // Interconnect signal.
  p->p_in(s); c->p_out(s);
  // Interconnect modules.
  p->p_w(c->ex_w);
  // Start simulation.
  sc_start();
}
```

Listing 1. Definition of a producer/consumer design in SystemC.

Listing 1 depicts an excerpt of a simple producer-consumer example in SystemC. It defines the `producer` that has two threads, `write` and `read`. The thread `write` sends the value stored in the variable `d` to the `consumer` by calling the function `put` in the `consumer`, and then wait for the event `e` to be notified. The method thread `read` reads from the channel bound to the input port `p_in` and notify the event `e`. It is sensitive to the input port `p_in`. A method thread only suspends itself by exiting the function and becomes runnable when the channel bound to the port is updated. The function `dont_initialize` makes the thread `read` not runnable at the beginning of simulation. The `consumer` consists of two threads: `compute` and `write_b`. The thread `compute` is triggered by the event `f` notified by the function `put` that was called by the `producer`. The interface `write_if` contains the signature of `put` and is derived from the SystemC interface. The thread `compute` increments the value sent by the producer, and then notifies the event `g` that subsequently activates the thread `write_b`. The thread `write_b` then writes the incremented value to the channel connecting the producer and the consumer through the port `p_out`. Finally, the `main` function shows that the producer `p` and the consumer `c` are connected via the signal channel `s`. The export construct of SystemC allows communication between components without any intermediate channel, as shown by the binding of port `p_w` of `producer` and port `ex_w` of `consumer`.

III. Model Checking SystemC via Sequentialization

In this section we describe the translation from SystemC designs into an equivalent sequential C programs by using the producer-consumer example introduced in the previous section.

A. Translating SystemC to C

In our translation each thread in the SystemC design is translated into a C function. Members of module instances,

```c
int d;    /* Global variable for producer. */
/* Events in the design */
int event_e; /* Status of event e. */
int event_f; /* Status of event f. */
int event_g; /* Status of event g. */
/* Local to thread producer::write() */
int write_pc;      /* Program counter. */
int write_state;   /* Status of thread. */
int event_write;   /* Status of thread event. */
int t_write;       /* Local variable t */
```

Listing 2. Excerpt of the C preamble.

channels, and events are translated into a set of global variables. We assume that the SystemC design does not contain any dynamic creation of such components. We also assume that each function call in the SystemC thread code can be inlined statically.

To model context switches that occurs during the SystemC simulation, for each thread t, we introduce the following supporting variables: (1) `t_pc` keeps track of the program counter of the thread; (2) `t_state` keeps track of the status of the thread, whose possible values are `SLEEP`, `RUNNABLE`, or `RUNNING`; (3) `event_t` describes the status of the event associated with the thread, whose possible values are `DELTA`, `FIRED`, `TIME`, or `NONE`; and (4) `event_time_t` keeps track of the notification time of the event associate with the thread. The status `DELTA` indicates that the event will be triggered at the transition from current delta cycle to the next one. The status `TIME` indicates that the event will be triggered at some time in the future. The status `FIRED` indicates that the event has just been triggered, while the status `NONE` indicates there is no notification or triggering applied to the event. Similarly, for each event e occurring in the design, we introduce a variable `event_e` whose values range over event status and a variable `time_event_e` that keeps track of the notification time. For succinctness of presentation, we do not prefix the above variables with the names of module instances that own the threads. Moreover, when the design has no time notification we omit the `TIME` status and the variable that keeps track of the notification time.

Member variables of a module instance are visible by all its threads. Thus, they are translated into global variables in the C program. For variables local to some thread, as context switches require saving and restoring such variables, we introduce for every local variable l of thread t a global variable l_t of the same type as l. Saving the value of l means assigning its value to l_t, while restoring the value of l means assigning l_t's value to l. Listing 2 shows the variables introduced for the thread `write` and for all the events of Listing 1.

Listing 3 shows the result of translating the thread `write` of `producer` into a C function. First, the function is annotated with program labels indicating the locations of context switches. The function starts with a jump table whose targets depend on the values of the program counter `write_pc` that points to the location at which the thread has to resume its execution. Second, we model calls to wait functions and their variants by the following instructions: (1) an assignment setting the thread's status to `SLEEP`; (2) an assignment setting the thread's program counter to the location where the thread has to resume its execution once it is woken up; (3) assignments saving the values of thread's local variables into the corresponding global variables introduced above; (4) a return statement; (5) a program label representing the location where the thread has to resume its execution; and (6) assignments restoring the values of thread's local variables. For example, for `wait(e)` in the thread `write`, we introduce the program label `WRITE_WAIT_2` and set the program counter `write_pc` to `WRITE_WAIT_2` before the function returns (see lines 25–32 of Listing 3). In the case of `wait(SC_ZERO_TIME)` in the thread `write`, the thread is suspended and will be woken up at the delta-cycle transition. To model this, we set the variable `event_write` to `DELTA`.

```c
 1 void write() {
 2   int t;
 3   /* Local jump table */
 4   if      (write_pc == WRITE_ENTRY)  goto WRITE_ENTRY;
 5   else if (write_pc == WRITE_WAIT_2) goto WRITE_WAIT_2;
 6   else if (write_pc == WRITE_WAIT_1) goto WRITE_WAIT_1;
 7  WRITE_ENTRY:
 8   /* wait(SC_ZERO_TIME); _BEGIN_ */
 9   write_state = SLEEP;
10   write_pc = WRITE_WAIT_1;
11   event_write = DELTA;
12   t_write = t; /* Save t. */
13   return;
14  WRITE_WAIT_1:
15   t = t_write; /* Restore t. */
16   /* wait(SC_ZERO_TIME); _END_ */
17   while (1) {
18     t = d;
19     /* inline consumer::put _BEGIN */
20     data = d;
21     event_f = FIRED;       /* f.notify() _BEGIN_ */
22     activate_threads();
23     event_f = NONE;        /* f.notify() _END_ */
24     /* inline consumer::put _END_ */
25     /* wait(e) _BEGIN_ */
26     write_state = SLEEP;
27     write_pc = WRITE_WAIT_2;
28     t_write = t; /* Save t. */
29     return;
30  WRITE_WAIT_2:
31     t = t_write; /* Restore t. */
32     /* wait(e) _END_ */
33     assert(d==t+1);
34   }
35 }
```

Listing 3. Sequential thread `write` of `producer`.

An event e can be specified to be notified at immediate time or at some time in the future. In the former case, every thread that depends on the notified event has to be triggered. To this end, we introduce for each thread t a function `is_t_triggered` that returns 1 if the thread is triggered, 0 otherwise. Now immediate notifications can be modeled by the following instructions: (1) an assignment setting the event's status to `FIRED`; (2) a list of queries checking if threads are triggered, and if they are triggered, their status are set to `RUNNABLE`; this list is represented by the function `activate_thread`; and (3) an assignment setting the event's status to `NONE`. Lines 21–23 of Listing 3 shows the translation of `f.notify()`. Listing 4 shows the code for thread activation. The notification by `e.notify(SC_ZERO_TIME)` is modeled similarly to `wait(SC_ZERO_TIME)`, that is, we set the variable `event_e` to `DELTA`. To model general time delayed notification, one needs a statement that assigns the delayed notification time to the variable associated with the event that keeps track of the notification time.

Next, we inline the function calls in the SystemC code. For instance, the inlining of the call `p_w->put(d)` in `write` is shown in lines 19–24 of Listing 3. As we will discuss later, function inlining can give advantages to the application of software model checking techniques, particularly in the encoding of the threads.

```
int is_write_triggered() {
  if ((write_pc == WRITE_WAIT_1)
      && (event_write == FIRED)) return 1;
  if ((write_pc == WRITE_WAIT_2)
      && (event_e == FIRED)) return 1;
  return 0;
}

void activate_threads() {
  if (is_write_triggered())    write_state = RUNNABLE;
  if (is_compute_triggered()) compute_state = RUNNABLE;
  if (is_write_b_triggered()) write_b_state = RUNNABLE;
  if (is_read_triggered())     read_state = RUNNABLE;
}
```

Listing 4. Thread activation.

A signal channel s is represented by two global variables s_old and s_new. Writing to and reading from a port bound to the channel is modeled as, respectively, an assignment to s_new and an assignment from s_old. For each channel, we include the update function of the channel in the resulting C program. For a signal s, the update function simply assigns s_old with the value of s_new if their values are different.

The SystemC scheduler is included in the C program resulting from the translation. The scheduler is shown in Listing 5. It consists of five phases: the initial phase, the evaluation phase, the update phase, the delta-notification phase, and the time phase. (We based the definition of the scheduler on [2])

In the initial phase all channels are updated by calling the corresponding update functions. The function `init_thread` changes the status of a thread to `SLEEP` if `dont_initialize` is specified for the thread. The function `fire_delta_events` simply changes the status of an event to `FIRED` if it was previously `DELTA`, while the function `reset_events` changes the status to `NONE`. Similarly for the function `fire_time_events`. In the evaluation phase, denoted by function `eval`, all runnable threads are run one at a time. Unlike the original SystemC scheduler that explores only one schedule, in the verification we want to explore all possible schedules. To this end, we use the function `nondet()` that returns a non-deterministic value.

The scheduler enters the update phase when there is no more runnable thread. In the update phase all channels are updated. The scheduler then moves to the delta-notification phase. This phase signifies the transition from the current delta phase to the next one. In this phase the scheduler triggers all events whose status are `DELTA`, and subsequently wakes up triggered events. The time phase is entered if there is no runnable thread after the delta-notification phase. In this phase the scheduler simply accelerates the simulation time. The scheduler quits if there are no more runnable threads. Note that, this encoding of the scheduler admits one impossible schedule where no runnable threads are selected to run. However, the existence of such a schedule is benign given we are focusing on the verification of safety properties.

To complete the translation, all variables related to threads and events must be initialized. The program counter is initialized to the entry label, for example, `write_pc` is initialized to `WRITE_ENTRY`. All variables whose values represent thread status are initialized to `RUNNABLE`, and all variables whose values represent event status are initialized to `NONE`. These initializations are performed in the function `init_model` called by the `main` function.

This translation from SystemC to sequentialized C preserves

```
void eval() {
  while (exists_runnable_thread()) {
    if (write_state == RUNNABLE && nondet())
      { write_state = RUNNING; write(); }
    if (compute_state == RUNNABLE && nondet())
      { compute_state = RUNNING; compute(); }
    if (write_b_state == RUNNABLE && nondet())
      { write_b_state = RUNNING; write_b(); }
    if (read_state == RUNNABLE && nondet())
      { read_state = RUNNING; read(); }
  }
}

void start_simulation() {
  update_channels(); /* Initialization phase. */
  init_threads();
  fire_delta_events();
  activate_threads();
  reset_events();
  do {
    eval(); /* Evaluation phase. */
    update_channels(); /* Update phase. */
    fire_delta_events(); /* Delta-notification phase. */
    activate_threads();
    reset_events();
    if (!exists_runnable_thread()) {
      fire_time_events(); /* Time-notification phase. */
      activate_threads();
      reset_events();
    }
  } while (exists_runnable_thread());
}

int main() {
  init_model(); start_simulation();
}
```

Listing 5. Sequential SystemC scheduler and `main`.

the behavior of the original SystemC design.

B. *Model Checking (SystemC as) C*

The translation from SystemC to C presented above opens up the possibility to reduce the verification of a SystemC design to the problem of verifying the translated C program. Verification of C programs is possible by using existing software model checkers, such as SATABS [17], BLAST [8], and CPACHECKER [18]. We notice that these model checkers implement approaches that are complementary to the ones that have been proposed in the past to verify SystemC.

Among the above approaches, one particularly promising is the idea of model checking via lazy abstraction [10]. The approach is based on the construction and analysis of an abstract reachability tree (ART) using predicate abstraction. The approach can be seen as combining an exploration of the control flow automaton (CFA) of the program with explicit-state techniques, while the data path is analyzed by means of predicate abstraction. (See also [8]–[11], [18] for a more thorough discussion) The ART represents reachable abstract states obtained by unwinding the CFA of the program. An ART node typically consists of a control flow location, a call stack, and a formula representing a region or data states (i.e. assignments to each variable of the program of a value in its domain).

An ART node is expanded by applying the strongest post operator followed by predicate abstraction to the region and the outgoing CFA edge of the location labelling the node [12], [18]. When the expansion reaches an error location, if the path from the root to the node with the error location is feasible, then the path is a counter-example witnessing the error (or assertion violation). Otherwise, the path is analyzed to discover new predicates to track and to determine the point in the ART where to start the refinement to discard the spurious behavior.

Predicate abstraction can benefit from advanced SMT techniques like [15] and [16]. Large block encoding (LBE) for lazy-abstraction has been proposed in [12] to reduce the number of paths (and nodes) in the ART that have to be explored independently. Intuitively, in LBE each edge in the CFA corresponds to a rooted directed acyclic graph (DAG) in the original CFA. Such an edge can be thought of as a summarization of the corresponding rooted DAG in the original CFA. In LBE function calls and loops in a program require block splitting. As we want to keep the number of blocks as small as possible, one can complementary apply function inlining to calls to non-recursive functions and loop unrolling to the loops whose bounds are known. The refinement can benefit from proof of unsatisfiability and from interpolation based techniques. For instance, in [11] it has been described an interpolation based refinement approach where the relevant predicates at each location of the infeasible path are inferred from the interpolant between the two formulas that define the prefix and the suffix of the path.

The idea of applying software model checking techniques to the C program resulting from the translation of SystemC is, to the best of our knowledge, novel. The hope is that the various abstraction techniques may provide some leverage to tackle the state explosion problem.

However, we remark that the exploration of the ART carried out during software model checking will treat in the same way both the code of the threads and the kernel model. In a sense, a general purpose technique is being applied to programs that have a very specific structure, resulting from the sequentialization of concurrency. In the next section, we propose a generalization to software model checking that exploits this feature of the analyzed problems.

IV. Explicit Scheduler + Symbolic Threads

In this section we propose a novel approach to the verification of SystemC designs. First, unlike the previous approach, here we decouple the scheduler from the threads. That is, the scheduler will no longer be part of the program, but is embedded in the model checking algorithm. Second, we combine the explicit model checking technique with the symbolic one based on lazy predicate abstraction. In this combination we still represent the state of each thread as a formula describing a region. But, unlike the classical lazy abstraction, we keep track of the states of scheduler explicitly. In the following, we refer to this technique as Explicit-Scheduler/Symbolic Threads (ESST) model checking. Fig. 1 shows an overview of this new approach.

We introduce several primitive functions to model SystemC synchronization mechanism and for interacting with the model checking algorithm. For example, the SystemC's wait functions `wait(`t`)` and `wait(`e`)` are modeled by primitive functions `wait(`t`)` and `wait_event(`e`)`, respectively. These primitive functions perform synchronization by updating the state of the scheduler. In the proposed algorithm the scheduler requires precise information about its state in order to schedule the threads. To this end, we assume that in the SystemC design the values of t and e in `wait(`t`)` and `wait_event(`e`)` can be determined statically. This assumption does not limit the applicability of

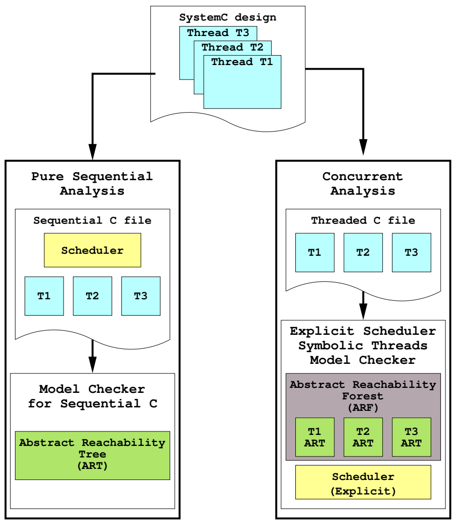

Fig. 1. An overview of the ESST approach.

our technique since, to the best of our knowledge, most real SystemC designs satisfy the assumption.

A. Abstract Reachability Forest

We build an *abstract reachability forest* (ARF) to describe reachable abstract states. An ARF consists of some ART's, each of which is obtained by unwinding the running thread. The connections between one ART with the others in an ARF describe context switches.

For a model with n threads, each node in the ARF is a tuple $(\langle q_1, s_1, \varphi_1\rangle, \ldots, \langle q_n, s_n, \varphi_n\rangle, \varphi, S)$, where (1) q_i, s_i, and φ_i are, respectively, the program location, the call stack, and the region of thread i, (2) φ is the region describing the data state of global variables, and (3) S is the state of scheduler. The state S does all the book keeping necessary to model the behavior of the scheduler. For example, it keeps track of the status of threads and events, the events that sleeping threads are waiting for, and the delays of event notifications.

To expand the ARF, we need to execute primitive functions and to explore all possible schedules. To this end, we introduce the function SEXEC that takes as inputs a scheduler state and a call to a primitive function f, and returns the updated scheduler state obtained from executing f. For example, the state $S' =$ SEXEC$(S,$`wait_event(`e`)`$)$ is obtained from the state S by changing the status of running thread to sleep, and noting that the now sleeping thread is waiting for an event e.

We also introduce the function SCHED that implements the scheduler. This function takes as an input a scheduler state and returns a set of scheduler states, each of which has exactly one running thread. These resulting states represent all possible schedules.

To describe the expansion of a node in ARF, we assume that there is at most one running thread in the scheduler state of the node. The rules for expanding a node $(\langle q_1, s_1, \varphi_1\rangle, \ldots, \langle q_n, s_n, \varphi_n\rangle, \varphi, S)$ are as follows:

E1. If there is a running thread i in S such that the thread performs an operation *op*, then the successor node is obtained in the following way:

- If *op* is *not* a primitive function, then the successor node is $(\langle q'_1, s'_1, \varphi'_1\rangle, \ldots, \langle q'_n, s'_n, \varphi'_n\rangle, \varphi', S')$ where $\varphi'_i = SP^{\pi}(\varphi_i \wedge \varphi, op)$, $\varphi'_j = SP^{\pi}(\varphi_j \wedge \varphi, \textsc{havoc}(op))$ for $j \neq i$, $\varphi' = SP^{\pi}(\varphi, op)$, $s'_k = s_k$ for all $k = 1, \ldots, n$, and $S' = S$. $SP^{\pi}(\varphi, op)$ computes the abstract strongest post condition w.r.t. precision π. In our case of predicate abstraction the precision π can contain (1) a set of predicates that are tracked for the global region φ, and (2) for all i, a set of predicates that are tracked for each thread region φ_i. HAVOC is a function that collects all global variables that are possibly updated by the operation *op*, and builds a new operation where these variables are assigned with new fresh variables. We do this since we do not want to leak variables local to the running thread in order to update the region of other threads.
- If *op* is a primitive function, then the new node is $(\langle q_1, s_1, \varphi_1\rangle, \ldots, \langle q_n, s_n, \varphi_n\rangle, \varphi, S')$ where $S' = \textsc{Sexec}(S, op)$.

E2. If there is no more running thread in S, then for each scheduler's state $S' \in \textsc{Sched}(S)$ we create a node $(\langle q_1, s_1, \varphi_1\rangle, \ldots, \langle q_n, s_n, \varphi_n\rangle, \varphi, S')$ such that the node becomes the root node of a new ART that is then added to the ARF. This represents the context switch that occurs when a thread gives the control back to the scheduler.

In the same way as the classical lazy abstraction, one stops expanding a node if the node is covered by other nodes. In our case we say that a node $(\langle q_1, s_1, \varphi_1\rangle, \ldots, \langle q_n, s_n, \varphi_n\rangle, \varphi, S)$ is *covered* by a node $(\langle q'_1, s'_1, \varphi'_1\rangle, \ldots, \langle q'_n, s'_n, \varphi'_n\rangle, \varphi', S')$ if (1) $q_i = q'_i$ and $s_i = s'_i$ for $i = 1, \ldots, n$, (2) $S = S'$, and (3) $\varphi \rightarrow \varphi'$ and $\bigwedge_{i=1,\ldots,n}(\varphi_i \rightarrow \varphi'_i)$ are valid. We also stop expanding a node if the conjuction of all thread regions and the global region is unsatisfiable.

We say that an ARF is *complete* if it is closed under the expansion rules described above and there is no node that can be expanded. An ARF is *safe* if it is complete and, for every node $(\langle q_1, s_1, \varphi_1\rangle, \ldots, \langle q_n, s_n, \varphi_n\rangle, \varphi, S)$ in the ARF such that $\varphi \wedge \bigwedge_{i=1,\ldots,n} \varphi_i$ is satisfiable, none of the locations $q_1, \ldots, q_n$ are error locations.

B. ARF construction

The construction of an ARF starts with a single ART representing reachable states of the main function. In the root node of that ART all regions are initialized with $True$, all thread locations are set to the entries of the corresponding threads, all call stacks are empty, and the only running thread in the scheduler's state is the main function. The main function then suspends itself by calling a primitive function that starts the simulation.

We expand the ARF using the rules E1 and E2 until either the ARF is complete or we reach a node where one of the thread's location is an error location. In the latter case we build a counterexample consisting of paths in the trees of the ARF and check if the counterexample is feasible. If it is feasible, then we have found a real counterexample witnessing that the program is unsafe. Otherwise, we use the spurious counterexample to discover predicates to refine the ARF.

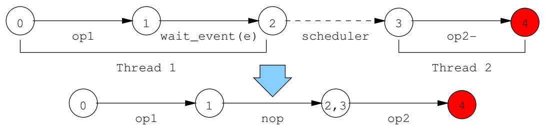

Fig. 2. An example error path.

C. Counterexample analysis and predicate discovery

The counterexample in our proposed technique is built in a similar way to that of in the classical lazy abstraction for sequential programs. In our case each call to a primitive function is replaced with a *nop* (no operation). The connections between trees induced by $\textsc{Sched}$ is removed and the two connected nodes are collapsed into one.

Let us consider the path represented in Fig. 2. There are two threads in this example. First, thread 1 moves from node 0 to node 1 with operation op_1, and then moves from node 1 to node 2 with `wait_event(e)` that makes thread 1 sleep and wait for the event `e` to be notified. The scheduler $\textsc{Sched}$ is then executed, and this execution creates a connection from node 2 to node 3, and also makes thread 2 as the running thread. Finally, thread 2 moves from node 3 to error node 4 with operation op_2. The counterexample is built by replacing the call to `wait_event(e)` labeling the transition from node 2 to 3 with *nop* and by collapsing nodes 2 and 3 into a new node 2,3. We thus obtain the path depicted in the lower part of Fig. 2. This final path corresponds to a "standard" path in the pure sequential software model checker, and is the path we consider for the counterexample analysis.

When the formula corresponding to the error path built above is unsatisfiable, then the proof of unsatisfiability is analyzed in order to extract new predicates. These predicates are then used to refine the abstraction in order to rule out this unfeasible error path in the expansion of ARF. For this purpose we re-use the same techniques used in the sequential case, e.g. Craig interpolants and unsatisfiable core. The newly discovered predicates are then used to update the precision. Depending on the nature of the predicates, they can be associated to all threads globally, which is the precision of the global region, or to a specific thread, which is the precision of the thread region. Due to lack of space, we refer the reader to [19] for a more thorough discussion of the refinement process.

D. Parametric Summarization of Control Flow Automata

CFA summarization based on the large block encoding (LBE) has been introduced in [12]. The encoding can also be applied to summarize the CFA representing a thread.

We define a parameterized version of the LBE w.r.t. a set $\Gamma \subseteq Ops$ of operations that is used to prevent the creation of a large block. The algorithm to compute parametric LBE is a variant of the algorithm described in [12]. First, a CFA is a tuple (L, G) where L is a set of control locations and $G \subseteq L \times Ops \times L$ is a set of edges. Without loss of generality, we assume that the CFA has at most one error location denoted by l_E. The LBE of Γ-*CFA Summarization* consists of the application of the rules we describe below: we first apply the rule R1, and then repeatedly apply the rule R2 and R3 until none of them are applicable.

R1. We remove all edges $(l_E, *, *)$ from G. This rule transforms the error location into a sink location.

R2. If $(l_1, op, l_2) \in G$ such that $l_1 \neq l_2$, $op \notin \Gamma$, l_2 has no other incoming edges, and for all $(l_2, op_i, l_i) \in G$ we have $op_i \notin \Gamma$, then $L = L \setminus \{l_2\}$ and $G = (G \setminus \{(l_1, op, l_2)\}) \cup \{(l_1, op; op_i, l_i) | \text{for all } i\}$. If the current operation op, or one of the outgoing operations is in Γ, then we stop summarizing the current block.

R3. If $(l_1, op_1, l_2) \in G$, $(l_1, op_2, l_2) \in G$, and none of op_1, op_2 are in Γ then $G = (G \setminus \{(l_1, op_1, l_2), (l_1, op_2, l_2)\}) \cup \{(l_1, op_1 \| op_2, l_2)\}$. Intuitively, if there is a choice and none of the two outgoing operations are in Γ, then we join the operations.

Since the parameter of summarization only prevents the creation of large blocks, the correctness of summarization as stated in [12] still holds for the above rules.

V. Related Work

There have been some works on the verification of SystemC designs. Scoot [20] is a tool that extracts from a SystemC design a flat C++ model that can be analyzed by SatAbs [17]. The SystemC scheduler itself is included in the flat model. Scoot, to the best of our knowledge, has only been used for race analysis [21], and for synthesizing a static scheduler to speed up simulation [22]. Our work on embedding the scheduler into the model-checking algorithm can benefit from the techniques described in [21] for reducing the number of schedules to explore.

CheckSyC [4] is a tool used for property and equivalence checking, and for simulation of SystemC designs. It relies on SAT based bounded model checking (BMC) and thus does not support unbounded loops. Moreover CheckSyC does not support SystemC constructs that have no correspondence in RTL, like channels.

Lussy [3] is a toolbox for the verification of SystemC designs at TLM. The tool extracts from a SystemC design a set of parallel automata that captures the semantics of the design, including the SystemC scheduler. These automata are then translated into Lustre or SMV model for the verification. The results reported in [23] show that the approach does not scale. An extension for the use of Spin is discussed in [6]. However, this translation is manual. Moreover, it is bound to not scale-up when the SystemC design requires to model nondeterministic signals with a large domain like e.g. an integer. For us, this is not a problem since we model them symbolically.

In [7] the SystemC design is encoded as a network of timed automata where the synchronization mechanism is modeled through channels. The execution semantics is specified through a pre-determined model of the scheduler, and by means of templates for events and threads. The resulting network of automata is verified using the UPPAAL model checker. This approach only supports bounded integer variables.

Formal verification of SystemC designs by abstracting away the scheduler, that is encoded in each of the threads, has been reported in [5]. This work does not handle channel updates and time delays. Our translation from SystemC to C can adopt the technique in the paper to simplify the resulting C program.

Works on the verification of multi-threaded C programs are related to our work. Software model checkers for multi-threaded programs such as Verisoft [24] and Zing [25] explore states and transitions using explicit enumeration. Although several state space reduction techniques (e.g. partial order reduction [26] and transaction based methods [27]) have been proposed, they do not scale well because of the state explosion caused by the thread interleaving. Extensions of the above techniques by using symbolic encodings [28] combined with bounded context switches [29] and abstraction [30] have been proposed. In [31] an asynchronous modeling is used to reduce the size of BMC problems. All of these techniques can be applied to the verification of SystemC designs by properly encoding the semantics of the SystemC scheduler. Our approach can benefit from these optimizations. In particular we expect that partial-order reduction that can reduce the number of schedules to explore will lead to dramatic improvements, but we leave it as future work.

VI. Experimental Evaluation

We have implemented a tool chain that supports the SystemC verification approaches presented in this paper. The front-end for handling SystemC is an extended version of Pinapa [13] modified to generate the flattened pure sequential C program described in Section III, and the output suitable for the new algorithm described in Section IV.

To deal with the sequential C program, we have implemented a new software model checker for C that we call SyCMC. Inspired by Blast [8], SyCMC implements lazy predicate abstraction. Furthermore, SyCMC also provides LBE and the Γ-CFA summarization described before. SyCMC is built on top of an extended version of NuSMV [14] that integrates the MathSAT [32] SMT solver and provides advanced algorithms for performing predicate abstraction by combining BDDs and SMT formulas [15], [16]. The new ESST model-checking algorithm is implemented within SyCMC. In SyCMC as well as in ESST each time we expand an ART (ARF) node we perform the check to verify whether the newly generated node is covered by another ART (ARF) node. Thus, it is fundamental to perform this check as efficiently as possible. Similarly to CpaChecker, in SyCMC as well as in ESST we use BDDs to represent the regions, and we exploit them for efficiently checking whether a node is covered by another node.

A. Results

We used benchmarks taken and adapted from the SystemC distribution [1], from [23], and from [33] to experiment with our approaches. To the best of our knowledge, none of the tools used in [3], [4], [7] is available for comparison. We first experimented with the translation of SystemC models to C programs, by running the benchmarks on the following model checkers: SatAbs [17], Blast [8], CpaChecker presented in [12], and SyCMC. We then experimented the ESST algorithm of

SyCMC on the same set of benchmarks. As the model checkers feature a number of verification options, we only consider what turned out to be the best options for the benchmarks. For Blast we used the `-foci` option, while for CpaChecker and for SyCMC we applied LBE, depth first node expansion with global handling of predicates, and restarting ART from scratch when new predicates are discovered. We have experimented the tools on an Intel-Xeon DC 3GHz running Linux, equipped with 4GB of RAM. We fixed the time limit to 1000 seconds, and the memory limit to 2GB.

The results of experiments are shown on Table I. In the second column we report S, U, or - to indicate that the verification status of the benchmark is safe, unsafe, or unavailable respectively. The unavailability of the status is due to time or memory out. In the remaining columns we report the running time in seconds. We use T.O. for time out, M.O. for memory out, and N.A. for not available.

The results show that the translation approach is feasible, but the model checkers often reached timeout. This is because the presence of the scheduler in the C program enlarges the search space that has to be explored by the model checkers. Moreover, we noticed that several iterations of refinement are needed to discover predicates describing the status of the scheduler in order to rule out spurious counterexamples. We notice that, as far as these benchmarks are concerned, CpaChecker outperforms Blast, while we have cases where SyCMC performs better than CpaChecker, and others where it performs worse. This is explained by the fact that the search in the two model checkers, although similar may end-up exploring paths in a different order and thus discovering different sets of predicates.

The table clearly shows that the ESST algorithm outperforms the other four approaches in most cases. In the case of `pipeline` design CpaChecker and SyCMC outperform the ESST algorithm. It turns out that for the verification of this design precise details of the scheduler are not needed. CpaChecker and SyCMC are able to exploit this characteristic and thus they end up exploring less abstract states than ESST. Indeed, for this design the ESST algorithm needs to explore many possible schedules that can be reduced by using techniques like partial-order reduction. For the `mem-slave` design SatAbs outperforms other model checkers. SyCMC and ESST employ a precise Boolean abstraction in the expansion of the ART. Such an abstraction is expensive when there are a large number of predicates involved. For this design, SyCMC and ESST already discovered about 70 predicates in the early refinement steps. SatAbs also discovered a quite large number of predicates (51 predicates). However, it performs a cheap approximated abstraction that turns out to be sufficient for the verification of this design.

All the benchmarks and the executable to reproduce the results reported in this paper are available at http://es.fbk.eu/people/roveri/tests/fmcad2010.

		Sequentialized				ESST
Name	V	SatAbs	Blast	CpaC.	SyCMC	SyCMC
toy1	S	22.790	T.O.	282.230	57.300	1.990
toy2	U	28.050	T.O.	621.120	35.300	0.690
toy3	U	20.290	T.O.	141.780	22.390	0.190
token-ring1	S	16.520	97.2000	14.590	36.990	0.010
token-ring2	S	62.240	888.2900	30.330	540.160	0.090
token-ring3	S	152.360	T.O.	141.860	T.O	0.190
token-ring4	S	602.300	T.O.	911.300	T.O	0.400
token-ring5	S	T.O.	T.O.	T.O.	T.O	1.000
token-ring6	S	T.O.	T.O.	T.O.	T.O	2.500
token-ring7	S	T.O.	T.O.	T.O.	T.O	6.390
token-ring8	S	T.O.	T.O.	T.O.	T.O	18.400
token-ring9	S	T.O.	T.O.	T.O.	T.O	54.290
token-ring10	S	T.O.	T.O.	T.O.	T.O	201.980
token-ring11	-	T.O.	T.O.	T.O.	T.O	M.O
transmitter1	U	2.230	1.2700	11.850	6.200	0.010
transmitter2	U	26.920	29.4000	18.210	640.750	0.010
transmitter3	U	61.460	501.3500	44.320	176.290	0.010
transmitter4	U	190.620	T.O.	113.490	T.O	0.090
transmitter5	U	472.180	T.O.	296.580	T.O	0.190
transmitter6	U	T.O.	T.O.	969.530	T.O	0.500
transmitter7	U	T.O.	T.O.	T.O.	T.O	1.390
transmitter8	U	T.O.	N.A.	T.O.	T.O	3.690
transmitter9	U	T.O.	N.A.	T.O.	T.O	11.690
transmitter10	U	T.O.	T.O.	T.O.	T.O	40.590
transmitter11	U	T.O.	T.O.	T.O.	T.O	150.480
transmitter12	-	T.O.	T.O.	T.O.	T.O	M.O
pipeline	S	T.O.	T.O.	130.610	178.490	T.O
kundu1	S	139.440	T.O.	232.310	T.O	2.900
kundu2	U	41.500	245.8500	57.160	T.O	0.900
kundu3	U	110.550	T.O.	129.370	T.O	2.900
bistcell	S	36.600	T.O.	10.560	38.000	1.090
pc-sfifo1	S	4.260	46.6500	13.110	7.690	0.300
pc-sfifo2	S	5.210	300.3800	28.490	34.790	0.300
mem-slave	S	77.210	T.O.	T.O.	T.O	677.010

TABLE I: Results for experimental evaluation.

B. Limitations

The approaches presented in this paper assume that the SystemC design does not contain any dynamic creation of threads, channels, and module instances. In particular, in the sequentialization approach the encoding of the scheduler requires those components to be known a priori. For example, to encode the evaluation and the channel update phases (the functions `eval` and `update_channels`, respectively) one needs to know all threads and channels that are involved in the simulation. In the threaded C approach we assume the values of t and e in `wait(`t`)` and `wait_event(`e`)` can be determined statically. Similarly for the translation to threaded C and in the ESST algorithm, at the moment we do not support dynamic creation of threads, channels, and module instances. It turns out that also the SystemC front-end we use for our translator suffers of these limitations. Indeed, Pinapa parses the SystemC design and executes it until the point just before the simulation begins. At that point Pinapa gives access to the abstract syntax tree (AST) of the design and to all the ground SystemC objects (i.e. module instances, channels, and threads) of the design. We remark that, these limitations do not affect the applicability of the proposed techniques since, to the best of our knowledge, most real SystemC design satisfy this assumption.

The Pinapa SystemC front-end at the current stage of development suffers of many other limitations. For example, as far as we know, it does not recognize all SystemC transaction-level modeling (TLM) constructs and does not fully support function pointers. Because of these limitations, our translator from SystemC to sequential C (and also to threaded C) does not handle such constructs either. For the experiments presented in this paper we extended Pinapa to handle simple TLM constructs like `sc_export`. Support for additional SystemC constructs can be added to PINAPA with a reasonable effort.

As far as the limitations of our translator are concerned, we do not yet support rich C++ features like standard template library (STL) data structures and respective constructs, and we do not yet support pointers, arrays, and dynamic creation of objects. To this end, we remark that most of the software model checkers currently available are not able to fully support all of them. We remark that, our translator can be extended to support such constructs wit a reasonable effort.

Finally, the new SYCMC and ESST model checkers are not yet able to handle designs that use complex data types (like e.g. records), pointers, arrays, dynamic creation of objects, and recursive function. However, support for all these constructs is currently argument of future extensions of the tools.

VII. CONCLUSIONS AND FUTURE WORK

In this paper we have presented two novel approaches aiming at lifting software model checking techniques to the verification of SystemC designs. We first presented a conversion of a SystemC design into an C program that can be verified by any off the shelf software model checker for C. Second, we presented a novel model checking algorithm that combines an explicit model checking technique to model the states of SystemC scheduler with lazy abstraction. Both approaches have been implemented in a tool set and an experimental evaluation was carried out showing the potential of the approach and the fact that the new algorithm outperforms the first approach.

As future work, we will investigate the applicability of static and dynamic partial order techniques to reduce the number of paths to explore. We will extend the set of primitives to interact with the scheduler to better handle TLM constructs. Moreover, we will investigate the possibility to handle the scheduler semi-symbolically by enumerating possible next states exploiting SMT techniques as to eliminate the current limitations of the ESST approach. Finally, we will also extend our back-end to support richer data like e.g. arrays [34], [35].

ACKNOWLEDGMENTS

We thank A. Griggio for the fruitful discussions on LBE and on the behavior of CPACHECKER. This work is sponsored by the EU grant FP7-2007-IST-1-217069 COCONUT.

REFERENCES

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

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

[3] 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.

[4] D. Große and R. Drechsler, "CheckSyC: An Efficient Property Checker for RTL SystemC Designs," in *ISCAS (4)*. IEEE, 2005, pp. 4167–4170.

[5] D. Kroening and N. Sharygina, "Formal Verification of SystemC by Automatic Hardware/Software Partitioning," in *MEMOCODE*. IEEE, 2005, pp. 101–110.

[6] C. Traulsen, J. Cornet, M. Moy, and F. Maraninchi, "A SystemC/TLM Semantics in Promela and its Possible Applications," in *SPIN*, ser. LNCS, vol. 4595. Springer, 2007, pp. 204–222.

[7] P. Herber, J. Fellmuth, and S. Glesner, "Model Checking SystemC Designs using Timed Automata," in *CODES+ISSS*. ACM, 2008, pp. 131–136.

[8] 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.

[9] K. L. McMillan, "Lazy Abstraction with Interpolants," in *CAV*, ser. LNCS, vol. 4144. Springer, 2006, pp. 123–136.

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

[11] T. A. Henzinger, R. Jhala, R. Majumdar, and K. L. McMillan, "Abstractions from Proofs," in *POPL*. ACM, 2004, pp. 232–244.

[12] D. Beyer, A. Cimatti, A. Griggio, M. E. Keremoglu, and R. Sebastiani, "Software Model Checking via Large-Block Encoding," in *FMCAD*. IEEE, 2009, pp. 25–32.

[13] M. Moy, F. Maraninchi, and L. Maillet-Contoz, "Pinapa: An Extraction Tool for SystemC Descriptions of Systems-on-a-Chip," in *EMSOFT*. ACM, 2005, pp. 317–324.

[14] 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.

[15] R. Cavada, A. Cimatti, A. Franzén, K. Kalyanasundaram, M. Roveri, and R. K. Shyamasundar, "Computing Predicate Abstractions by Integrating BDDs and SMT Solvers," in *FMCAD*. IEEE, 2007, pp. 69–76.

[16] A. Cimatti, J. Dubrovin, T. Junttila, and M. Roveri, "Structure-aware Computation of Predicate Abstraction," in *FMCAD*. IEEE, 2009, pp. 9–16.

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

[18] D. Beyer, T. A. Henzinger, and G. Théoduloz, "Configurable Software Verification: Concretizing the Convergence of Model Checking and Program Analysis," in *CAV*, ser. LNCS, vol. 4590. Springer, 2007, pp. 504–518.

[19] A. Cimatti, A. Micheli, I. Narasamdya, and M. Roveri, "Verifying SystemC: a Software Model Checking Approach," FBK-irst, Tech. Rep., 2010, http://es.fbk.eu/people/roveri/tests/fmcad2010.

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

[21] N. Blanc and D. Kroening, "Race Analysis for SystemC using Model Checking," in *ICCAD*. IEEE, 2008, pp. 356–363.

[22] ——, "Speeding Up Simulation of SystemC using Model Checking," in *SBMF*, ser. LNCS, vol. 5902. Springer, 2009, pp. 1–16.

[23] M. Moy, "Techniques and Tools for the Verification of Systems-on-a-Chip at the Transaction Level," INPG, Grenoble, Fr, Tech. Rep., Dec 2005.

[24] P. Godefroid, "Software Model Checking: The VeriSoft Approach," *F. M. in Sys. Des.*, vol. 26, no. 2, pp. 77–101, 2005.

[25] T. Andrews, S. Qadeer, S. K. Rajamani, J. Rehof, and Y. Xie, "Zing: A Model Checker for Concurrent Software," in *CAV*, ser. LNCS, vol. 3114. Springer, 2004, pp. 484–487.

[26] C. Flanagan and P. Godefroid, "Dynamic Partial-Order Reduction for Model Checking Software," in *POPL*. ACM, 2005, pp. 110–121.

[27] S. D. Stoller and E. Cohen, "Optimistic Synchronization-based State-Space Reduction," *F. M. in Sys. Des.*, vol. 28, no. 3, pp. 263–289, 2006.

[28] V. Kahlon, A. Gupta, and N. Sinha, "Symbolic Model Checking of Concurrent Programs using Partial Orders and On-the-Fly Transactions," in *CAV*, ser. LNCS, vol. 4144. Springer, 2006, pp. 286–299.

[29] S. Qadeer and J. Rehof, "Context-Bounded Model Checking of Concurrent Software," in *TACAS*, ser. LNCS, vol. 3440. Springer, 2005, pp. 93–107.

[30] I. Rabinovitz and O. Grumberg, "Bounded Model Checking of Concurrent Programs," in *CAV*, ser. LNCS, vol. 3576. Springer, 2005, pp. 82–97.

[31] M. K. Ganai and A. Gupta, "Efficient Modeling of Concurrent Systems in BMC," in *SPIN*, ser. LNCS, vol. 5156. Springer, 2008, pp. 114–133.

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

[33] S. Kundu, M. K. Ganai, and R. Gupta, "Partial Order Reduction for Scalable Testing of SystemC TLM Designs," in *DAC*. ACM, 2008, pp. 936–941.

[34] A. Armando, M. Benerecetti, and J. Mantovani, "Abstraction Refinement of Linear Programs with Arrays," in *TACAS*, ser. LNCS, vol. 4424. Springer, 2007, pp. 373–388.

[35] R. Jhala and K. L. McMillan, "Array Abstractions from Proofs," in *CAV*, ser. LNCS, vol. 4590. Springer, 2007, pp. 193–206.