Efficient Encoding of SystemC/TLM in Promela—Full Version

Kevin Marquet and Bertrand Jeannet and Matthieu Moy

Verimag Research Report n^o TR-2010-7

November 22, 2010

Efficient Encoding of SystemC/TLM in Promela—Full Version

Kevin Marquet and Bertrand Jeannet and Matthieu Moy

Verimag
Centre quation - 2, avenue de Vignate 38610 Gires - FRANCE

November 22, 2010

Abstract

To deal with the ever growing complexity of Systems-on-Chip, designers use models early in the design flow. SystemC is a commonly used tool to write such models. In order to verify these models, one thriving approach is to encode its semantics into a formal language, and then to verify it with verification tools. Various encodings of SystemC into formal languages have already been proposed, with different performance implications. In this paper, we investigate a new, automatic, asynchronous means to formalize models. Our encoding supports the subset of the concurrency and communication constructs offered by SystemC used for high-level modeling. We increase the confidence in the fact that encoded programs have the same semantics as the original one by model-checking a set of properties. We give experimental results on our formalization and compare with previous works.

Keywords: SystemC, model-checking, encoding, spin, promela

Reviewers: Florence Maraninchi

Notes:

How to cite this report:

@techreport { verimag-TR-2010-7,
title = { Efficient Encoding of SystemC/TLM in Promela—Full Version},
author = { Kevin Marquet and Bertrand Jeannet and Matthieu Moy},
institution = { Verimag Research Report },
number = {TR-2010-7},
year = { 2010},
note = { }
}

1 Introduction

As the complexity of embedded systems grows, the need for new methods has appeared for the co-design of hardware and software. Indeed, low-level hardware description languages such as VHDL and Verilog simulate slowly, can hardly be used to design complex systems and therefore make early software development difficult. Consequently, higher-level modeling tools have appeared, allowing hardware and software descriptions.

Transaction-Level Modeling [4] (TLM) is an approach in which the architecture and the behavior of a System-on-Chip (SoC) are described in an executable model, but the micro-architecture details and precise timing behavior are abstracted away. SystemC [20] has become the *de facto* standard for TLM modeling. It contains a simulation kernel that can execute concurrent processes communicating through channels and shared variables, using C++ libraries. In this paper, we are interested in TLM programs, written in SystemC. We focus on the subset of SystemC needed for TLM modeling, leaving apart the constructs originally introduced in SystemC to write lower-level programs (like RTL).

SystemC descriptions are C++ concurrent programs that can be tested and/or verified in order to detect design flaws. Verifying a concurrent program can be done with various approaches. One thriving approach is to describe its semantics formally, and then to verify this semantics using verification tools. The first step is called *model extraction* and leads to the translation of the program into a formal representation, and the second step is the verification performed on the formal representation. Different representations can be chosen, that model differently time and concurrency, and that are connected to different verification tools.

This paper focuses on the issue of *model extraction*, in the context of the verification of SoC modeled as System C concurrent programs. Our contributions are as follows:

1) We present **new encoding principles** in section 4 for the extraction of formal representations from SystemC programs, and in particular for modeling the semantics of SystemC scheduler. We argue that this encoding is simple and elegant, although it involves some subtle points. Its main goal is however to favor the efficiency of verification tools. Moreover, this extraction is performed in a fully **automatic** way by our verification chain. The implementation is open-source and available from `http://gitorious.org/pinavm`.
2) In order to **validate their correctness**, we define properties that must hold for an encoding to be valid. These properties and how they are tested are detailed in section 5.
3) At last, section 6 presents **experimental results** on SystemC examples translated to *Promela*, the asynchronous formalism used as input to the SPIN model-checker. Our results show major improvements over past similar works, thanks to the fact that our encoding does not introduce complex behaviors limiting the applicability of formal verification tools. We show in particular a tremendout reduction of the number of states that SPIN needs to explore.

Before presenting these, we present SystemC in section 2 and compare our approach to related works in section 3.

2 SystemC

We give a very partial overview of SystemC, focusing on the points that are relevant for this paper.

A SystemC program defines an *architecture*, i.e. a set of components and connections between them, and a *behavior*, i.e. components have a behavior defined by one or several processes and communicate with each other through ports. Once the architecture is defined (*elaboration phase* performed at the beginning of execution), the *simulation phase* starts: processes execute according to the SystemC scheduling policy. As an example, figure 1 shows a SystemC module containing two processes, one waiting for an event, the other notifying it.

The actual scheduler includes a notion of δ-cycles [20], inspired from traditional HDL languages, which we do not consider here, since it is not useful for TLM models (but this implies that we do not support SystemC constructs like wait(SC_ZERO_TIME), which makes a process wait until the next evaluation phase). We focus on the following constructs of SystemC, which are the basis for TLM modeling:

wait(d: int) Stops executing the current process, yields back the control to the scheduler and makes the current process to wait for the given duration.

```
SC_MODULE(mytop) {
    sc_event e;
    SC_CTOR(mytop) {
       SC_THREAD(myFctP); SC_THREAD(myFctQ);
    }
    void myFctP() {...; wait(e); ... }
    void myFctQ() {...; e.notify(); ... }
}
```

Figure 1: A basic SystemC module

wait(e: event) Stops executing the current process, yields back the control to the scheduler and makes the current process to wait for the event to occur. SystemC also allows the constructs **wait(e1 & e2)** and **wait(e1 | e2)** to wait for conjunctions and disjunctions of events.

event.notify() Makes processes waiting for the specified event eligible (without stopping the current process).

event.notify(delay: int) Triggers a notification after the given delay. In SystemC, only the earliest timed notification is kept, which simplifies the semantics of this primitive.

SystemC scheduling follows a *non-preemptive* scheduling policy. When several processes are eligible at the same time, the scheduler runs them in an unspecified order.

Concerning communications between process, we use shared variables to model several threads belonging to the same module communicating by accesses to the fields of the module. Communication channels allows to exchange information in a more complex way. In TLM, channels are modeled with SystemC modules, and the standard way to perform communication is to perform method calls from a module to another, using the C++ interfaces defined in the standard [21]. Technically, these method calls are done through ports (or sockets in TLM-2), but the underlying semantics is basically the one of a call. We therefore focus on the notion of method calls, and do not provide special modeling for components like `sc_signals` and `sc_fifos`. Our implementation does not (yet) manage TLM ports explicitly, but require the function calls from modules to modules to be done in plain C++ (only the syntax differs).

Restricting ourselves to a strict subset of SystemC implies that we cannot handle SystemC programs written using specific constructs outside our subset, but it also makes our approach more general in the sense that it could easily be adapted to other discrete-event cooperative simulator (like the cooperative version of jTLM [2]).

3 Overview of the problem and Related Works

General overview

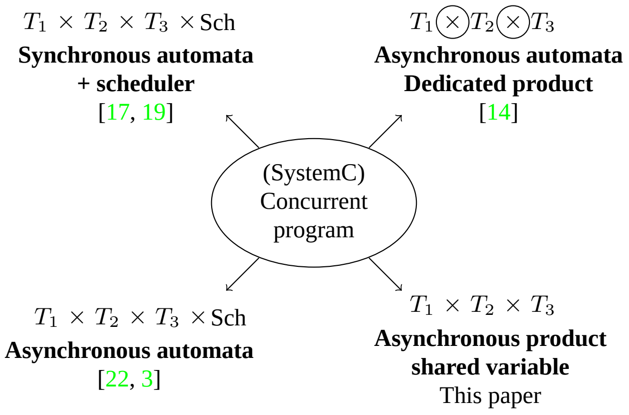
The challenge raised by formal verification of SystemC models is that SystemC has not been designed for this purpose. Few verification tools are available for general C++ programs, especially when the goal is to check *functional properties*. Moreover, a general verifier would have to reanalyze the SystemC class library and to rediscover by itself its high-level semantics. For these reasons, most related work proceeds differently, by first translating and abstracting a SystemC program to formal models accepted by the targeted verification tools, and then applying verification tools on the simplified model.

In other words, while most verification tools parse the user's code and *translate* it, the SystemC library itself is never parsed. Instead, its semantics is directly built-in the tool.

Representation of the SystemC scheduler

Modeling the semantics of the SystemC library reduces mainly to modeling the SystemC scheduler. Three options can be imagined to represent the scheduler in a formal representation: 1. choose the deterministic behavior corresponding to the reference implementation described in the SystemC standard [20], and model it, 2. model a non-deterministic scheduler as an explicit additional process, 3. or model it in the semantics of the synchronization instructions (typically the ones described above).

Choosing arbitrarily a specific, deterministic scheduler allows only to explore a subset of the behaviors. We do not want such restriction and therefore do not consider solution 1.



$T_1 \times T_2 \times T_3 \times \mathrm{Sch}$
Synchronous automata + scheduler
[17, 19]
$T_1 \otimes T_2 \otimes T_3$
Asynchronous automata Dedicated product
[14]
(SystemC) Concurrent program
$T_1 \times T_2 \times T_3 \times \mathrm{Sch}$
Asynchronous automata
[22, 3]
$T_1 \times T_2 \times T_3$
Asynchronous product shared variable
This paper

Figure 2: Different approaches for translating SystemC programs into other formalisms

Solution 2 is interesting as it does not restrict the set of possible behaviors. This is the solution considered in [17]. However, encoding the scheduler as a special process interacting with the SystemC processes complexifies the behavior of the global system. Typically, such an encoding induces additional communications between processes, compared to the original SystemC semantics. For instance, the encoding of the **event.notify()** primitive is likely to induce a context-switch (as it changes the state of the scheduler), which does not occur in the original SystemC semantics. The bad consequence is that such additional communications may prevent verification tools to perform powerful optimizations. Typically, partial-order reduction relies on a notion of "independent transitions", and cannot be applied if the notion of "transition" of the model does not correspond to the notion of atomic sections in SystemC.

Consequently, we have chosen the approach of point 3: we do not encode the scheduler as an explicit process composed in *parallel* with the SystemC processes. Instead, we integrate the scheduler in the semantics of the synchronization primitives that are used *sequentially* inside each SystemC process, without introducing any "artificial" context-switches.

Related work

LusSy [17] is a prototype of a complete verification chain. It encodes the processes *and* the scheduler in synchronous automata. The intermediate formalism is called *HPIOM*. The main drawback of this formalism is that it breaks down relevant information into lower-level ones, making the task harder for verification tools, that are unable to handle real case studies. A similar work [7] describes how to generate UPPAAL models from SystemC programs. Several other translation-based approaches have been proposed [19, 11], also introducing a lot of complexity in the encoding.

Recent works attempted to tackle this problem by using asynchronous formalisms. We will show in section 4.4 that SystemC's semantics is encoded naturally and efficiently with deadline variables (similar to "clocks") evolving asynchronously, unlike the semantics of timed automata used in UPPAAL, in which clocks evolves synchronously.

In [14], a SystemC process is encoded with a *MicMac* automaton which distinguishes *micro-states* and *macro-states*. *Micro-states* represent points where the process can not yield, contrarily to *macro-states* that are yielding points (typically following a `wait()`). MicMac automata can be composed in parallel using dedicated product exploiting the notion of micro-states. This approach cannot be used directly in existing verification tools that are not aware of micro-states. [22] proposes first to encode a SystemC programs into MicMac automata *and then* to encode MicMac automata into Promela. However, the last translation loses the specific benefits of MicMac formalism. Moreover, we show that some SystemC notions are encoded naturally in Promela (in particular, atomic sections of SystemC correspond to directly to the `atomic` statement in Promela), while using MicMac as an intermediate formalism prevents such direct translation and introduces unnecessary complexity in the encoding. To sum up, the approach implies the re-encoding in an explicit and asynchronous way of some mechanisms that verification tools, including SPIN, can tackle very efficiently *when the corresponding native mechanisms are used*.

Our approach: asynchronous formalism + shared variables

This paper proposes a solution based on an asynchronous model (namely Promela) to encode TLM concurrent programs, that consists in modeling the asynchronous communications and the semantics of the scheduler by inserting synchronization primitives manipulating shared variables into the code of the processes. The expected gain of this approach is to minimize the interactions between processes, so as to let verification tools freely apply reduction techniques such as symmetry or partial order reductions.

Alternatives to Translation-Based Approaches

Other verification approaches do not need to translate the code to verify, and can apply verification based on execution. The obvious one is testing, but more elaborated techniques like runtime-verification [6] and explicit model-checking [5] can perform more exhaustive exploration of the state-space. These methods showed to be very efficient to explore the possible schedulings of a system, but are fundamentally limited to explicit-state exploration, and cannot be extended to perform symbolic model-checking or abstract interpretation. A hybrid approach is presented in [3], which executes C++ code natively for `SC_METHOD`s, but relies on translation for `SC_THREAD`s. This work is probably the closest to the one presented in this paper, as the encoding does not rely on a separate process for the scheduler. The translation scheme proposed is done manually, while we propose an automatic tool chain.

4 Translation from C++ and encoding of SystemC scheduler

We first remind the general principles of our tool chain for SystemC, then we describe precisely the encoding of SystemC synchronization primitives, and last we discuss some alternatives. Among the primitives mentioned in section 2, we will not consider delayed notifications, or waiting for conjunctions or disjunctions of events, but discuss in section 4.4 how to extend our encoding to handle such constructs.

4.1 Translating from C++

Translating SystemC automatically requires the use of a complete SystemC front-end. Borrowing some ideas from Pinapa [16], we set up a SystemC front-end called PinaVM [15] able to take as input a SystemC program and produce an intermediate representation. This front-end is based on the compiler infrastructure LLVM [13] and the intermediate representation is mainly composed of basic blocks containing SSA (*Static Single Assignment*) instructions. PinaVM executes the elaboration phase like Pinapa, and uses a *Just-In-Time* compiler to retrieve SystemC information on events or ports to enrich intermediate representation obtained from LLVM.

From the intermediate representation produced by our front-end, a back-end produces automatically a Promela program. Each SSA instruction is translated into an equivalent in Promela instruction. Although Promela provides some of the structuring mechanisms of a call definition, these mechanism provide no benefit for the verification engine compared to a static inlining, therefore, we chose to inline directly all function calls, which is made easy by `llvm::InlineFunction()`.

In the encoding of SystemC synchronization primitives, we rely on three features related to concurrency that are provided by Promela:

1. The ability to use shared variables.
2. The blocked($cond$) primitive, which stops the execution of the current process until condition $cond$ on shared variables becomes true, and gives the control to another process (the actual syntax in Promela is simply `[`*cond*`]`).
3. The notion of atomic section, that can be interrupted with the blocked primitive.

In the translation, each SystemC thread generates a Promela process, we do not consider in this paper dynamic creation of processes, that are seldom encountered in SoC models.

4.2 Encoding synchronization primitives

In the sequel we denote by E^k the event k, with $1 \leq k \leq N_e$ and the set of N_p processes is denoted P.

p**::wait(**E^k**):**	p**::**E^k**.notify():**
1 $W_p := k$	3 $\forall i \in P \mid W_i == K$
2 blocked($W_p == 0$)	4 $\quad W_i := 0$

Table 1: Encoding events alone

```
p::wait(d):
1  $T_p := T_p + d$
2  blocked($T_p == \min_{i \in P}(T_i)$)
```

Table 2: Encoding time alone

Events

Processes waiting for an event are eligible immediately *after* the event is notified. This means than SystemC events are *non persistent*: an event E^k notified before the execution of a **wait(**E^k**)** instruction will be ignored by this instruction, that will block until the next notification of E^k. An important consequence is that a process can be waiting for at most one event (we currently do not consider the construct `wait(e1 & e2)` of SystemC): the instruction wait(E^k) is blocking, and takes into account only notifications taking place after its execution.

For encoding events, we thus associate to each process p a bounded integer $0 \leq W_p \leq N_e$ such that:

- $W_p == k$ when process p waits for E^k;
- $W_p == 0$ when process p is not waiting for an event and is eligible;

and we define the wait and notify instructions by Tab. 1. We need for this encoding $N_p \log_2(1 + N_e)$ bits.

Time

SystemC time management internally assumes a discrete time semantics, although in the API timed functions use floating-point durations. We thus assume that we have a specific construct `wait(d:int)` to wait for the *discrete* duration d to elapse.

For encoding time, we attach an internal *deadline variable* $T_p : int$ to each process p. It represents the next deadline for p when p is waiting, and the current date when p is running. It is not necessary to examine the state of the process p for each value of T_p, we only need to respect the schedulings allowed by the durations waited for by the processes. Consequently, we define the encoding `wait(d)` by Tab. 2:

- T_p is incremented with d;
- p becomes eligible if its deadline variable is the minimum of all deadline variables.

Notice that we could also maintain a global clock T_g to $\min_{i \in P}(T_i)$ and replace the blocking condition by **blocked(**$T_p == T_g$**)**. The advantages and drawbacks of this option w.r.t. the efficiency of the verification process is hard to assess *a priori*.

Interaction between time and events

Events and time interact together, and things become subtle when some processes are waiting for events and others for a time duration. We propose the encoding is given on table 3, based on the following principles:

(1) The value of a deadline variable T_p is meaningful *only if* $W \neq 0$ (process p is not waiting for an event). When a process is waiting for an event, T_p is not updated. The main invariant becomes thus: "*the deadline variable of a running or eligible process is the minimum of the deadline variables of processes not waiting for an event.*"

(2) Concerning the **wait(d)** instruction, the blocked process becomes eligible as soon as its deadline variable is the minimum of deadline variables *of processes not waiting for an event*, according to principle 1).

(3) When process p notifies an event E^k, not only should the variables W_i be reset (for processes i waiting for E^k), but also should their deadline variable be updated to the current date (which is equal to the deadline variable T_p of the running process p). This is because of principle (1): these deadline variables

```
$p$::wait(d):
1 $T_p := T_p + d$
2 blocked($T_p == \min_{\substack{ı\in P\\ W_i==0}} (T_i)$)

$p$::wait($E^k$):
3 $W_i$ := K
4 blocked($W_i$ == 0)

$p$::$E^k$.notify():
5 $\forall i \in P \mid W_i == k$
6     $W_i := 0$
7     $T_i := T_p$
```

Table 3: Encoding events and time

```
$p$::wait($E^k$)
$e_p^k$ := true
blocked($e_p^k$ == $false$);

$p$::$E^k$.notify()
$\forall i \in P$
    $e_i^k$ := $false$;
```

Table 4: Encoding of events with 1 Boolean per process per event

becomes meaningful again, and the invariant above should be maintained. This is important to make a sequence **wait(E^k); wait(d)** behave correctly in a process p.

The aim of section 5 is to get confidence on the correctness of this rather subtle encoding.

Alternate Encoding of Events with Booleans

In a first attempt, we encoded each event with one Boolean e^k per event. However, it is not sufficient, as explained in appendix A.

We consider one Boolean per event *and* per process, and we denote by e_p^k the Boolean associated to event k for process p. Intuitively, when e_p^k is true, process p is waiting for event k. The correct encoding for process p is given by table 4.

In this case, we need at most $N_p \cdot N_e$ shared Booleans, with N_p the number of processes, and N_e the number of events in the system. In fact, theoretically, we only need, for each event, one Boolean for each process that can wait for this particular event. This optimization is not taken into account in encoding given here in the examples but has been implemented in our verification chain.

4.3 Back on time: coding relative deadlines

Our encoding of time uses unbounded deadline variables, which make unsuitable for finite-state model-checkers like SPIN [8]) (although timed automata model-checkers (e.g., UPPAAL [12]) or static analyzers (e.g. [10]) can handle them). Fig. 3 shows an example where the state space is infinite. This system is periodic, as variable **shared** is consecutively valued: 0, 1, 2, 0 etc. (if the incrementing process is scheduled first). Proving that **shared** never takes the value 3 should be easy but the reachable state space is infinite.

However, two global states agreeing on the differences $T_i - T_j$ between deadline variables and differing only on some absolute value T_i are equivalent w.r.t. the semantics of the synchronization primitives of Tab. 3. But of course a finite-state model-checker will not detect this and will not quotient automatically the infinite state-space according to this relation in order to obtain a finite state-space.

The solution is to perform this quotient in the encoding, by considering relative time instead of absolute time. We need to shift — at some points — all meaningful deadline variables so as to make the main invariant stated in section 4.2 becomes: *"the deadline variable of a running or eligible process is the minimum, which is zero, of the deadline variables of processes not waiting for an event."* We propose to shift them in **wait(d)** instructions, which results in the definition of Tab. 5. This ensures the following invariant that guarantees that this encoding makes deadline variables bounded.

```
int shared := 0;

Thread increment:
  while (true) {
    shared := shared +1;
    wait(5);
  }

Thread decrement:
  while (true) {
    shared := shared - 2;
    wait(10);
  }
```

Figure 3: Example with an infinite reachable state-space

```
$p$::wait(d)
  1 $T_p := T_p + d$
  2 $\forall i \in P \mid W_i == 0$
  3         $T_i = T_i - \min\limits_{W_j==0}(T_j)$
  4 blocked($T_p == 0$)

$p$::wait($E^k$)
5 $W_i$ := K
6 blocked($W_i == 0$)

$p$::$E^k$.notify()
7 $\forall i \in P|W_i == k$
8       $W_i := 0$
9       $T_i := T_p$
```

Table 5: Encoding of relative time (with integers for events)

Proposition 1 *If D denotes the maximal time duration D appearing in a* ***wait(d:int)*** *instruction, and if the main invariant $\min\limits_{W_i==0}(T_i) == 0$ is satisfied, then $\forall p \in P \ : \ 0 \leq T_p \leq D$.*

Indeed:

- Deadline variables are initially zero and thus the initial state satisfies the invariant.
- If the invariant is satisfied, it is maintained by the encoding of p ::E^k**.notify()** instruction (line 9).
- If the invariant is satisfied, and the instruction p ::**wait(d)** starts to execute, this first implies that $W_p = 0$ and $T_p = 0$. After line 1, we obtain $0 \leq T_p = d \leq D$, and after line 3, we have $0 \leq T_p \leq d' \leq d \leq D$ for some D'.

4.4 Discussion

Our encoding implements in some way an asynchronous time semantics, as opposed as the synchronous time semantics of timed automata used in tools like IF [1] or UPPAAL [12], in which clocks evolves synchronously. Our approach thus does not enable the use of these tools. On the other hand, we hardcode in our approach the fact that we only need to know the next deadlines, and not all the possible intermediate values that a synchronous clock would take between the current time and the next deadline.

In particular, it is known that the use of *discrete synchronous* clocks is a bad idea with finite-state model-checkers, as they enumerate all the possible successive values of such clocks. But in our case, the analogous of clocks is our deadline variables, the value of which jumps directly from the current value to the next deadline to meet, thus avoiding an unnecessary enumeration of the intermediate values. As a result, multiplying all the durations by a constant factor does not impact the size of the reachable state-space with our encoding.

Implementing delayed notification on a single event could be done with the principles we followed in this section. This would require to add another deadline variable in each process.

Implementing waiting for conjunction or disjunction of events would require the following modifications:

- The bounded integer variables $0 \leq W_p \leq N_e$ should be replaced by N_e Boolean variables $W_{p,k}$ with $1 \leq k \leq N_e$ denoting the event E^k, because a process p can know wait for a set of events.
- We should also add a Boolean variable per process to distinguish whether the process is waiting for a conjunction or a disjunction of events.

To sum up, our approach can easily model such constructs, at the cost of additional finite-state variables.

5 Testing the encoding principles

Although the encoding for SystemC primitives defined above are intuitively correct, we want to verify it.

The ideal solution would be to prove that our encoding is correct in any context, that is, in any program using it. Such a quantification on programs requires the use a proof-assistantlike Coq [9], which is a very demanding task. This would require to give a formal semantics to SystemC, which means to C++, as well as to Promela, and to prove that two programs are equivalent. Since the formalization of SystemC itself cannot be formally proved, even such approach do not actually prove formally the complete translation scheme.

The approach we have chosen is to construct a set of properties and to verify them on a set of examples, in order to check that the encoded models has the same semantics as the original examples and to get confidence in the correctness of the encoding. We first present these properties and then we describe how we checked them using SPIN. Generally speaking, it is quite hard to find bugs in concurrent programs; those verifications were very useful, allowing us to detect bugs in several preliminary versions of our encoding.

5.1 Invariants with absolute time

1. **"The deadline variable of a running or eligible process is always the minimum of all meaningful deadline variables"** This is the main invariant stated in section 4.2.
2. **"If process i notifies event E^k for which process j is waiting, then $T_i \geq T_j$"**

   In other words, a process notifying the event E^k executes after the processes waiting for it.
3. **"When a process p waiting for an event is made eligible by a notifying process (line (7) of Fig. 3), the deadline T_p of the process p does not change until its election as the running process."**
4. **"before and after yielding, i_waiting == $\bigvee_{j \in P} e_j^k$"** (specific to the Boolean encoding)

5.2 Invariants with relative time

The 3 invariants above reduces to the 3 following invariants when considering the encoding based on relative time in section 4.3.

1. **"The deadline variable of a running or eligible process is always 0"** This is the main invariant stated in section 4.3, which implies proposition 1, which implies in turn that "the deadline variable of a running or eligible process is always the least of all deadline variables".
2. Because of invariant 1, this reduces to a trivial invariant: **"If process i notifies event E^k for which process j is waiting, then $T_i = T_j = 0$"** This is because process i is the running process, and process j was running when executing **wait(E^k)**.
3. This invariant is formulated the same way as in section 5.1.

Property	Assertions	Line number (for assertions)
1	Assertions before and after each yielding point	**Booleans:** before lines 2 and 4, tab. 4
		Integers: before lines 2 and 5, tab. 3
2	Assertion in `notify(event)`	**Booleans:** Before line 7, tab. 4
		Integers: Before line 6, tab. 3
6	Additional variable + assertions in wait(e)	**Booleans:** Variable after line 8, assertion after line 5, tab. 4
		Integers: Variable after line 7, assertion after line 4, tab. 3
4	**Booleans:** Assertions before and after yielding points	Before and after lines 2 and 4, tab. 4

Table 6: Using SPIN to verify properties

5.3 Verifying invariants

Verifying the invariant directly on our implementation would require theorem-proving on C++ code using complex libraries, and is not realistic, although desirable. Instead, we verify the invariants on instances of the translation, just like certifying compilers [18] verify the result of each compilation.

We verified all invariants on various examples, which allows increasing the confidence in the correctness of the encoding.

Table 6 sums up how each invariant was verified using SPIN, for the encoding with absolute time.The verification of the invariants for relative time are a mere derivation of these ones. Mainly, two techniques were used: direct assertions in the code; and a “monitoring” process, for properties not related to a specific line number. This process only contains assertions, which can be detected as violated in the automata product performed by SPIN.

Invariant 3 was checked using additional variables and assertions. Intuitively, with P_1 the process waiting for an event e, P_2 the initial running process which notifies P_1, this is why the property cannot become false:

- With our encoding, P_1 is immediately eligible and cannot be notified anymore before being elected.
- For a third process P_3 to change T_1 before P_1 be elected, P_3 must be elected after P_2. This implies $T_3 \leq T_1$. As $T_1 == Min(T_i)$, $T_3 \nless T_1$ and consequently, $T_3 == T_1$, even if P_1 could be notified before being elected, a notification by P_3 would set T_1 to the same deadline variable.

As the examples we considered are deadlock-free (use of well-known algorithms), we also verified that the encoding does not introduce deadlocks (for instance, by scheduling processes in the wrong order).

The examples on which we checked these properties are the following. First, we experimented on an adaptation of the reader/writer problem in which two writers and one reader access a FIFO. Second, we considered a model of a communication between a Memory, a DMA, a bus and a CPU. Third, we considered the example used in a previous translation from SystemC to SPIN [22], described in appendix A.1.

6 Experiments and efficiency of our encoding

The aim of the previous section was to formally check that our encoding effectively reflects SystemC semantics. However, our motivation for the encoding we propose is to enable better performances of model-checkers, compared to other encoding approaches described in section 3. We now show how our case study was translated to Promela in order to apply the SPIN model-checker, and then we compare experimentally the efficiency of our encoding w.r.t. model-checking with the encoding proposed in [22] applied to the same example.

6.1 A SystemC example

Fig. 4 shows our encoding translated to Promela. Our verification chain actually unrolls loops and generates specialized functions for the different events and processes.

Using SPIN, the use of a T_g representing the minimum of all T_i (see section 4.2) could be accurate as since this would allow to use T_g to verify properties. Although this does not change the complexity of the

```
SC_MODULE(MyModule)
{
MyModule *initiator;
sc_event e;

SC_HAS_PROCESS(MyModule);

MyModule(sc_module_name name) {
SC_THREAD(compute);
sensitive << e;
}

void fonct() {
e.notify();
}

void compute() {
wait(e);
initiator->fonct();
}
};
```

# modules	3		5		7		9		11	
	time	states	time	states	time	states	time	states	time	states
no bug	0.00	39	0.00	121	0.00	419	0.01	1581	0.08	6199
bug	0.00	32	0.00	74	0.00	188	0.00	590	0.03	2144

# modules	13		15		17		19		21	
	time	states	time	states	time	states	time	states	time	states
no bug	0.46	24641	2.46	98379	12.26	393301	65.56	1572959	326.08	6291561
bug	0.19	8306	1.04	32900	5.49	131222	28.22	524456	145.36	2097338

Table 7: Experimental results

verifications performed by SPIN, we did not use a global T_g in order to simplify the code.

Fig. 7 in the appendix shows the pseudo-code for the reader/writer example described above.

Our test model is the one used in [22] and partly detailed in Fig. 6.1. It consists of a chain of modules. The first module triggers an interrupt in the next one. This interrupt notifies an event, allowing the module to trigger an interrupt in the next module, and so on. The last module contains an assertion which is either always false (bug) or always true (no-bug). The latter forces SPIN to compute the whole state space when checking for invalid assertions. While this program may seem artificial, it exhibits the characteristics found in more complex real-world models and leading to state explosion: many processes, synchronized by SystemC events, which can thus be lost depending on the execution order of the various statements. Such study allows to experiment on how the state space that needs to be explored grows depending on parameters. As this test model is untimed, we test here only the efficiency of the encoding of events.

6.2 Results

The results presented in Fig. 5 focuses on the main parameter which is the number of modules. It shows the number of states computed by SPIN during the model-checking of the example presented above.

Table 7 gives a bit more details. The line “no-bug” corresponds to the example described, whereas the line “bug” shows the number of states computed before finding a counter-example in the case where an `assert(false)` has been introduced in the last module.

We mainly observe two things: First, the number of states is growing exponentially, although the acceleration is not high: there is a factor 3 between the number of states computed for 3 modules and 5, a factor of about 4 between results for 15 and 17 modules and also a factor of about 4 between 19 and 21.

Second, those results show a reduction by a factor of about 10 compared to previous results presented in [22]. The comparison between the two approaches, in the case where there is no bug is shown in figure 5. We can see that, with our encoding, SPIN is able to model check up to 21 processes, compared to 15 in the other approach. In addition, one of our main results is that the encoding presented here have been fully automated.

```
int e[NBTHREADS];
int T[NBTHREADS];
bool end[NBTHREADS];

inline init_coding(i) {
  i = 0;
  do :: i == NBTHREADS -> break;
     :: else ->
        e[i] = 0; T[i] = 0; end[i] = false;
        i++; od;
}

inline notify(pid, nevent, i) {
  i = 0;
  do :: i < NBTHREADS && e[i] == nevent ->
        e[i]=0;  T[i]=T[pid];  i++;
     :: i < NBTHREADS && e[i] != nevent ->
        i++;
     :: i == NBTHREADS -> break; od;
  i = 0;
}

inline wait(pid, time) {
  T[pid] = T[pid] + time;
  ((end[0]) || (e[0] != 0) || (T[pid] <= T[0]) &&
   (end[1]) || (e[1] != 0) || (T[pid] <= T[1]) &&
   (end[2]) || (e[2] != 0) || (T[pid] <= T[2]));
}

inline wait_e(pid, nevent) {
  e[pid] = nevent;
  e[pid] == 0;
}
```

Figure 4: Encodings in Promela. Compared to Tab. 3, we add the `end` array to handle the particular case where a task is completed in the `wait(d:int)` instruction.

7 Conclusion

We investigated the formalization of models of SoC in the form of asynchronous automata. We proposed an encoding of synchronization primitives related to events and time using shared variables and sequential instrumentation of processes. This choice contrasts with other approaches in which parallel instrumentation is used, under the form of an additional process modeling the SystemC scheduler added to the system. We ensured the encoding principles are correct by verifying a number of invariants. The given principles are general and apply to different back-end languages.

We experimented on the SPIN model-checker, showing that our encoding leads SPIN to explore ten times less states during model-checking of the encoded model, compared to an encoding based on parallel instrumentation. This confirms the conjecture we express in section 3. In addition, the translation has been fully automated: our tool reads SystemC code directly, and generates Promela code without human intervention. This shows that our results are due to our encoding and not to specific optimizations. The tool can be downloaded freely from `http://gitorious.org/pinavm`.

We see at least two point to investigate in the future. First we have yet to compare our time management to other approaches. We intend to compare this solution to approaches based on timed automata and relying on the UPPAAL [7] tool for model-checking to validate our discussion of section 4.4 on the asynchronous encoding of time in SystemC. A second perspective to evaluate the relevance and the efficiency of static analysis tools such as CONCURINTERPROC [10] for checking safety properties of timed SystemC models.

References

[1] M. Bozga, S. Graf, and L. Mounier. IF-2.0: A validation environment for component-based real-time systems. In K.G. Larsen Ed Brinksma, editor, *Proceedings of CAV'02 (Copenhagen, Denmark)*, volume 2404 of *LNCS*, pages 343–348. Springer-Verlag, July 2002. 4.4

[2] Giovanni Funchal and Matthieu Moy. jTLM: an experimentation framework for the simulation of transaction-level models of systems-on-chip. In *DATE*, 2011. (to appear). 2

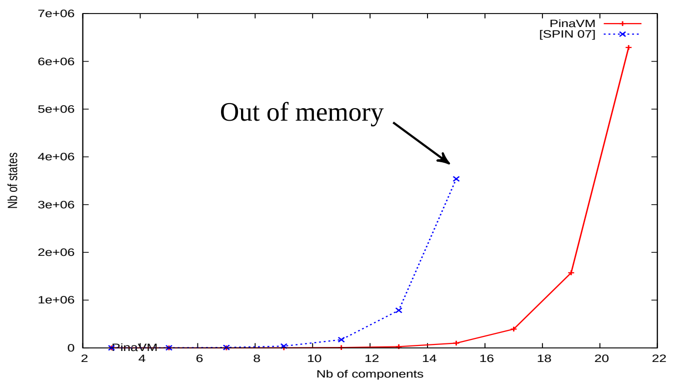


Figure 5: Experimental results of the two approaches

[3] Hubert Garavel, Claude Helmstetter, Olivier Ponsini, and Wendelin Serwe. Verification of an Industrial SystemC/TLM Model using LOTOS and CADP. In *7th ACM-IEEE International Conference on Formal Methods and Models for Codesign MEMOCODE'2009*, Cambridge, MA United States, 2009. 3, 3

[4] Frank Ghenassia. *Transaction-Level Modeling with SystemC: TLM Concepts and Applications for Embedded Systems*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006. 1

[5] Claude Helmstetter. TLM.open: a SystemC/TLM Front-end for the CADP Verification Toolbox. Extended abstract for SBDCES workshop (http://unit.aist.go.jp/cvs/workshop/SBDCES.html) Work financed by the Multival project. 3

[6] Claude Helmstetter, Florence Maraninchi, and Laurent Maillet Contoz. Full simulation coverage for SystemC transaction-level models of systems-on-a-chip. *Formal Methods in System Design*, 35(Number 2 / October, 2009):pages 152–189, 06 2009. 3

[7] Paula Herber, Joachim Fellmuth, and Sabine Glesner. Model checking SystemC designs using timed automata. In *CODES/ISSS '08: Proceedings of the 6th IEEE/ACM/IFIP international conference on Hardware/Software codesign and system synthesis*, pages 131–136, New York, NY, USA, 2008. 3, 7

[8] Gerard J. Holzmann. *Design and validation of computer protocols*. Prentice-Hall, Englewood Cliffs, NJ, 1991. 4.3

[9] INRIA. The coq proof assistant. http://coq.inria.fr/. 5

[10] B. Jeannet. Relational interprocedural verification of concurrent programs. In *Software Engineering and Formal Methods, SEFM'09*. IEEE, November 2009. to appear. 4.3, 7

[11] D. Karlsson, P. Eles, and Z. Peng. Formal verification of systemc designs using a petri-net based representation. In *Proceedings of the conference on Design, automation and test in Europe: Proceedings*, page 1233. European Design and Automation Association, 2006. 3

[12] K. G. Larsen, P. Pettersson, and W. Yi. UPPAAL in a Nutshell. *Int. Journal on Software Tools for Technology Transfer*, 1(1–2):134–152, October 1997. 4.3, 4.4

[13] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *CGO '04: Proceedings of the international symposium on Code generation and optimization*, page 75, Washington, DC, USA, 2004. IEEE Computer Society. 4.1

[14] F. Maraninchi, M. Moy, J. Cornet, L. Maillet-Contoz, C. Helmstetter, and C. Traulsen. SystemC/TLM semantics for heterogeneous system-on-chip validation. In *NEWCAS-TAISA 2008: Proceedings of the Joint 6th International IEEE Northeast Workshop on Circuits and Systems and TAISA Conference*, pages 281–284, 2008. 3, 3

p::**wait(E^k):** E^k := true; blocked($e^k == false$); e^k := true;	p::E^k**.notify():** e^k := false;

Table 8: **Incorrect** encoding with with 1 Boolean per event

[15] Kevin Marquet and Matthieu Moy. PinaVM: a SystemC front-end based on an executable intermediate representation. In *International Conference on Embedded Software International Conference on Embedded Software*, page 79, Scottsdale, USA, 10 2010. SD B.4.4, I.6.4, D.2.4 OpenTLM (projet Minalogic). 4.1

[16] Matthieu Moy, Florence Maraninchi, and Laurent Maillet-Contoz. Pinapa: An extraction tool for SystemC descriptions of systems-on-a-chip. In *EMSOFT*, September 2005. 4.1

[17] Matthieu Moy, Florence Maraninchi, and Laurent Maillet-Contoz. LusSy: an open tool for the analysis of systems-on-a-chip at the transaction level. *Design Automation for Embedded Systems*, 2006. special issue on SystemC-based systems. 3, 3, 3

[18] G.C. Necula and P. Lee. The design and implementation of a certifying compiler. *ACM SIGPLAN Notices*, 33(5):333–344, 1998. 5.3

[19] B. Niemann, C. Haubelt, M. Oyanguren, and J. Teich. Formalizing TLM with communicating state machines. *Advances in Design and Specification Languages for Embedded Systems*, pages 225–242, 2007. 3, 3

[20] Open SystemC Initiative. *IEEE 1666 Standard: SystemC Language Reference Manual*, 2005. `http://www.systemc.org/`. 1, 2, 1

[21] Open SystemC Initiative (OSCI). *OSCI TLM-2.0 Language Reference Manual*, July 2009. Version JA32, available from `http://www.systemc.org/downloads/standards`. 2

[22] Claus Traulsen, Jérôme Cornet, Matthieu Moy, and Florence Maraninchi. A SystemC/TLM semantics in Promela and its possible applications. In *14th Workshop on Model Checking Software SPIN*, July 2007. 3, 3, 5.3, 6, 6.1, 6.2

A Examples of incorrect encodings

It is not sufficient to use only one Boolean per event. For instance, let us consider the encoding described in table 8, where several processes are waiting for event E^k:

In this case, if several processes are waiting on E^k, only one of them, at most, is unblocked. Indeed, since the process chosen as running immediately sets e^k to false, the others remain blocked on the `blocked(`$e^k == true$`);`

A.1 Wrong Encoding of Time and Events with Booleans

This encoding could seem to be correct but is in fact invalid. Figure 6 presents an example where this encoding fails to execute processes in the correct order. Three processes are concurrently executing. The instructions executed by processes are represented in circles (1A, 1B...) and are separated by SystemC constructs: waiting or notifying events (`wait(e)`, `notify(e)`, waiting time (`wait(20)`). The right hand side of the Fig. details the concurrent execution of the three processes. The process with least deadline variable is always chosen, but the encoding leads to the following error:

1. T1 performs a wait(90) ;
2. T2 should be eligible at this point but it is not because 2_waiting is still true ;

```
p::wait(d):
1  $T_p := T_p + d$
2  blocked($T_p == \min_{i \in P,\ i\_waiting == false} (T_i)$)

p::wait(E^k):
3  $e_p^k$ := true
4  $p$_waiting := true
5  blocked($e_p^k == false$)
6  $p$_waiting := false

p::E^k.notify():
7  $\forall i \in P | e_i^k == true$
8      $e_i^k := false$
9      $T_i := T_p$
```

Table 9: **Incorrect** complete encoding with Booleans

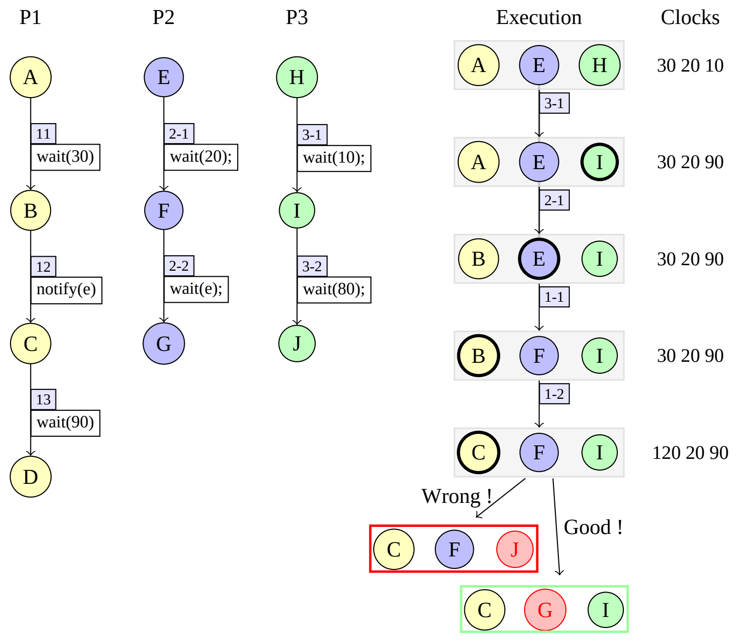


Figure 6: Counter-example for invalid encoding

3. so T3 is chosen as running.

The error here is that when a process i waiting for an event is notified, it becomes eligible while i_waiting is not set to false. Therefore, it can be chosen as running even if its deadline variable is greater than the deadline variable of a process waiting for time.

A.2 The reader-writer example

```
int empty_fifo;
#define MAX 3

#define LOOP_NB_W 10
#define LOOP_NB_R 20


procedure
write_in_fifo(pnumber, time)
{
  empty_fifo--;
  notify(pnumber, EVENT_WRITE);
  wait(pnumber, time);
}


Thread writer0:
  int j, temp;
  int  pnumber = 0;

  for (j = 0; j < NB_LOOP_W; j++) {
    while (empty_fifo == 0)
      wait_e(pnumber, EVENT_READ);
    i = 0;
    write_in_fifo(pnumber, TIME0);
  }
```

```
#define TIME0 5
#define TIME1 3
#define TIME2 1

#define NBTHREADS 3


procedure
read_in_fifo(pnumber, time)
{
  empty_fifo++;
  notify(pnumber, EVENT_READ);
  wait(pnumber, time);
}


Thread writer1:
  int j, temp;
  int  pnumber = 1;

  for (j = 0; j < NB_LOOP_W; j++) {
     while (empty_fifo == 0)
       wait_e(pnumber, EVENT_READ);
     i = 0;
     write_in_fifo(pnumber, TIME1);
  }
```

```
Thread reader:
  int i, j, temp;
  int  pnumber = 2;

  while (j = 0; j < NB_LOOP_R; j++) {
    while (empty_fifo == MAX)
      wait_e(pnumber, EVENT_WRITE);
    i = 0;
    read_in_fifo(pnumber, TIME2);
  }
```

Figure 7: Reader/writer example