A Schedulerless Semantics of TLM Models Written in SystemC via Translation into LOTOS

Olivier Ponsini, Wendelin Serwe

A Schedulerless Semantics of TLM Models Written in SystemC via Translation into LOTOS

Olivier Ponsini and Wendelin Serwe

INRIA, 655, avenue de l'Europe, 38334 Saint-Ismier Cedex, France,
{olivier.ponsini,wendelin.serwe}@inria.fr

Abstract. TLM (Transaction-Level Modeling) was introduced to cope with the increasing complexity of Systems-on-Chip designs by raising the modeling level. Currently, TLM is primarily used for system-level functional testing and simulation using the SystemC C++ API widely accepted in industry. Nevertheless, TLM requires a careful handling of asynchronous concurrency. In this paper, we give a semantics to TLM models written in SystemC via a translation into the process algebra LOTOS, enabling the verification of the models with the CADP toolbox dedicated to asynchronous systems. Contrary to other works on formal verification of TLM models written in SystemC, our approach targets fully asynchronous TLM without the restrictions imposed by the SystemC simulation semantics. We argue that this approach leads to more dependable models.

1 Introduction

Systems-on-Chip combine several hardware components with embedded software in a single integrated circuit. TLM (Transaction-Level Modeling) was introduced to cope with the increasing complexity and time-to-market pressure of Systems-on-Chip by using reference descriptions closer to system-level. Compared to traditional RTL (Register Transfer Level) based design flows, TLM reduces both the development time of virtual test platforms and the simulation time, allowing to run the embedded software earlier and to perform functional testing of the system.

TLM is still a rather informal concept. In this paper, we use the definition given in [4]. TLM models describe both system architecture and behavior. The hardware part of a system is not required to be completely detailed, but only to be sufficient to develop and run the embedded software. A TLM model is a set of interconnected modules, whose behavior is represented by asynchronous concurrent processes communicating only through transactions and events. Dealing with asynchronous concurrency is known to be difficult due to the many possible interleavings of concurrent tasks. TLM models are no exception: explicitly and completely defining the synchronizations between processes is the key to ensure model correctness; unfortunately, this is also very error prone, so that formal validation techniques are required.

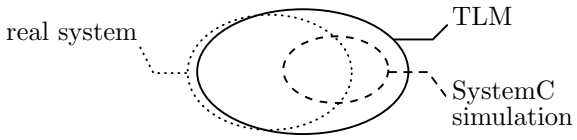

Fig. 1. Observable system behaviors

In general, TLM models are written in system-level design languages, among which the SystemC standard [16] has become the most popular. SystemC is a C++ library providing (1) types, methods, and macros to describe systems, including hardware, at various abstraction levels and (2) a simulation kernel, in particular a scheduler, to simulate the execution of the modeled systems. Simulation greatly helps functional validation, yet it is well-known that testing cannot prove the absence of errors, since exhaustiveness is impossible (at reasonable cost). This holds all the more for asynchronous concurrent systems, where the process interleaving space must be covered in addition to the data space. On the other hand, formal methods and tools dedicated to concurrent systems have a proper handling of asynchronous concurrency and can guarantee a property for all possible executions of a system.

Moreover, and to the contrary of the nondeterministic and asynchronous nature of TLM, the scheduler of the SystemC simulation kernel is nonpreemptive, has synchronous features, and imposes that for the same input the order of process execution does not vary from run to run. These properties of the scheduler are useful for testing and debugging, since they allow to reproduce a simulation run. However, they do not suit verification needs since they restrict the set of executions. Thus, as will be shown in Sect. 3.3, one may miss executions leading to erroneous states of the system.

Figure 1 compares the possible observable behaviors of different models and the real system. A TLM model is an abstraction of the real system, thus its behavior does not always exactly coincide with the behavior of the real system; further differences might be introduced during synthesis, since the step from TLM to hardware is not formally defined. A model based on the SystemC scheduler can only exhibit a subset of the TLM model behaviors. We advocate that verification over the increased number of behaviors of TLM leads to more dependable models and thus to more dependable embedded software. Therefore, we aim at a formal semantics of TLM models written in SystemC independent from the SystemC scheduler and its simulation semantics.

The contribution of this paper is a formal semantics of TLM defined via a translation from a TLM-subset of SystemC into the standard process algebra LOTOS [8], so as to enable the use of CADP (Construction and Analysis of Distributed Processes) [3], a rich formal verification toolbox that allows on the fly, compositional model-checking and equivalence checking of asynchronous systems. The translation has the following features:

– It regards SystemC as a description language for TLM and does not superimpose the SystemC simulation semantics to TLM semantics. This allows to exhibit behaviors that might occur if the embedded software were run on hardware, but that would not be revealed by simulation with SystemC.
– It is parameterized to control asynchronous behaviors according to verification needs; in particular, the SystemC scheduler semantics can be reproduced if required.
– It preserves the architectural hierarchy (encapsulation of modules) of the SystemC description, since TLM models are not flattened to an unstructured set of processes. This facilitates compositional verification as well as going back and forth from the formal model to the SystemC code.

The rest of the paper is organized as follows. Related work is presented in Sect. 2. Section 3 surveys TLM and SystemC and discusses the limitations of the SystemC simulation semantics. The translation itself and a brief introduction to LOTOS are given in Sect. 4. Some experimental results are discussed in Sect. 5. Section 6 concludes.

2 Related Work

Both TLM and SystemC lack an authoritative semantics to which formal approaches could refer. As for TLM, there is no standard definition: [17] is a proposal seeking better interoperability between TLM models but it is still incomplete. The works addressing the issue of giving a formal semantics for TLM and/or SystemC differ mainly as regards the formal methods used and level of abstraction, *e.g.* cycle-accurate RTL, algorithmic level, or the so-called transaction levels, themselves divided into TLM PV (Programmer's view) untimed models and TLM PVT (PV + Timing) timed models. Focussing on a particular level allows to optimize the formal model, but may require to choose a subset of SystemC constructs. For instance, SystemC signals are important to RTL but not to TLM. The chosen level may also determine the target formalism: for instance, synchronous models seem adequate for RTL, whereas asynchronous models seem appropriate for TLM.

We distinguish four main lines of work targeting either low-level SystemC, full SystemC, TLM with SystemC, or TLM alone.

SystemC was initially designed as a language for modeling circuits, providing low-level hardware constructs such as hardware signals. A first line of work targets this low-level SystemC. For instance, [5] addresses temporal property checking for SystemC RTL level descriptions. [14] proposes an operational semantics for low-level SystemC, which uses distributed Abstract State Machines but limits asynchrony to two modules: the SystemC scheduler and the set of all SystemC processes, i.e. there is no asynchrony between the SystemC processes. [18, 6] define a denotational semantics for a restricted subset of SystemC. In this paper, we are interested in modeling of systems above this low-level SystemC.

A second line of work targets full SystemC, *i.e.* low-level SystemC as well as higher levels of abstraction. Various formalisms have been used to give a

semantics to full SystemC, *e.g.* labeled Kripke structures [10], synchronous languages [19], Petri nets [9] or process algebra [12]. Contrary to our proposal, these approaches do not take advantage of the higher abstraction levels, in particular they are tied to the synchronous features of low-level SystemC. Moreover, most of these works flatten the hierarchical description allowed by SystemC.

A third line of work is interested in dedicated methods for TLM descriptions written in SystemC. In [13], a verification tool chain for TLM is developed upon synchronous communicating automata, with interfaces to synchronous languages and their model checkers. Recently, [20] proposed an encoding of a TLM-subset of SystemC in PROMELA, allowing a connection with verification tools based on asynchronous formalisms. However, these works have in common with the previous ones to integrate the SystemC scheduler as part of the formalized model, either explicitly [9, 13] or implicitly [10, 12, 19, 20]. Whereas the problem of covering more schedules is an active research direction (see [7] for instance), these works do not depart from the SystemC simulation semantics and its restrictions on the possible schedules, *e.g.* nonpreemption. Our work belongs to this third line, since we give a formal semantics to a TLM-subset of SystemC: we translate the PV level of TLM into the process algebra LOTOS. More importantly, we aim at verification of TLM, and distinguish between SystemC as a description language and SystemC as a simulation tool, *i.e.* we give a formal semantics to TLM models written in SystemC, not to TLM models as simulated by SystemC. Thus, in contrast to other works, our formal semantics of TLM does not integrate the SystemC scheduler and the simulation semantics it implies.

The fourth line consists in relatively few works interested in formalizing TLM models not written in SystemC. For instance, [21] presents the modeling of an on-chip bus protocol in LOTOS at TLM level. Seeking more generality, [15] gives first a formal definition of what a transaction should be (according to criteria applying to transactions in databases) and then derives guidelines how to implement complying transactions in SystemC, considering the SystemC scheduler specificities. Our approach is just the opposite: we start from a SystemC implementation of a TLM model and translate it into a formal language.

3 TLM Subset of SystemC

Although TLM is in principle not tied to a particular language, the SystemC standard [16] has gained wide acceptance for describing TLM models. In this section, we first outline the SystemC subset relevant to TLM PV. Then, we describe the SystemC scheduler and its limitations as regards verification.

3.1 TLM Principles

Basically, a TLM model is a set of components whose behavior and communication aspects are clearly separated. The behavior of a component is captured by a set of concurrent processes. Communication between modules is captured by transactions, which can transfer data and/or trigger events.

TLM models range over several abstraction subclasses whose boundaries depend mainly on timing accuracy and data granularity. In this work, we focus on PV models as this level is intended for embedded software development and functional verification. PV models are untimed and the data granularity should fit the intended application rather than the hardware micro-architecture (*e.g.* a frame for a video processing unit, rather than the actual bus packets or hardware signals).

The PV model of computation is summed up by four points [4, p. 34]:

1. concurrent execution of independent processes,
2. respect for causal dependencies between processes using system synchronization,
3. bit-true behavior, and
4. bit-true communication.

Points (1) and (2) define an asynchronous model of computation where process interleavings are only controlled by explicit synchronizations between processes; points (3) and (4) ensure that functional verification is possible.

3.2 SystemC Description Language

The SystemC C++ library defines classes and convenience macros to describe system architectures. The components of a system are *modules* (`SC_MODULE`) whose behavior is specified by a set of processes. For simulation efficiency and synthesis concerns, SystemC distinguishes several kinds of processes, but we focus, without loss of generality, on *threads* (`SC_THREAD`). To our knowledge, TLM models have a static number of processes; thus, we do not handle process spawning.

Modules contain *ports* (`sc_port`) through which they communicate with other modules. Ports are connected either to other ports or *channels*.

Channels are used to encapsulate communication protocols, they are either primitive or hierarchical. Primitive channels, *e.g.* `sc_signal`, are only used in modeling levels lower than PV. Hierarchical channels are not very different from other modules.[1] Hence, we will not make the distinction in the sequel.

A *transaction* is a call of a method in another module (through a port). The calling process executes the code encapsulated in the other module. The two involved modules exchange data through method parameters and return value. Methods used for transactions are declared in *interfaces* (`sc_interface`) inherited by the module implementing the transaction.

Processes can also synchronize via *events* (`sc_event`). A process can suspend its execution waiting for a specific event `e` (`wait(e)`). When `e` is notified (`e.notify()`), all suspended processes waiting for `e` resume their execution; if there is no waiting process, the notification is lost.

[1] The difference between hierarchical channels and ordinary modules is that the former also inherit from interfaces specifying the implemented transactions.

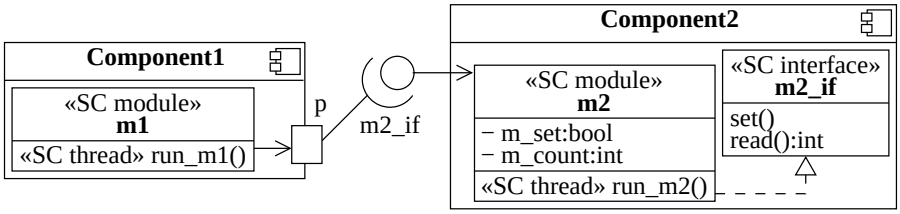

Fig. 2. UML diagram of the set-counter example

Finally, SystemC has also constructs related to timing and to synchrony — *e.g.* delayed wait and notification, update-request mechanism of primitive channels — implemented in its simulation kernel by the so-called *delta-cycles*. These features are inherited from earlier versions, before the language was extended to support TLM, and they are not relevant to our discussion of the PV level.

For illustration, we use the set-counter example depicted in Fig. 2. Its SystemC code is:

```
SC_MODULE(M1) {
  sc_port<m2_if> p;
  SC_CTOR(M1) { SC_THREAD(run_m1); }
  void run_m1() { p->set(); cout << p->read(); }
};
class m2_if: virtual public sc_interface {
  virtual void set() = 0;
  virtual int read() = 0;
};
SC_MODULE(M2): public m2_if {
  SC_CTOR(M2): m_set(false), m_count(0) {
    SC_THREAD(run_m2);
  }
  void run_m2() {
    while(true) { wait(e); m_count++; m_set=false; }
  }
  void set() { m_set=true; e.notify(); }
  int read() { return m_count; }
private:
  sc_event e; bool m_set; int m_count;
};
int sc_main (int argc, char *argv[]) {
  M1 m1("module1");  M2 m2("module2");
  m1.p.bind(m2); sc_start(-1); return 0;
}
```

Module `m2` allows module `m1` to set a flag (`m_set`) and to read how many times it was set. The program entry point `sc_main` instantiates the two modules and binds port `p` of `m1` to module `m2`. Thus, in the thread `run_m1`, method calls `p->set()` and `p->read()` are transactions, in which `run_m1` executes the meth-

ods `set` and `read` defined in `m2`. These two methods are declared in interface `m2_if` inherited by module `m2`. Transaction `set` sets data member `m_set` to true and notifies event `e`. Transaction `read` returns the value of counter `m_count`. The behavior of `m2` is described by `run_m2`: it waits for the notification of `e`, increments `m_count` and resets `m_set`.

3.3 SystemC Scheduler

In order to simulate the concurrent execution of several processes on a single processor, the SystemC simulation kernel uses a scheduler to select the process gaining control of the processor. In this section, we discuss two limitations, as regards verification, of the SystemC scheduler: immutable order of process execution and nonpreemption.

Immutable Order of Process Execution. Except for explicit synchronizations, TLM does not impose any order of execution of processes: they run concurrently with an asynchronous semantics. Synchronizations between processes define a partial order on process interleavings, allowing several different schedules. Although the choice of a schedule by the SystemC scheduler is implementation dependent, the SystemC standard [16] requires that all the simulation runs with the same input will choose the same schedule. This helps debugging since it allows to easily reproduce an erroneous behavior of the system. The downside is that other schedules may lead to different erroneous behaviors that will never show up with SystemC simulation.

In the set-counter, asynchronous concurrency means that the `wait(e)` statement of process `run_m2` can occur either before or after process `run_m1` has performed the transaction `p->set()`, *i.e.* either before or after event `e` is notified. This leads to two different system behaviors.

If the notification of `e` occurs *before* the wait, then the event is lost and `run_m2` will deadlock since it will eventually wait for an event that will not be notified anymore. If the notification of `e` occurs *after* the wait, then `run_m2` will resume its execution and eventually proceed.

These two behaviors are intended in TLM. However, the SystemC scheduler imposing an order of process execution no matter the number of simulation runs, only one out of the two behaviors will be simulated. Hence, simulation may miss the deadlock.

This deadlock can be prevented by replacing the statement `wait(e)` by `while(!m_set) wait(e)`. With this modification, a correct synchronization between the wait of `run_m2` and the notification of event `e` by `run_m1` is ensured.

Non-Preemption. The SystemC scheduler is not preemptive, *i.e.* a process runs without interruption until it explicitly gives control back with a `wait` statement. This is known as collaborative multithreading and is generally found easier to program with than preemptive multithreading, *e.g.* it simplifies access control to shared variables. However, this major difference with the concurrent model

of computation of TLM may hide interleavings intended in the model and the final system.

For instance, when `run_m1` gains control, it performs the two transactions, `set` and `read`, in a row. Although `run_m2` is resumed by transaction `set`, with the simulation semantics of SystemC, this second process has no chance to execute before `run_m1` explicitly gives control back, *i.e.* on termination in this case. Therefore, the value of `m_count` read by `run_m1` during the transaction `read` is never updated by `run_m2`.

However, in real asynchronous concurrency as in TLM, once process `run_m2` is resumed, the update of `m_count` could occur before or after `run_m1` performs its transaction `read`. Therefore, the value of `m_count` read by `run_m1` can either be the updated one or not. These two behaviors are intended in the TLM model, but only the second one is permitted by the SystemC scheduler. Hence, this synchronization problem between successive transactions is missed by simulation, and by formal semantics based on the SystemC simulation semantics.

4 Translation of the TLM-subset of SystemC into LOTOS

In this section, we briefly present LOTOS and outline the translation of TLM descriptions written in SystemC into LOTOS. In the following, $\overline{\texttt{L}}$ denotes a list $\texttt{L}_1, \ldots, \texttt{L}_n$ of, depending on context, gates, variables, type-variable couples or values. We use the set-counter without deadlock as a running example.

4.1 LOTOS

The standard process algebra LOTOS (Language Of Temporal Ordering Specification) [8] allows to describe asynchronous concurrent processes communicating and synchronizing by *rendez-vous* on *gates*. LOTOS specifications are composed of a data part and a behavior part. For a complete description of LOTOS, we refer the reader to existing tutorials, such as [1]; in the following we briefly introduce the notions occurring in the examples of this paper.

Data values and operations are described by algebraic specifications in the style of ACTONE [2]. Types define a collection of *sorts*, *operations* on sorts and *equations* describing the meaning of operations. The verification toolbox CADP also allows to use external C data types. In the examples of this paper, we suppose that we are given an implementation of Booleans and natural numbers.

Behaviors are expressed by *terms* combining processes with algebraic operators. Figure 3 gives a grammar of behaviors; lower case identifiers stand for terminals and upper case identifiers for non terminals (P is a process name, G a gate name, X a variable name, S a sort name, and F a function name).

The semantics of LOTOS is formally defined by labeled transition systems. Here, we only sketch the meaning. A *rendez-vous* "$G\ \overline{O}$`;` B" on a *gate* G allows to communicate several values $\overline{O}$, called *offers*, either for emission (`!`) or reception (`?`); then behavior B is executed. Hidden gates $\overline{G}$ of B in "`hide` $\overline{G}$ `in` B" are unobservable, and unavailable for synchronization with other behaviors.

$$
\begin{array}{lll}
B ::= & G\ O_1 \ldots O_n\ \texttt{;}\ B & \textit{rendez-vous} \\
| & \texttt{hide}\ G_1, \ldots, G_n\ \texttt{in}\ B & \textit{hiding} \\
| & B_1\ \texttt{[]}\ B_2 & \textit{choice} \\
| & B_1\ \texttt{|||}\ B_2 & \textit{interleaving} \\
| & B_1\ \texttt{|[}G_1, \ldots, G_n\texttt{]|}\ B_2 & \textit{parallel} \\
| & B_1\ \texttt{>>}\ \texttt{accept}\ X_1\texttt{:}S_1, \ldots, X_n\texttt{:}S_n\ \texttt{in}\ B_2 & \textit{sequence} \\
| & \texttt{exit(}V_1, \ldots, V_n\texttt{)} & \textit{termination} \\
| & \texttt{[}V\texttt{]->}\ B & \textit{guard} \\
| & \texttt{let}\ X\texttt{:}S\texttt{=}V\ \texttt{in}\ B & \textit{variable definition} \\
| & P\texttt{[}G_1, \ldots, G_m\texttt{](}V_1, \ldots, V_n\texttt{)} & \textit{process call} \\
O ::= & \texttt{!}V \mid \texttt{?}X\texttt{:}S & \textit{offer} \\
V ::= & X \mid F\texttt{(}X_1, \ldots, X_n\texttt{)} & \textit{value}
\end{array}
$$

Fig. 3. Grammar of LOTOS behaviors

"B_1 `[]` B_2" implements a nondeterministic choice between behaviors B_1 and B_2. "B_1 `|[`$\overline{G}$`]|` B_2" is the parallel composition of B_1 and B_2 synchronizing on the gates $\overline{G}$; pure *interleaving* "B_1 `|||` B_2" is the special case where $\overline{G}$ is empty. Synchronization on gates with offers only occurs if the offers are compatible (same number and types, same values for matching emissions). In the sequence "B_1 `>> accept` $\overline{X\texttt{:}S}$ `in` B_2", on successful termination, behavior B_1 uses the operator "`exit(`$\overline{V}$`)`" to pass results $\overline{V}$ of types $\overline{S}$ to B_2 through variables $\overline{X}$ (of types $\overline{S}$). A behavior B can be *guarded* by a Boolean expression V: "`[`V`]->` B". A "`let` $X\texttt{:}S\texttt{=}V$ `in` B" construct allows to define a variable X of sort S that can be used in B and is initialized to value V. Finally, a behavior B can be encapsulated in a recursive process P as follows: "`process` P `[`$\overline{G}$`](`$\overline{X\texttt{:}S}$`):` E`:=` B `endproc`" where E is either `noexit` or `exit(`$\overline{S}$`)`.

4.2 Overview of the Translation

Our translation into LOTOS maps SystemC threads, transactions, shared variables, and modules into the single concept of LOTOS process. SystemC types are translated into LOTOS types. Two additional LOTOS processes are required. The *event manager* process is an implementation of the event communication mechanism used in TLM. The *lock manager* process is not the translation of a TLM concept. It is added to the LOTOS model so as to adjust the degree of asynchrony to verification needs.

Several LOTOS implementations may exist for a given concept (*e.g.* event communication). Due to lack of space, we will only briefly mention these alternatives. Translation of shared variables and locks are based on standard techniques from process algebra (e.g. [11]). In the following, we will write thread for a LOTOS process corresponding to the translation of a SystemC thread.

4.3 Variables of Modules

A variable v of a module can be shared, *i.e.* accessed by several threads or transactions of the module. If this is not the case, the variable is added as a

parameter of the thread using it. Otherwise, it is necessary to introduce a dedicated process $\texttt{shared}_v$ that offers rendez-vous to read and write v. In LOTOS, this supplementary process avoids synchronizations between processes accessing the same shared variables.

If the type of v is `Bool`, $\texttt{shared}_v$ can be defined as:

```
process shared_v[read_v,write_v](v:Bool): noexit :=
  read_v !v; shared_v[read_v,write_v](v) []
  write_v ?new_v:Bool; shared_v[read_v,write_v](new_v)
endproc
```

A single process comprising all read/write rendez-vous suffices to handle all shared variables of a module.

Moreover, if a shared variable of a module m is accessed by threads of m and by threads of other modules, then it is necessary to duplicate the gates accessing this variable in order to avoid n-ary rendez-vous between the threads of m and the other threads. This is the case in the set-counter example with the shared variables of module `m2`: `m_set` (resp., `m_count`) is accessed for reading (resp., writing) by both `run_m2` and transaction `set` (resp., `read`). Consequently, we introduce the two supplementary gates `w_m_set_ext` and `r_m_count_ext` that will be used by the transactions. The LOTOS code for the shared variables of module `m2` is:

```
process shared_var[w_m_set,r_m_set,w_m_count,r_m_count,
                   w_m_set_ext,r_m_count_ext]
                  (m_set:Bool, m_count:Nat) : noexit :=
  w_m_set ?v:Bool;
  shared_var[w_m_set,r_m_set,w_m_count,r_m_count,w_m_set_ext,
             r_m_count_ext] (v,m_count)
 []
  r_m_set !m_set;
  shared_var[w_m_set,r_m_set,w_m_count,r_m_count,w_m_set_ext,
             r_m_count_ext] (m_set,m_count)
 []
  w_m_count ?v:Nat;
  shared_var[w_m_set,r_m_set,w_m_count,r_m_count,w_m_set_ext,
             r_m_count_ext] (m_set,v)
 []
  r_m_count !m_count;
  shared_var[w_m_set,r_m_set,w_m_count,r_m_count,w_m_set_ext,
             r_m_count_ext] (m_set,m_count)
 []
  w_m_set_ext ?v:Bool;
  shared_var[w_m_set,r_m_set,w_m_count,r_m_count,w_m_set_ext,
             r_m_count_ext] (v,m_count)
 []
  r_m_count_ext !m_count;
  shared_var[w_m_set,r_m_set,w_m_count,r_m_count,w_m_set_ext,
             r_m_count_ext] (m_set,m_count)
endproc
```

4.4 Locks

For various reasons (*e.g.* debugging, efficiency, knowledge of the system), it may be desirable to control the level of asynchrony in (parts of) the LOTOS model. This is possible using different *locking* strategies: locks ensure mutually exclusive execution of selected code parts. For instance, one lock per module acquired by each transaction of the module prevents simultaneous transactions in the same target module, whereas a single global lock acquired by each thread reproduces the nonpreemptive semantics of the SystemC scheduler. Several lock granularities for different parts of the system can be used to fine tune the desired behaviors of a model.

One or several lock manager processes can propose rendez-vous to acquire or release locks on gates `lock` and `free`. These gates may take an offer identifying a desired lock if a centralized lock manager is used for several locks.

To illustrate the use of locks with the set-counter, we implement a locking policy that prevents transactions and the thread of `m2` from executing simultaneously. Thus, a thread or a transaction starts by acquiring the lock of `m2`, which is then freed only on suspension or termination. The corresponding lock manager process is:

```
process lock_manager[lock,free](m2_locked : Bool) : noexit :=
  [not(m2_locked)] -> lock !m2; lock_manager[lock,free](true)
 []
  free !m2; lock_manager[lock,free](false)
endproc
```

4.5 Event Communication

For an event e, an event manager process is used to record which processes are waiting for e and to resume them all nondeterministically on notification of event e. A `wait(`e`)` is translated into a sequence `suspend !id_p !`e`; resume !id_p` where `id_p` is the identifier of the waiting process. A e`.notify()` translates into a rendez-vous `notify !`e; if a process p is waiting for e, the event manager offers a rendez-vous `resume !id_p` to resume p.

For each process p possibly waiting for an event e (this can be known statically), we use one Boolean parameter of the event manager to record whether p is waiting for e or not. An event manager for an event `e` with two possible waiting processes $p1$ and $p2$ can then be defined as:

```
process event_manager[notify,resume,suspend]
                     (id_p1_e, id_p2_e:Bool): noexit :=
  suspend !id_p1 !e;
  event_manager[notify,resume,suspend](true,id_p2_e)
 []
  suspend !id_p2 !e;
  event_manager[notify,resume,suspend](id_p1_e,true)
 []
```

```
  (notify !e;
   (
     ([id_p1_e]->resume !id_p1; exit [] [not(id_p1_e)]->exit)
    |||
     ([id_p2_e]->resume !id_p2; exit [] [not(id_p2_e)]->exit)
   ) >> event_manager[notify,resume,suspend](false,false) )
endproc
```

For each process, an additional Boolean parameter can suffice to encode whether the process is waiting for a conjunction or disjunction of events. Finally, a single process may manage all event/process combinations, or several processes can be used to manage events local to groups of modules.

In the set-counter, there is only one event and `run_m2` is the only thread waiting for it, so the event manager is simpler than the more generic one above:

```
process event_manager [n, r, s](b_run_m2 : Bool) : noexit :=
  s; event_manager[n, r, s](true)
 []
  [not (b_run_m2)] -> n; event_manager[n, r, s](b_run_m2)
 []
  [b_run_m2] -> n; r; event_manager[n, r, s](false)
endproc
```

4.6 Threads and Transactions

A SystemC thread T is translated into a LOTOS process whose behavior is the translation of the body of T. C++ constructs occurring in threads are translated as follows: an assignment to a local variable becomes a `let` construct, a conditional branching becomes a choice between behaviors guarded by mutually exclusive conditions (`[cond]->if_part [][not(cond)]->else_part`), a loop becomes a recursive process, and a method call becomes either a process call or a call to a C function if the method only processes data without synchronizing. In this latter case, CADP calls the C function to compute a value if needed.

A transaction is also translated into a process P. Unlike threads, transactions may have input (request) and output (response). Inputs become parameters of P while outputs become results returned by P via the `exit` operator. Calling a transaction through a port is calling the corresponding process – which one is statically known.

In the set-counter example, there are two transactions, `set` and `read`. Their translation is:

```
process set[lock,free,notify,w_m_set_ext] : exit :=
  lock !m2;
  w_m_set_ext !true; notify;
  free !m2; exit
endproc
```

```
process read[lock,free,r_m_count_ext] : exit(Nat) :=
  lock !m2;
  r_m_count_ext ?n:Nat;
  free !m2; exit(n)
endproc
```

Then, the translation of thread `run_m1` calling the transactions is:

```
process run_m1[lock,free,notify,cout,w_m_set_ext,
               r_m_count_ext] : noexit :=
  set[lock,free,notify,w_m_set_ext]
    >> read[lock,free,r_m_count_ext]
      >> accept n:Nat in cout !n; stop
endproc
```

Finally, the translation of thread `run_m2` is:

```
process run_m2[lock,free,resume,suspend,r_m_set,w_m_set,
               r_m_count,w_m_count] : noexit :=
  lock !m2; run2[lock,free,resume,suspend,r_m_set,w_m_set,
                 r_m_count,w_m_count]
where
  process run2[lock,free,resume,suspend,r_m_set,w_m_set,
               r_m_count,w_m_count] : noexit :=
    r_m_set ?v:Bool;
    (
      [not(v)]-> suspend; free !m2; resume; lock !m2;
        run2[lock,free,resume,suspend,r_m_set,w_m_set,
             r_m_count,w_m_count]
     []
      [v]-> r_m_count ?n:Nat; w_m_count !n+1; w_m_set !false;
        run2[lock,free,resume,suspend,r_m_set,w_m_set,
             r_m_count,w_m_count]
    )
  endproc
endproc
```

4.7 Modules and Complete System

A SystemC module is translated into a parallel composition of the process handling its state variables with the translation of its threads.

As an example, the following is a translation of a module M with two threads P_1 and P_2, which use the gates $\overline{A_{\text{int}}}=\overline{A_{\text{int}_1}}\cup\overline{A_{\text{int}_2}}$ to access variables of M and the gates $\overline{A_{\text{ext}}}=\overline{A_{\text{ext}_1}}\cup\overline{A_{\text{ext}_2}}$ to access variables of other modules. Contrary to SystemC, transactions are not encapsulated in the owner module[2]. Consequently, the scoping rules of LOTOS require M to expose its variables via gates (here $\overline{A_{\text{t}}}$) to make them accessible to threads of other modules (through transactions of M).

[2] This solution has been investigated and leads either to complex handling of contexts, or to duplication of code; it is not exposed here.

```
process M[notify,resume,suspend,A_t,A_int,A_ext]: noexit :=
   shared_var[A_int,A_t](var)
  |[A_int]|
   (P1[notify,resume,suspend,A_int1,A_ext1]
   |||
    P2[notify,resume,suspend,A_int2,A_ext2])
endproc
```

The module `m2` of the set-counter is translated into:

```
process m2[lock,free,notify,resume,suspend,
           w_m_set_ext,r_m_count_ext] : noexit :=
  hide r_m_set,w_m_set,r_m_count,w_m_count in
  (
    shared_var[w_m_set,r_m_set,w_m_count,r_m_count,
               w_m_set_ext,r_m_count_ext] (false,0)
   |[w_m_set,r_m_set,w_m_count,r_m_count]|
    run_m2[lock,free,resume,suspend,r_m_set,w_m_set,
           r_m_count,w_m_count]
  )
endproc
```

Module `m1` contains only one thread, thus `m1` is translated into a call to `run_m1`.

The entire system is the parallel composition of all modules with the event and lock managers. Modules are synchronized with each other on the gates to access their variables (the union of the $\overline{A_{\text{ext}}}$ and $\overline{A_{\text{t}}}$ gates).

Finally, the translation of the entire set-counter system is:

```
  (
    ( run_m1[lock,free,notify,cout,w_m_set_ext,r_m_count_ext]
     |[w_m_set_ext,r_m_count_ext]|
      m2[lock,free,notify,resume,suspend,
         w_m_set_ext,r_m_count_ext]        )
   |[notify,resume,suspend]|
    event_manager[notify,resume,suspend](false)
  )
 |[lock,free]| lock_manager[lock,free](false)
```

5 Experimental Results

We developed the LOTOS model of the set-counter without deadlock using two locking policies: without (a global lock has to be acquired by each thread and released on suspension or termination) and with thread preemption (no lock at all). The former reproduces the behaviors of the TLM model observable with the SystemC simulation kernel. The latter allows transaction interleavings, as required by TLM semantics. We also wrote a μ-calculus formula expressing what values of the counter may be read by initiator module `m1` (*cf.* Sect. 3.3).

For both versions, we used CADP [3] to generate the corresponding automata and check the property (as expected, it holds only with preemption). We also

Table 1. Results for the set-counter example

i=initiator t=target		**1i 1t**	**2i 1t**	**1i 2t**	**2i 2t**	**3i 2t**
w/ preemption	generation time (s)	1.1	1	1.4	1.8	28.4
	number of states	77	1,201	1,109	48,149	1,940,977
	formula checking (s)	0.1	0.3	0.1	0.7	35.8
w/o preemption	generation time (s)	1	1	1.4	1.5	1.5
	number of states	35	141	149	770	3,334
	formula checking (s)	< 0.1	< 0.1	< 0.1	< 0.1	1.1
checking w/o $\subsetneq$ w/ (s)		1.4	1.7	2.7	32.6	450.4

verified that, modulo branching equivalence, the model without preemption was included in the model with preemption, but not vice versa.

Table 1 shows the results for different configurations of initiator (`m1`) and target (`m2`) modules. When several targets are available, each initiator performs `set` and `read` transactions with each target in sequence. Experiments were done on a Sun UltraSparc IIIi 1.6 GHz with 2 GB memory running Solaris 10 (time is in seconds and "generation" refers to the automata construction from LOTOS).

As a consequence of showing more behaviors, preemptive models (lines "w/") produce automata with a greater number of states than nonpreemptive models (lines "w/o"). However, first experiments show that minimization with respect to branching bisimulation reduces automata of preemptive models by factors up to 10^3. Therefore, compositional approaches might be very effective.

6 Conclusion

TLM models are nondeterministic and asynchronous, as may be the underlying hardware. Hence, they are difficult to apprehend and formal methods can help their understanding and verification. Since there is no formal semantics of TLM, most verification approaches refer to the simulation semantics of SystemC and its nonpreemptive scheduler. Such approaches cannot exhibit all behaviors of a TLM model, possibly leaving errors undetected, as we have shown in this paper.

We presented a translation from a TLM-subset of SystemC into LOTOS using a schedulerless semantics; our translation can be easily tuned to support the nonpreemptive semantics as a particular case. Although the interleaving semantics abstraction of concurrency, in which our approach is rooted, may not always correspond to physical true concurrency, it is widely accepted and proved efficient in many application domains. Experimenting our translation on several TLM models with CADP, we showed that our semantics is a strict superset of a nonpreemptive one and that the additional behaviors may reveal errors.

Automating a translation of TLM into LOTOS is a difficult task, since the former is informal whereas the latter has a precise formal semantics. The formalization of TLM is a necessary first step, to which the translation rules of this paper contribute.

References

[1] T. Bolognesi and E. Brinksma. Introduction to the ISO Specification Language Lotos. *Computer Networks and ISDN Systems*, 14(1):25–59, Jan. 1988.

[2] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification 1: Equations and Initial Semantics*, vol. 6 of *EATCS Monographs on Theoretical Computer Science*, 1985.

[3] H. Garavel, F. Lang, R. Mateescu, and W. Serwe. CADP 2006: A Toolbox for the Construction and Analysis of Distributed Processes. In *CAV*, vol. 4595 of *LNCS*, pp 158–163, July 2007.

[4] F. Ghenassia, ed. *Transaction-Level Modeling with SystemC: TLM Concepts and Applications for Embedded Systems*. Springer, 2005.

[5] D. Große and R. Drechsler. CheckSyC: An Efficient Property Checker for RTL SystemC Designs. In *ISCAS*, vol. 4, pp 4167–4170, May 2005.

[6] A. Habibi and S. Tahar. Design and Verification of SystemC Transaction-Level Models. *IEEE Transactions on VLSI Systems*, 14(1):57–68, 2006.

[7] C. Helmstetter, F. Maraninchi, L. Maillet-Contoz, and M. Moy. Automatic Generation of Schedulings for Improving the Test Coverage of Systems-on-a-Chip. In *FMCAD*, pp 171–178, 2006.

[8] ISO/IEC. Lotos — A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour. International Standard 8807, ISO, Genève, 1989.

[9] D. Karlsson, P. Eles, and Z. Peng. Formal Verification of SystemC Designs Using a Petri-net Based Representation. In *DATE*, pp 1228–1233, 2006.

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

[11] J. Magee and J. Kramer. *Concurrency: State Models and Java Programs*. Wiley, 2nd edn., Apr. 2006.

[12] K. L. Man. SystemC^{FL}: A Formalism for Hardware/Software Codesign. In *European Conference on Circuit Theory and Design*, vol. 1, pp 193–196, 2005.

[13] M. Moy, F. Maraninchi, and L. Maillet-Contoz. LusSy: A Toolbox for the Analysis of Systems-on-a-Chip at the Transactional Level. In *ACSD*, pp 26–35, June 2005.

[14] W. Müller, J. Ruf, D. Hoffmann, J. Gerlach, T. Kropf, and W. Rosenstiel. The Simulation Semantics of SystemC. In *DATE*, pp 64–70, Mar. 2001.

[15] B. Niemann and C. Haubelt. Towards a Unified Execution Model for Transactions in TLM. In *MEMOCODE*, pp 103–112, 2007.

[16] Open SystemC Initiative. *IEEE Standard SystemC Language Reference Manual*. IEEE Computer Society, 2006. IEEE Std 1666-2005.

[17] A. Rose, S. Swan, J. Pierce, and J.-M. Fernandez. *Transaction Level Modeling in SystemC*. Open SystemC Initiative, 2005. www.systemc.org.

[18] A. Salem. Formal Semantics of Synchronous SystemC. In *DATE*, pp 376–381, 2003.

[19] J.-P. Talpin, P. L. Guernic, S. K. Shukla, and R. Gupta. A Compositional Behavioral Modeling Framework for Embedded System Design and Conformance Checking. *International Journal of Parallel Programming*, 33(6):613–643, 2005.

[20] C. Traulsen, J. Cornet, M. Moy, and F. Maraninchi. A SystemC/TLM Semantics in Promela and its Possible Applications. In *SPIN Workshop*, vol. 4595 of *LNCS*, pp 204–222, July 2007.

[21] P. Wodey, G. Camarroque, F. Baray, R. Hersemeule, and J.-P. Cousin. LOTOS Code Generation for Model Checking of STBus Based SoC: The STBus Interconnect. In *MEMOCODE*, pp 204–213, June 2003.