A SystemC/TLM semantics in Promela and its Possible Applications

Claus Traulsen, Jérôme Cornet, Matthieu Moy, Florence Maraninchi

A SystemC/TLM semantics in Promela and its possible applications

Claus Traulsen[1,2], 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 described LusSy [14] 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 a *asynchronous* formalism, namely Promela (the input language for Spin). We comment on the possible uses of this new encoding, and compare it with the synchronous encoding.

1 Introduction

SystemC [16] is a C++ library/language used for the description of Systems-on-a-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 the *Transaction Level Models* (TLM) [5] 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 *in parallel*. 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 can yield 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 communications *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 become to be 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. Sometimes this formalism only reflects the register-transfer-level (RTL) and cannot be used to express the specificities of a TL model. 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 the transaction level of abstraction, it is often quite far from real SystemC designs (see Section 5).

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 [6], SMV [11] or Esterel [2] tool-chains; if we translate it into an asynchronous formalism, we have access to Spin [8], IF [3], 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 [3] or UPPAAL [9] 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 [14, 12], 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 [11] 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. While the latter approach produces more readable results, the fact is that existing verification tools can not deal directly with *ad-hoc* products, but usually 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. The translation into Promela, which has a well defined semantics, is another way to give a formal semantics to SystemC/TLM. Thanks to the simulator provided with Spin, the semantics is also executable, and it will be possible to test the faithfulness of the semantics 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? It may depend on the type of property.

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) 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 communications 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.

Since we are only interested in modeling at transaction level, we make some strong assumptions on the SystemC code we analyze. We assume that we never use a built-in primitive channel or the synchronous update mechanism, which are used for RTL modeling. We consider two types of communications between modules: transactions and interrupts. Both types use blocking function calls (*i.e.*, the caller is blocked until the callee terminate its code) implemented using the SystemC TLM Library [15]. Interrupts proceed directly to the target module whereas transaction have to pass through a bus model that routes them to the corresponding target module, depending on the transaction address.

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 7ns, before performing some action α. Here α is the abstraction of some real, local computation, an alternative way to see this is: the computation α takes 7ns. 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 5ns before performing β (or performs β, which takes 5ns). 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. The two processes will always be executed in lockstep.

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).
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 `port.f()` is determined by the link between the port `port` 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.

Pre-processing for 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 assume that we have performed a preprocessing on the SystemC program ensuring the following property: Each function is called by at most one process. The preprocessing consists in duplicating and renaming functions which are called multiple times. We ignore recursive calls, which is sound for transaction level models (communications functions are not used to compute values). Function parameters and return values are not represented but can be taken into account by using global variables for both.

To avoid such a pre-processing copy of the functions, we would need to use a formalism in which a single object can model a process that is being executed in several distinct instances at the same time. Petri nets would do the job, but the verification tools that are able to deal with such models (symbolically, without expanding the model) are usually more complex than Spin, and they are tailored for the cases where the number of copies is not known statically. In our case, since the number of copies of a function is indeed known statically, we do not need the expressive power of Petri nets, and the expansion can be performed before the verification tool is called.

We also assume that only one process waits for each event. Removing this limitation is a matter of encoding. n processes waiting for an event e can be transformed into n processes each waiting for its own event e_i.

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 again be ensured by simple preprocessing of the SystemC program.

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 automaton. Guards and effects of transitions can either be the usual tests and assignments to Boolean or integer variables, or they can consider clocks. 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.

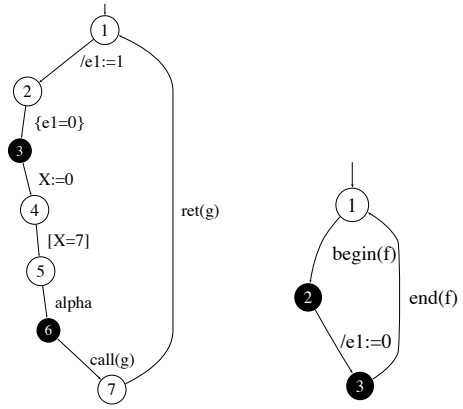
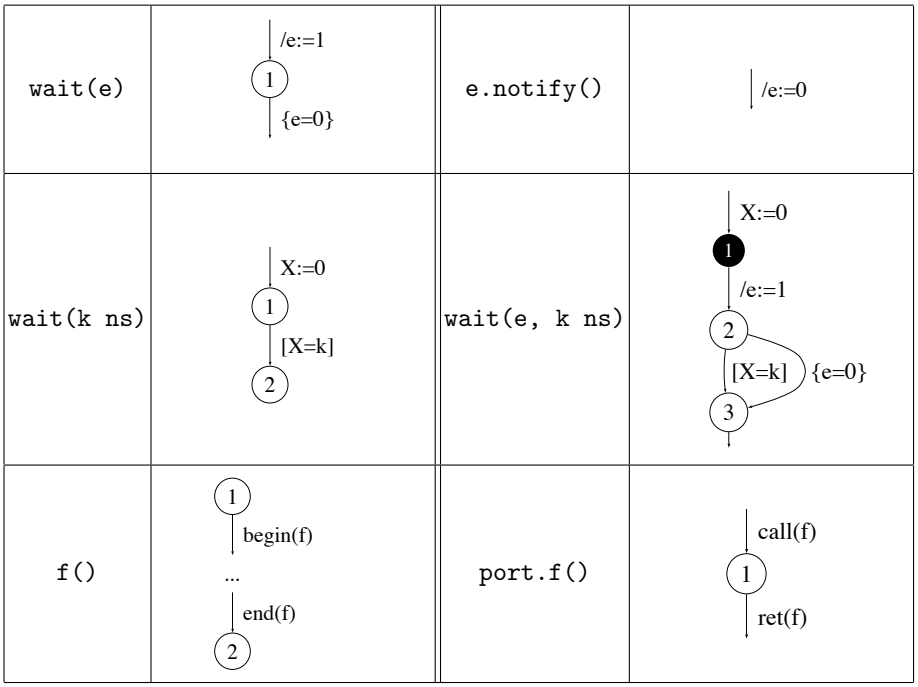


Fig. 2. Automata for Module 1

The 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 may 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:

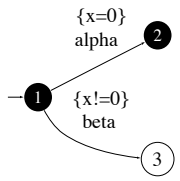


3.3 Detailed Encoding

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 processes can perform one 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.

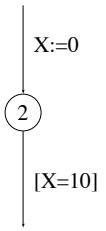



```
myturn && State==1 && x==0
   -> alpha; State=2; M=this;
myturn && State==1 && x==1
   -> beta; State=3; M=0;
```

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 preemptivity. 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` 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.




```
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.

Clocks For every process that waits for time, we declare a clock (see Figure 4). Since time is considered discrete, every clock is represented by 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 a 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`.

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

- calling f: `F` $:= 1$
- returning from the call: $\{$`F` $== 0\}$
- begin of the declaration of f: $\{$`F` $== 1\}$
- end of the declaration: `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 && Stae==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 no process call a function, we could simply remove the function. 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.

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 a 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 assures 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 Promela 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.*, it is in the right state and the guard evaluates 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.

```
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

3.4 Validation of the Semantics

The main benefit of a semantics for SystemC in Promela is the possibility to formally reason about properties of the model. But the simulation of the semantics is also very important.

Since SystemC itself is executable, having an executable semantics does not look like a big gain. But since we are performing some abstractions and transformations on the model, the simple possibility to execute the semantics and compare the resulting traces with the original SystemC model gives us confidence in the validity of the approach. See appendix B of [12] for an example on the problems that can occur when using a non-executable language to give semantics to a model.

```
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

This worked very well, for our first encoding, which did not use `goto`s. However, we tried another expression of the semantics using a combination of atomicity and jumps (see section 3.5), which lead to unexpected results.

The scheduling of the parallel processes in the model is non-deterministic according to the SystemC specification. In practice, it depends on the implementation of the SystemC scheduler. An interactive simulation of the Promela model makes these choices explicit, by prompting the user.

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 `goto`s 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, which 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. That the output of a verification depends on verification parameters which should not affect them, makes it hard to understand, why a specific behavior is possible or not. 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. A deadlock might, for example, occur 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 this states as a valid end state.

A deadlock might occur in the simple example in Figure 1, if we remove the `wait(5, SC_NS)` 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). In the following, both processes wait for events, but none is ever notified.

Another property we want to assure, 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 of the clocks, so the property can still be verified.

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 sub-set of a real-world mpeg decoding platform, which modeled the synchronization between the different components, with good results:

	without bug		with bug		mpeg	
	[s]	states	[s]	states	[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

All tests were performed on a Intel Celeron with 2.80GHz and 1GB 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. 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 oder reduction when checking the encoding with `goto`.

# modules	3		5		7		9	
	[s]	states	[s]	states	[s]	states	[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	
	[s]	states	[s]	states	[s]	states	[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

The entry NT, for not tested, indicates that we had abort the checking for lack of memory. Both encoding 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. 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 linear 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.

A number of other approaches have been proposed for the description of heterogeneous hardware/software systems with an emphasis on formal analysis. See, for instance, Metropolis. In this type of approach, the (formal) definition of the description language is part of the game. Other works like SystemC$^{\mathrm{FL}}$ [10] define a new language from scratch, but claim its similarity with SystemC. As opposed to SystemC, SystemC$^{\mathrm{FL}}$ doesn't have a non-preemptive scheduler, doesn't seem to manage a notion of "event" (which is the basic synchronization primitive on top of which everything else is built in SystemC),... Our approach is different. We want to deal with actual SystemC code, and the actual SystemC semantics.

An early work on SystemC semantics is [7], where the semantics is expressed in terms of Abstract State Machines. It only models SystemC 1.0 which was the only available at that time, and has an inaccurate model of the scheduler: in their model, the processes are executed in parallel whereas the language reference manual explicitly says the opposite.

[17] does the same with another formalism (denotational semantics). It models only a very strict subset of SystemC, with explicit limitation both on the kind of processes and on the content of processes (only `wait` statements and assignments are modeled, without a notion of control-flow).

While those approaches are theoretically interesting, the fact that the target formalism is abstract limits the possible applications: one can reason "on paper" about denotational semantics and abstract state machines, but until these formalism get a concrete syntax and an associated toolkit, these reasonings will not be automatizable or verifiable automatically. Furthermore, the question of the faithfulness of the semantics is not solved: starting from a language without a formally-defined semantics, one can not do anything better than examining the proposed semantics and just "claim" its faithfulness.

The approach followed by CheckSyC [4] goes one step further, providing a complete chain from the SystemC source code to a proof engine. They get relatively good experimental results. However, this work do not target the same SystemC subset as we do: we are not interested in RTL synthesizable programs, but target the transaction level model.

We already experimented with the connection of SystemC to a proof engine. The first output of this work was the tool-chain LusSy [12]. 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 are defined by the product. The second difference is that LusSy uses a synchronous formalism, while we are experimenting here with Spin, which is asynchronous.

[18] describes a general approach to extract a behavioral type from a system using the polychronous model of computation (using Signal [1] as a support language). This applies to languages and execution models such as Java, SystemC and SpecC. Their starting point is the GNU SSA (static single assignment) form, which is the intermediate representation in GCC, since version 4.0. The SSA form has the interesting property that each basic block in the abstract syntax tree can be executed in a single clock tick in Signal, leading to an elegant encoding of the SystemC code.

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 [13], 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. Marius Bozga, Susanne Graf, Iulian Ober, Ileana Ober, and Joseph 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.
4. Rolf Drechsler and Daniel Groe. CheckSyC: an efficient property checker for rtl SystemC designs. In *IEEE International Symposium on Circuits and Systems, 2005. ISCAS.*, volume 4, pages 4167– 4170, May 2005.
5. Frank Ghenassia, editor. *Transaction Level Modeling with SystemC, TLM Concepts and Applications for Embedded Systems*. Springer-Verlag, 2005.
6. 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.
7. Dirk Hoffmann, Joachim Gerlach, Juergen Ruf, Thomas Kropf, Wolfgang Mueller, and Wolfgang Rosenstiehl. The simulation semantics of systemC, December 21 2001.
8. Gerard J. Holzmann. *The SPIN Model Checker*. Pearson Education, 2003.
9. Kim G. Larsen, Paul Pettersson, and Wang Yi. UPPAAL in a Nutshell. *Int. Journal on Software Tools for Technology Transfer*, 1(1–2):134–152, October 1997.
10. Ka L. Man. Formal communication semantics of systemC^{FL}. In *DSD*, pages 338–345. IEEE Computer Society, 2005.
11. K. L. McMillan. The SMV system, November 06 1992.
12. Matthieu Moy. *Techniques and Tools for the Verification of Systems-on-a-Chip at the Transaction Level*. PhD thesis, INPG, Grenoble, France, December 2005.
13. Matthieu Moy, Florence Maraninchi, and Laurent Maillet-Contoz. Pinapa: An extraction tool for systemc descriptions of systems-on-a-chip. In *EMSOFT*, September 2005.
14. 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*, 10(2-3):73–104, September 2006.
15. Open SystemC Initiative. *SystemC Transaction Level Modeling Library 1.0*, 2004. `www.systemc.org`.
16. Open SystemC Initiative. *IEEE 1666: SystemC Language Reference Manual*, 2005. `www.systemc.org`.
17. Ashraf Salem. Formal semantics of synchronous systemc. In *DATE*, pages 10376–10381, 2003.
18. 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.