A SystemC/TLM semantics in PROMELA and its possible applications

Claus Traulsen[2,1], Jérôme Cornet[1], Matthieu Moy[1], and Florence Maraninchi[1]

[1] Verimag, Centre Équation - 2, avenue de Vignate, 38610 GIÈRES — France
[2] Dept. of Computer Science, Christian-Albrechts-Universität zu Kiel, Olshausenstr. 40, 24098 KIEL — Germany

Abstract. SystemC has become a *de facto* standard for the modeling of systems-on-a-chip, at various levels of abstraction, including the so-called *transaction level* (TL). Verifying properties of a TL model requires that SystemC be translated into some formally defined language for which there exist verification back-ends. Since SystemC has no formal semantics, this includes a careful encoding of the SystemC scheduler, which has both synchronous and asynchronous features, and a notion of time. In a previous work, we presented a complete chain from SystemC to a synchronous formalism and its associated verification tools. In this paper, we describe the encoding of the SystemC scheduler into an *asynchronous* formalism, namely PROMELA (the input language for SPIN). We comment on the possible uses for this new encoding.

1 Introduction

SystemC [17] is a C++ library/language used for the description of Systems-on-Chip (SoCs) at different levels of abstraction, from cycle-accurate to purely functional models. It comes with a simulation environment, and is becoming a *de facto* standard in the SoCs industry. SystemC is being increasingly used for writing *Transaction Level Models* (TLM) [7] that allow embedded software development on a virtual prototype of the final chip.

A TL model written in SystemC is based on an *architecture*, *i.e.*, a set of components and connections between them. Components behave *concurrently*. Each component has typed connection *ports*. Its behavior is given by a set of communicating *processes* programmed in full C++ and managed by a non-preemptive *scheduler*. Synchronization mechanisms include *events*, which can be waited for or notified. A process yields control back to the scheduler either by waiting for an event or by waiting for a given period of time to elapse.

Communications between modules proceed by function calls traversing components or communication *channels* (for instance bus models). At the transaction level, such function calls are used to model two types of communication: transactions (atomic exchange of data between modules) and interrupts.

Since the TL models are considered as *reference* models in the SoC design flow, it is necessary to validate properties at this level of abstraction. This is currently done by intensive testing, but formal verification is being investigated

for some years now, in both industry and academia. However, the definition of SystemC, while being an IEEE norm, lacks formal semantics.

Some work on verifying properties of SoC assume that they are described in some parallel formalism inspired by SystemC. Often, this formalism only reflects the subset of the language used for Register Transfer Level (RTL) models and is useless to express the specificities of a TL model (see for instance [9], [5] ; more references in Section 5). Formal verification for RTL designs has been studied a lot, and providing such analysis for designs written in SystemC does not bring new results. Moreover, even if the formalism reflects in some way the transaction level of abstraction, it is often quite far from real SystemC designs.

We are interested in verifying properties of real SystemC designs, at the transaction level. This requires that SystemC and TLM-specific features be translated into some formally defined language for which there exists verification back-ends. This includes a careful encoding of the SystemC scheduler, which has both synchronous and asynchronous features, and a notion of time.

Choosing the formal language in which to translate SystemC is important because it often restricts the set of verification back-ends that can be applied. If we translate SystemC into a symbolic synchronous formalism, we have access to tools like the Lustre [8], SMV [13] or Esterel [2] tool-chains; if we translate it into an asynchronous formalism, we have access to Spin [10], IF [4], etc. Since the semantics of SystemC processes is neither entirely synchronous, nor entirely asynchronous, any choice implies some encoding. The encoding itself may be responsible for a significant part of the model size.

Another important point is the way time is interpreted. Since SystemC contains explicit constructs to wait for time, the translation into the input language of a timed-model checker like IF [4] or UPPAAL [12] could seem to be an appropriate choice. However, we do not need the full expressive power of timed automata to encode SystemC, and using timed automata would imply to pay the price of the symbolic analysis needed for clocks in the verification back-ends. Consequently, we will choose a discrete interpretation of time in SystemC, and encode timers into some ordinary variables.

In a previous work [16, 14], we described a complete chain from SystemC to a synchronous formalism. It is based upon a systematic encoding of SystemC processes into a flavor of synchronous automata. In such a case, SystemC processes are encoded one by one into automata, communicating with an additional automaton that encodes the scheduler specification. The automata corresponding to the user processes are produced specifically to communicate with this scheduler automaton, using additional synchronous signals and the instantaneous dialogue mechanism available in a pure synchronous semantics. The set of automata can then be translated into several synchronous formalisms (SMV [13] input, Lustre), without computing the intrinsic *products* between them, hence delegating the potential state explosion to the verification back-ends that are better equipped to tackle the problem. Another good property of the translation into Lustre is that we get something *executable*. It means that we can, at least,

compare the Lustre encoding with the official SystemC semantics, on benchmark programs.

However, the encoding into a synchronous formalism renders manual reading difficult, and the amount of additional synchronizations needed to reflect the semantics of SystemC can be put in question again.

Another approach would be to define a direct semantics by using an *ad hoc* product to represent the effect of the scheduler, that is creating a dedicated parallel composition which would include the main characteristics of the SystemC scheduling specification. While this approach produces more readable results, the fact is that it also requires to create a dedicated model-checker and prevents from using existing verification tools, that deal only with well-known formalisms.

In this paper, we explore the encoding of the SystemC scheduler into an *asynchronous* formalism, namely PROMELA, the input language for the tool SPIN [10]. The translation into PROMELA is another way to give a formal semantics to SystemC/TLM. Thanks to the simulator provided with SPIN, the semantics is executable, and it will be possible to test the faithfulness of our encoding w.r.t. the official scheduler.

So far, we experimented our translations for model-checking and intensive testing of properties like deadlocks and assertions.

The alternative encoding of SystemC into an asynchronous formalism will also allow the comparison of the two verification chains, with respect to the size of the models, the power of the verification tools, etc. In other words, we try to answer the following questions: for a given SystemC model, and a given property of it, is it more efficient to use a synchronous verification chain, or an asynchronous one? This may depend on the type of property, and is still being investigated.

The rest of the paper is structured as follows: Section 2 gives an short introduction to TL modeling with SystemC. Section 3 describes the translation and Section 4 how to use it for verification. We consider related work in Section 5 and conclude in Section 6.

2 Transaction Level Modeling with SystemC

2.1 Subset of SystemC

We briefly describe the main characteristics of SystemC, when used for Transaction Level Models.

Globally, a TLM *component*, or *module*, is an encapsulated piece of code that contains active code (processes to be scheduled by the global scheduler) and passive code (functions offered to the external world, that will be called from a process of another component, by a control flow transfer). Inside such a component, the processes and the functions may share variables and events in order to synchronize with each other. Note that a function code and an active process are conceptually *in parallel*, since the function will be called by another flow of control.

In SystemC, the architecture of the platform is built by a piece of code (the so-called elaboration phase that runs first), but this is conceptually equivalent to a quite traditional architecture-description-language (ADL) (see for instance [6]) that connects the ports exposed by the components.

Communications between modules proceed by function calls traversing components or communications *channels* (for instance bus models). SystemC provides built-in primitive communication channels such as `sc_signal` to model hardware signals at the Register Transfer Level of abstraction. Synchronization associated with the communications is performed by events and shared variables inside modules and/or channels.

In the sequel, we only consider SystemC models at Transaction Level. By such restriction we mean that neither the built-in primitive channels nor the so-called *request update* mechanism (intended for RTL modeling) are used. We will also assume that there is no dynamic creation of processes (see section 3.1 for justification). Otherwise, we fully support other SystemC features.

2.2 A Simple Example

Consider the simple example in Figure 1, consisting of two modules. The first module contains a process that waits for an event $e1$. After receiving the event, it waits an additional 7 ns, before performing some action α. Here α is the abstraction of some real, local computation.When α is finished, the first process will trigger an interrupt in the second module, by calling the function g. The first module also offers an interrupt port by the function f, which will notify the local event $e1$. Similarly, the second module contains a process that waits 5 ns before performing β. Thereafter it will trigger the interrupt in the first module, and wait for the event $e2$, *i.e.* for an interrupt that triggers the function g.

Module 1

```
// Process 1
while(true)
{
  wait(e1);
  wait(7, SC_NS);
  cout << "alpha";
  interrupt_out.g();
}

// Function f
void f()
{
  e1.notify();
}
```

Module 2

```
// Process 2
while (true)
{
  wait(5, SC_NS);
  cout << "beta";
  interrupt_out.f();
  wait(e2);
}

// Function g
void g()
{
  e2.notify();
}
```

Fig. 1. A simple example of a TL model with two modules.

While this example is trivial, it contains the relevant parts of a TL model: waiting for event notification, waiting for time and function calls to other modules, which are used to model transactions and interrupts.

There exist three difficulties when defining the semantics of SystemC:

1. The non-preemptive scheduler: we have to ensure that a running process is not interrupted by any other process unless it explicitly relinquishes the control back to the scheduler, by performing a wait either on time or on an event.
2. The SystemC scheduler ensures that time can only elapse when no process is eligible (while SystemC has no control on *real* time, the *simulation* time is merely a counter that the scheduler can decide to increment or not). So all statements are executed instantaneously, except when waiting either for time or for an event that is not notified immediately.
3. Function call communications: they are used for both interrupts and transactions. A process P performing a function call communication lets its control flow go outside its component to finally reach the destination component where a function is executed. The execution of P may continue only after the function call is finished. This means that if the receiver's function performs a `wait()` statement, P will yield the control back to the scheduler, and when elected again will resume its execution in the receiver's function.

3 Expressing SystemC semantics in PROMELA

3.1 General Ideas

The Architecture First, we will abstract from the architecture description part. In a real SystemC design, the function that is actually called when a process P in a module M executes an instruction `p.f()` is determined by the link between the port `p` of M and a port `p'` on another module M' containing a function f' associated with p'.

In the rest of the paper, we will consider that the architecture is hard coded in the function names. In other words, we will consider a process P calling a function f, and another module containing f.

Processes and Functions The modeling of a SystemC program into PROMELA transforms each process and each function into a PROMELA process. SystemC distinguishes several types of processes (`SC_THREAD`, `SC_METHOD`, etc.). We consider here the use of `SC_THREAD`, which is the most general, since the encoding will not benefit from the restrictions enforced on the other types of processes.

We decided to encode the functions as PROMELA processes in order to keep them well separated from the processes. Thus, the transformation stays modular and is easier to code. Another possibility would have been to do the inlining of the functions inside the processes calling them, and transforming the resulting code into a PROMELA process. Our choice implies to handle the problem of functions

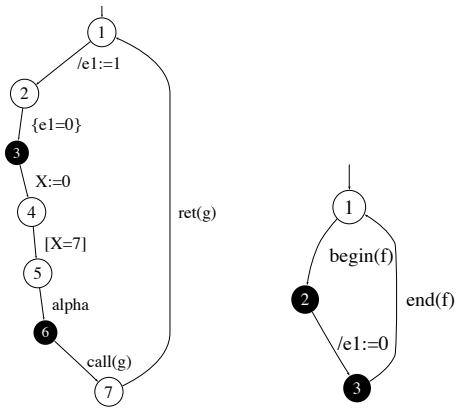


Fig. 2. Automata for Module 1. The left automaton corresponds to Process 1, the right one to function f. The variable X is used to model time. We distinguish between clock guards (square brackets) and guards on variables (curly brackets).

that can be called by multiple different processes. We perform a preprocessing on the SystemC program that consist in duplicating and renaming such functions in order to ensure that each function is called by only one process. This is possible because the function calls we consider are used for *communications* and therefore do not exhibit recursion (this argument also applies when choosing to inline the functions). The number of copies to do for a given function is bounded statically by the number of instantiated processes: while SystemC has recently added a feature to create processes dynamically, this feature is not used (to our knowledge) for Transaction Level models because the number of needed processes is linked to the number of master ports to drive, and ports cannot be created dynamically. In this article's examples, function parameters and return values are not represented but they could be taken into account by using global variables for both.

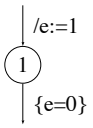
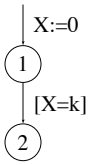
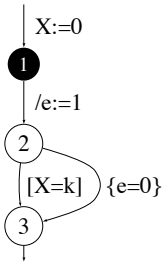
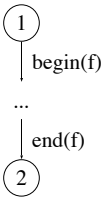
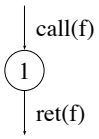
All time values appearing in the SystemC model must be integer multiples of some basic time granularity. We also assume that all variables can be declared globally, without any naming conflicts. All these restrictions can be ensured by simple preprocessing of the SystemC program without loss of generality.

3.2 Intuitive Idea: Representation with Automata

Our translation to PROMELA can also be seen as a translation into a set of automata. Each process and each function is translated into one interpreted automaton, *i.e.* manipulating some variables. The variables are used 1) to represent shared variables in the SystemC code 2) to encode SystemC events and 3) to count time. In the latter case, we will talk about *clocks*, which will be integers (see later). The transitions hold classically guards on variables as well as assignments. Clocks can either be tested for equality with a constant or a variable, or they can be set to 0. The automata derived from Module 1 of the simple example can be found in Figure 2. They simply reflect the control structure of the SystemC code.

The various automata then have to be asynchronously interleaved, respecting the non-preemptive specification of the scheduler. To obtain this, the automata have two different kinds of states, which we represent as black and white states. Black states represent local control flow: when an automaton is in a black state, it cannot not be preempted by the scheduler. White states basically represent `wait` statements: when an automaton is in a white state, it can be preempted, hence interleaved with any other process. A special case is the one of function calls, which directly transfer the control to the automaton that implements the function. This means we have to synchronize the transitions labeled by `call` (resp. `ret`) with the corresponding transition labeled by `begin` (resp. `end`).

Since we are mainly interested in the synchronization between different modules, we will not describe all the possible data processing inside the modules. This abstracted code could also be represented by encoding its control flow using only black states. The key elements of SystemC are translated in the following way:

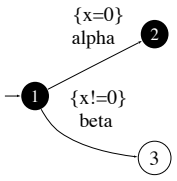
`wait(e)`		`e.notify()`	
`wait(k ns)`		`wait(e, k ns)`	
`f()`		`port.f()`	

3.3 Detailed Encoding

In the following we will show how our encoding in PROMELA deals with the three problematic parts of SystemC: non-preemptive scheduling, time-elapse and function calls.

Non-Preemptive Scheduling We distinguish between control points where the scheduler may transfer the control to another process (white states), and

internal control points (black states) of one process. The non-preemptive execution of one automaton is realized with an additional shared variable `M` of type `int`. This variable can take the value 0, to mean that any process can perform a step, or N, to mean that the process number N is the only one to be activated. A graphical representation and the actual SPIN encoding are shown in Figure 3.




```
myturn && State==1 && x==0
   -> alpha; State=2; M=this;
myturn && State==1 && x==1
   -> beta; State=3; M=0;
```

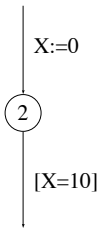
Fig. 3. Representation of non-preemption in PROMELA. The next value of `M` depends on the color of the next state.

This variable is set to 0 by each process that performs a `wait`. Otherwise, a process sets `M` to its own identifier, to indicate that it will take an additional step. For function calls, `M` is used to transfer the control to another process explicitly without the possibility of any other interleaving. We also considered a different encoding that only relied on atomic sections in PROMELA to model non-preemption. We sketch the problems with that approach in Section 3.5.

For simple examples, the use of an extra variable for ensuring atomicity is surely not efficient. We could use the `atomic` or `d_step` statement in PROMELA to ensure that no process can interleave a black state. In the trivial example in Figure 2, where each black state has exactly one incoming and one outgoing transition, we could also merge these and remove the black states all-together. But since the atomic behavior represented by the black states may be any complex control-flow, connected to white states by multiple in- and outgoing transitions, this simplification is not possible in the general case.

Clocks For every process that waits for time, we declare a clock (see Figure 4). We consider *discrete time*, that is every clock is an integer. A dedicated `clock_update` process increments the clocks synchronously as soon as no other process can run, which is tested using PROMELA's `timeout` statement. Hence time will never elapse if there is an instantaneous loop, *i.e.*, when there exists a cycle in the automaton which never performs a wait. Before a wait occurs, the process resets the corresponding clock to zero. We flag for each clock X whether the corresponding process is currently waiting on it, with a Boolean variable X`_used`. A clock is reset to zero when no process waits for it, and it is never increased in this time (it is a dead variable). Hence, the values of the clock are in the range between 0 and the value of the corresponding `wait`. This handling of clocks is similar to the one in discrete time Spin [3].

Function Calls For each function f, a global Boolean variable `F` is introduced. The effects of f are transformed into:

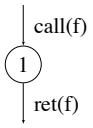


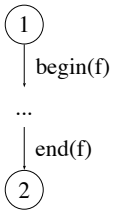

```
active proctype update_clock(){
end:
   do
   :: atomic{timeout && M==0 && X_used
              -> X=X+X_used;}
   od
}
   ...
active proctype A(){
do
:: atomic{myturn && state==1
          -> X=0; X_used=1;M=0;state=2}
:: atomic{myturn && state==2 && X=10
          -> X=0;X_used=0;...}
...
```

Fig. 4. Representation of clocks, which are needed to wait for time, as an automaton and in Promela. The process `update_clock` lets time elapse synchronously for all processes. Time is incremented whenever at least one clock is used, and only the time for clocks which are actually used is incremented.

- calling f: $\texttt{F} := 1$
- returning from the call: $\{\texttt{F} == 0\}$
- begin of the declaration of f: $\{\texttt{F} == 1\}$
- end of the declaration: $\texttt{F} := 0$

This is illustrated by Figure 5.






```
...
:: myturn && State==1
   -> f=1; State=2; M=2
:: myturn && State==2 && f==0
   -> State=1;M=this;
...
```

```
...
:: myturn && State==1 && f==1
   -> State=2; M=this;
:: myturn && State==2
   -> alpha; State=3; M=this;
:: myturn && State=3
   -> f=0; State=1; M=1;
...
```

Fig. 5. Representation of function calls in Promela. Assume that the caller has the identifier 1 and the function has identifier 2.

Additionally, a `call` sets `M` to the id of the automaton that implements f. Similarly, `end` sets `M` back to the value of the caller. Since each function is called by at most one process, this value is unique. Simply setting `M` without using `F` is not sufficient, because we have to make sure that the caller is blocked until the

function is completed. Since the function might perform a `wait`, the scheduler could otherwise decide to execute the caller again too early. Similarly, we have to prevent the function from being executed without being called.

If a function is never called, we could simply remove it. On the other hand, it might be the case that our model is only partially defined, that is: one or several modules of the system are not known, and considered as black-boxes (possibly with Byzantine behavior). In this case, we can give an over approximation of the possible behavior of unknown modules, by assuming that a called function can take an arbitrary time, and that each process may call any function at any time. This is reflected by the macro state between `call` and `ret`.

Functions are used both to model interrupts and transactions in TLM. The call representing an interrupt is directly done to another module. On the other hand, transactions are handled by a bus that performs a routing depending on the transaction's address. In the examples, we forget about the bus, and for each transaction directly call the recipient component's function. This simplification assumes we have determined manually to which component each transaction is addressed. In the general case, the code responsible for routing, as well as specific bus behavior can be taken into account by modeling the functions and processes inside the bus, as automata, like any other component. We would then need at least an address parameter for each transaction function calls.

SystemC Events Each SystemC event (`sc_event`) is encoded using a Boolean variable. The encoding has to reflect the fact that these events are not persistent: if the event is notified before a process waits for it, the notification is lost. This is done by setting the variable to 1 before yielding, which overwrites any previous notifications, the latters done by setting the variable to 0. This encoding assumes that only one process is waiting for the event. When multiple processes wait for the same event, each process taken individually can either miss or get the notifications of the event, depending on its order of execution. This situation can be handled simply by duplicating the event in order to get has many events as waiting processes, with only one waiting process per event. Other notification and wait constructs can be encoded following the same principle, as showed on some additional examples in the array of Section 3.2.

Simple Example The global definitions and the clock update for the example can be found in Figure 6. First the scheduling variable is declared. The macro `myturn` is an abbreviation to indicate that an automaton is enabled, *i.e.* it is either itself in a black state, or all automata are in white states. Since we do not have variables in the example, we only need to declare the variables for the events. Additionally, we have an integer value and a Boolean flag for each clock, indicating whether we are currently waiting for the clock. Time may elapse, *i.e.* the `update_clock` process is enabled when no other process can perform a step, all processes are in a white state and at least one process waits on time. Using `X=X+X_used` to update time ensures that time will only elapse for a clock that is used.

```
int M=0;
#define myturn (M==0 || M==this)

//Variables
bool e1=0;
bool e2=0;

//Functions
bool f=0;
bool g=0;

//Clocks
#define time_enabled timeout && M==0 && (X_used || Y_used)

int X=0;
bool X_used=0
int Y=0;
bool Y_used=0

active proctype update_clock(){
end:
   do
   :: atomic{time_enabled -> X=X+X_used; Y=Y+Y_used; }
   od
}
```

Fig. 6. Global definitions and the process for synchronous time elapse.

The SPIN code for Module 1 can be found in Figure 7. Each process has a variable to store its active state and its id. We could also use the process id that is automatically assigned by SPIN, but using a new value makes it easier to compute the id for function calls, without increasing the number of reachable states. The translation of the automata is straightforward. Each transition becomes an atomic action, that first tests whether the automata can run, *i.e.* is in the right state with the guard evaluating to true. Thereafter the effect of the transition is performed and the new state is set. At last, the scheduling variable is set according to whether the new state is black or white and whether a function call or termination is performed. Since a clock is explicitly set to 0 before each wait, and only at such a point, we also set the clock to "used" when leaving state 3. Similarly, when state 4 is left, we declare the clock as "not used anymore". Labels that abstract real, local computations like `alpha` are printed. The code for Module 2 can be found in Figure 8.

3.4 Validation of the Semantics

Our semantics for SystemC as an encoding in PROMELA is done in order to get the same effect when composing the automata with the asynchronous product of SPIN as when executing the corresponding SystemC code with a valid SystemC scheduler. To check that our semantics was corresponding to SystemC, we instrumented the SystemC models in order to produce test traces. We included in the PROMELA automata the same elements as produced in the SystemC text traces. Each observable action becomes a possible message in a global channel. When an action is performed, the corresponding message is sent to the channel. In order to reduce the size of the channel, the message is read from the channel again directly after. The SystemC trace we want to check is transformed into a `notrace` declaration; now we can use the build-in SPIN test to check whether the

```
active proctype module1(){
   byte state=1;
   byte this =1;

   do
   :: atomic{myturn && state==1 -> e1=1;state=2;M=0}
   :: atomic{myturn && state==2 && e1==0 -> state=3;M=this}
   :: atomic{myturn && state==3 -> X=0;X_used=1;state=4;M=0}
   :: atomic{myturn && state==4 && X==7 -> X=0; X_used=0; state=5;M=0}
   :: atomic{myturn && state==5 -> printf("MSC: alpha\n");state=6;M=this}
   :: atomic{myturn && state==6 -> g=1;state=7;M=4}
   :: atomic{myturn && state==7 && g==0 -> state=1;M=this}
   od
}

active proctype fun_f(){
   byte state=1;
   byte this =2;

   do
   :: atomic{myturn && state==1 && f==1 -> state=2;M=this}
   :: atomic{myturn && state==2 -> e1=0;state=3;M=this}
   :: atomic{myturn && state==3 -> f=0;state=1;M=3}
   od
}
```

Fig. 7. PROMELA code for Module 1 with the function f

```
active proctype module2(){
   byte state=1;
   byte this =3;

   do
   :: atomic{myturn && state==1 -> Y=0;Y_used=1;state=2;M=0}
   :: atomic{myturn && state==2 && Y==5 -> Y=0;Y_used=0;state=3;M=0}
   :: atomic{myturn && state==3 -> printf("MSC: beta\n");state=4;M=this}
   :: atomic{myturn && state==4 -> f=1;state=5;M=2}
   :: atomic{myturn && state==5 && f==0 -> state=6;M=this}
   :: atomic{myturn && state==6 -> e2=1;state=7;M=0}
   :: atomic{myturn && state==7 && e2==0 -> state=1;M=this}
   od
}

active proctype fun_g(){
   byte state=1;
   byte this =4;

   do
   :: atomic{myturn && state==1 && g==1 -> state=2;M=this}
   :: atomic{myturn && state==2 -> e2=0;state=3;M=this}
   :: atomic{myturn && state==3 -> g=0;state=1;M=1}
   od
}
```

Fig. 8. PROMELA code for Module 2 with the function g

behavior of the trace is a valid behavior of the SPIN model. The only problem was to produce all the text traces allowed by the SystemC specification given a SystemC program. While the specification allows any order of execution when multiple processes can be executed, the official SystemC simulator (as well as third party tools) takes only one order, deterministically. A modification of the official simulator existed in the lab, allowing to execute every possible scheduling. We used it to produce every possible traces, and checked that all these traces were included in our semantics.

3.5 Alternative Encoding

We also considered another encoding, which completely relied on PROMELA's atomic sections to model non-preemption. Each state was transformed into a `goto` label, followed by an atomic section that contained all outgoing transitions. Additionally, all black states where combined in one atomic section, including the labels. The semantics of PROMELA ensures that a jump from inside an atomic section to a label which is also contained in an atomic section preserves atomicity. The first problem with this encoding is that we have to inline all functions in order to tell to which point the function returns. But we have the benefit that we do not need the variable `M` for the scheduling, or the variables to hold the current state of each process. Furthermore, the implementation with `gotos` is much more efficient than using a loop with a non-deterministic choice.

But the main problem with this encoding is that the simulator of SPIN interleaves jumps from atomic section to another atomic section (although the documentation and the prover do the opposite). While SPIN proves properties that rely on the fact that such jumps are atomic, it also generates traces that violate the property.

The combination of `goto`s and atomic sections also make it impossible to use partial order reduction. Intuitively, `goto m1` and `atomic{goto m1}` are equivalent when `m1` is *inside* an atomic section, because a single statement is always atomic and every possible interleaving that could occur after the `goto` could as well occur before it. However, for the explicit atomic, SPIN will not allow any interleaving neither after nor before the goto, when partial order reduction is enabled.

```
bool X=0;

active proctype A(){
  assert(X==0);
}
```

```
active proctype B(){
  X=1;
  atomic{goto m0};
  atomic{skip;
    m0: X=0
  }
}
```

Fig. 9. A program whose verification depends on whether partial order reduction is enabled.

Consider for example the program in Figure 9. The assertion is violated, when B just executes `X=1` before A is executed. SPIN finds this error, when

the program is verified without partial order reduction, while it proves that all assertions hold when partial order reduction is enabled.

Because of these problems, this encoding does not have the benefits of an executable semantics. Therefore, we choose the not so efficient, but more robust encoding as the default.

4 Verification

4.1 Generic Properties

There are a number of properties that should hold for every TL model. First, it should never deadlock. For instance, a deadlock occurs when a process is waiting for an event that is never notified. A deadlock in the SystemC model corresponds directly to the fact that all PROMELA processes are blocked. With SPIN, this can be checked by verifying that no "invalid end states" exist, which is built in the prover. Since we only increase time when at least one process performs a wait on time, it can never be the case that the `clock_update` process runs forever, which would make it impossible for SPIN to detect a deadlock. On the other hand, when all processes terminate, the `update_clock` process will be blocked. Therefore, we explicitly declare the corresponding states as valid end states.

A deadlock might occur in the simple example in Figure 1, if we remove the `wait(5, SC_NS)` statement from the second process. Then the scheduler can choose to execute the second process first, and let it notify the event without a process waiting for it (this is indeed a common error for SystemC programmers). After that, both processes wait for events, but none is ever notified.

Another property we want to check is that no process runs forever without yielding. This can be expressed by the formula $\Box\Diamond M = 0$. For models with clocks, we can also check that time will always elapse, using the formula $\Box\Diamond enabled(update\_clock)$.

Of course, these last two properties are liveness formula and can only be checked if all abstractions preserve them (over-approximations of the behaviors preserve safety properties, but do not preserve liveness properties, in general). For the simple example, we do not need to perform any abstractions at all. If, however, the model becomes too large, the first abstraction that comes to mind is to remove all clocks, and to change every clock-guard to *true*. This implies that a process might halt in an arbitrary number of steps before a guard. But the property that a thread is monopolizing the behavior is independent from the clocks, so it can still be verified that way.

These tests work very well on the small example both with the possible deadlock and without. The proof is almost instantaneous, and the number of states remains small. We also checked these properties for the subset of a real-world MPEG decoding platform, which modeled the synchronization between the different components, with good results (see Figure 10).

This example contains many interleavings, a characteristic of real-world platforms. In the bug version, a deadlock may occur. The MPEG example is a mod-

	without bug		with bug		MPEG	
	time (s)	states	time (s)	states	time (s)	states
deadlock	< 0.1	35	< 0.1	9	< 0.1	126
no yielding	< 0.1	49	< 0.1	55	< 0.1	209
time elapse	< 0.1	57	< 0.1	9	< 0.1	226

Fig. 10. Benchmarks for the mpeg example.

ular version, with more parallel automata. All tests were performed on an Intel Celeron with 2.80 GHz and 1 GB RAM.

4.2 Checking Assertions

Checking assertions in the TL model is straightforward. Assertions are simply inserted at the corresponding transition, and directly written into the SPIN code. So far, we are mainly using assertions to check that some part of the model is never executed. This could also be modeled using specific error states. Since assertions are always safety properties, they are not effected by possible abstractions.

4.3 Benchmarks

Our test model 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 model 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. The results presented in Figure 11 focus on the main parameter which is the number of modules. We also tried to experiment with adding an arbitrary number of black states inside the processes, which for clarity is not in the table.

The normal encoding uses the global variable `M`, to assure atomicity, while the `goto` version is our alternative encoding. In order to allow all intended behavior, we disabled partial order reduction when checking the encoding with `goto`.

The entry NT (not tested) indicates that the checking has aborted due to lack of memory. Both encodings find the bug very fast.

When computing the whole state space, we see that the encoding using gotos is more efficient, but the number of states increases exponentially for both encodings. This is due to the increase of white states. We can solve the bug by waiting before the notification, in order to make sure that no event is lost.

# modules	3		5		7		9	
	time (s)	states	time (s)	states	time (s)	states	time (s)	states
normal: bug	< 0.1	32	< 0.1	48	< 0.1	64	< 0.1	80
normal: no-bug	< 0.1	3919	0.5	64831	11.8	104576	NT	NT
goto: bug	< 0.1	10	< 0.1	14	< 0.1	18	< 0.1	22
goto: no-bug	< 0.1	287	< 0.1	1535	< 0.1	7679	0.2	36863

# modules	11		13		15		17	
	time (s)	states	time (s)	states	time (s)	states	time (s)	states
normal: bug	< 0.1	104	< 0.1	120	< 0.1	136	< 0.1	152
normal: no-bug	NT	NT	NT	NT	NT	NT	NT	NT
goto: bug	< 0.1	34	< 0.1	38	< 0.1	42	< 0.1	46
goto: no-bug	1.1	172031	7	786431	47	353894	NT	NT

Fig. 11. Benchmarks for the chain call example.

While this makes the model completely deterministic, the number of states is still growing exponentially. Adding deterministic, local computations, increases the number of reachable states linearly for the normal encoding, and not at all for the encoding with gotos.

4.4 Comments on Performance

There are two possible sources for state explosion, making the model too large for automatic verification: the clocks and the interleaving between the processes. Modeling time by integers is usually not a good idea. However, our clocks are always bounded by the time of the corresponding `wait`, which is usually a rather small value, and since the clocks are updated synchronously, the actual increase of the state space is moderate. The size of the state space depends on the maximum time a process waits for, and the number of unrelated `wait` statements in parallel processes.

One way to cope with the state explosion is to use Spin not for formal verification, but for intensive testing. This is encouraged by the fact that in the benchmark the existing bugs were found very fast. This also allows to use more efficient algorithms in Spin, like hash-compact search, which only give approximate results.

On the other hand, our benchmark shows that introducing white states lead to state-explosion, while introducing black states has only a minor impact. Typical case-studies contain mostly control-flow, and only a few `waits`; therefore we are confident that we can model check programs of interesting size with our approach.

5 Related Work

The problem of SystemC having no *official* formal semantics is not new. Most of the research work studying SystemC in a formal context starts giving it a semantics in another well-defined formalism.

The majority of the approaches to give a semantics to SystemC are limited to its RTL subset, that is models describing synchronous circuits in detail. For instance, [9] expresses the semantics for the RTL subset in terms of Abstract State Machines. In this work, the processes executes concurrently, while the SystemC specification explicitly says the opposite (*non-preemptive scheduling*). [18] does the same using denotational semantics, but without taking into account the notion of control-flow inside the processes. These previous works have in common the fact that the target formalism that is used does not have concrete tools, and therefore it is impossible to check on examples that the given semantics is faithful. Moreover, the lack of connection to verification tools questions the possible applications for these works.

The approach followed by CheckSyC [5] goes one step further, in the sense that it provides a complete chain from SystemC parsing to formal verification, with relatively good experimental results. The main idea in this work is to recognize and extract well-known synchronous automata from RTL SystemC models and to reuse tools that model-check efficiently synchronous hardware. [19] follows the same idea using the GNU SSA (static single assignment) form of GCC for parsing the code and the synchronous language Signal [1] as the target formalism. Additional benefits, such as compositional reasoning, are presented but still on the RTL subset, whereas we are interested in handling models at higher abstraction levels.

The work in [11] apparently supports any SystemC program, by separating *hardware* (combinatorial functions, FSM, etc., corresponding to the RTL subset) from *software* (other unconstrained SystemC processes) to use different verification approaches. This separation is useless for us, since code for Transaction Level Models, while modeling hardware, falls in the second category. The semantics for SystemC is given by parallel automata with rendez-vous. Variables encoding the status for each process are used in the global state space to model the effect of the scheduler. However, nothing is said on `wait(`*`time`*`)` statements, which is a non-trivial point of the semantics.

We already experimented with the connection of SystemC to a proof engine, with the objective of analyzing Transaction-Level Models. The first output of this work was the tool-chain LusSy [14]. Starting from a SystemC program, we use a SystemC front-end to parse it, generate an intermediate representation called HPIOM made of communicating synchronous automata. We can then generate Lustre or SMV code to connect to a variety of proof engines. The connection to Lustre provides both provability and executability. The work presented here differs on several points: the first one is that LusSy uses a SystemC-independent intermediate formalism, and models the details of the scheduler using an explicit automaton. As opposed to this, we are using here a representation with automata in which the notion of non-preemption is built-in. The details of the scheduler do

not need to appear in a separate automaton, but the main scheduling principles are reflected by the automaton encoding, in such a way that the product of SPIN does the intended work. The second difference is that LusSy uses a synchronous formalism, while we are experimenting here with SPIN, which is asynchronous. Finally, the encoding in LusSy over-approximates the possible behavior when time elapses. Whereas the values waited for by the processes determine their order of execution, the encoding considers any possible order. The present work with SPIN computes the actual order in which the processes will be executed. The abstraction in LusSy is conservative for verifying safety properties, but at the expense of larger state spaces.

6 Further Work and Conclusion

We have presented a way of translating TL models written in SystemC into PROMELA. This is one way of giving a formal semantics to SystemC. We use this encoding to perform verification of TL models, like checking for deadlocks and assertions.

The asynchronous encoding seems to be worth further investigation, compared to a synchronous one. When translating SystemC to a synchronous framework, the atomicity between two white states is obtained by a quite complex synchronisation between the automata for the processes, and the automata that represents the scheduler. Conversely, when translating SystemC to PROMELA, the atomicity is built-in. Therefore, if the number of white states is small compared to the number of black states, the formal verification should be easier for the asynchronous encoding

On the other hand, translating SystemC into Lustre or SMV has the advantage of producing a symbolic description of the system that can be exploited by symbolic model-checkers and abstract-interpretation tools.

The use of SPIN is probably better for bug tracking, while the use of a symbolic tool is probably better for performing aggressive abstractions and approximate property verification.

Right now, the transformation from SystemC to PROMELA is manual. While interesting as a first approach to the problem, it would be necessary to implement the principles presented here in a complete tool-chain to apply the approach on a larger case-study and compare it with the synchronous encoding. This would mean to reuse a front-end like Pinapa [15], that we developed for LusSy, a transformation into a structure representing the particular form of automata used here, and a SPIN code generator. We already have a prototype for the data structure and the code generator, but the biggest part of the work is the transformation from the actual SystemC code.

References

1. A. Benveniste, P. le Guernic, and C. Jacquemot. Synchronous programming with events and relations: the SIGNAL language and its semantics. Technical Report 459, IRISA, Rennes, France, 1989.
2. G. Berry. The foundations of Esterel. *Proof, Language and Interaction: Essays in Honour of Robin Milner*, 2000. Editors: G. Plotkin, C. Stirling and M. Tofte.
3. D. Bosnacki and D. Dams. Discrete-time Promela and Spin. In *Formal Techniques in Real-Time and Fault-Tolerant Systems*, volume 1486 of *LNCS*, pages 307–310. Springer-Verlag, 1998.
4. M. Bozga, S. Graf, I. Ober, I. Ober, and J. Sifakis. The IF toolset. In *4th International School on Formal Methods for the Design of Computer, Communication and Software Systems: Real Time, SFM-04:RT, Bologna, Sept. 2004*, LNCS Tutorials, Springer, 2004.
5. R. Drechsler and D. Große. CheckSyC: An Efficient Property Checker for RTL SystemC Designs. In *ISCAS*, volume 4, pages 4167– 4170, May 2005.
6. Peter Feiler. Architecture Analysis & Design Language (AADL). Technical Report AS5506, SAE International, 2004.
7. F. Ghenassia, editor. *Transaction Level Modeling with SystemC, TLM Concepts and Applications for Embedded Systems*. Springer-Verlag, 2005.
8. N. Halbwachs, F. Lagnier, and C. Ratel. Programming and verifying critical systems by means of the synchronous data-flow programming language LUSTRE. *IEEE Transactions on Software Engineering, Special Issue on the Specification and Analysis of Real-Time Systems*, September 1992.
9. D. Hoffmann, J. Gerlach, J. Ruf, T. Kropf, W. Mueller, and W. Rosenstiehl. The Simulation Semantics of SystemC. In *DATE*, pages 64–70, 2001.
10. G. J. Holzmann. *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley Professional, 2004.
11. D. Kroening and N. Sharygina. Formal Verification of SystemC by Automatic Hardware/Software Partitioning. In *MEMOCODE*, pages 101–110, 2005.
12. Kim 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.
13. K. L. McMillan. The SMV system, November 06 1992.
14. M. Moy. *Techniques and Tools for the Verification of Systems-on-a-Chip at the Transaction Level*. PhD thesis, INPG, Grenoble, France, December 2005.
15. M. Moy, F. Maraninchi, and L. Maillet-Contoz. Pinapa: An extraction tool for systemc descriptions of systems-on-a-chip. In *EMSOFT*, September 2005.
16. M. Moy, F. Maraninchi, and L. Maillet-Contoz. LusSy: an open Tool for the Analysis of Systems-on-a-Chip at the Transaction Level. *Design Automation for Embedded Systems*, 10(2-3):73–104, September 2006.
17. Open SystemC Initiative. *IEEE 1666: SystemC Language Reference Manual*, 2005. `www.systemc.org`.
18. A. Salem. Formal Semantics of Synchronous SystemC. In *DATE*, volume 1, pages 10376–10381, 2003.
19. J.-P. Talpin, P. Le Guernic, S. Shukla, and R. Gupta. Compositional behavioral modeling of embedded systems and conformance checking. *International Journal on Parallel processing, special issue on testing of embedded systems*, 2005.