A Comparison of Two SystemC/TLM Semantics for Formal Verification

Claude Helmstetter
INRIA - LIAMA
Beijing 100080, China
claude@liama.ia.ac.cn

Olivier Ponsini
INRIA
38334 Saint-Ismier Cedex, France
olivier.ponsini@inria.fr

Abstract

The development of complex systems mixing hardware and software starts more and more by the design of functional models written in SystemC/TLM. These models are used as golden models for embedded software validation and for hardware verification, therefore their own validation is an important issue. One thriving approach consists in describing the semantics of SystemC/TLM in a formal language for which a verification tool exists. In this paper, we use Lotos and the CADP toolbox as a unifying framework to define and experiment with two possible semantics for untimed SystemC/TLM, emphasizing either the nonpreemptive semantics of SystemC or the concurrent one of TLM. We also discuss and illustrate on a benchmark the qualitative versus quantitative performance trade-off offered by each semantics as regards verification. When associated with locks, our concurrent semantics appears both to provide more flexibility and to improve the scalability.

1. Introduction

Parts of the hardware development process, e.g. the synthesis task, usually require a description at the register transfer level (RTL). However, RTL descriptions are long to write and slow to simulate. This can be a significant drawback for other tasks — such as architecture exploration, embedded software simulation, interconnect protocol validation or quality-of-service evaluation — that would benefit from more abstract models of the real hardware.

Transaction-level modeling (TLM) [6] is intended to offer this higher level of abstraction. It allows to describe the architecture and the behavior of a system thanks to modules, concurrent processes and transactions through communication channels. Transactions abstract data transfers and synchronizations so as to accelerate both model design and simulation. SystemC [15] is the most popular language to describe TLM models of hardware and heterogeneous systems, i.e. mixing hardware and embedded software. It is a C++ library that includes a fast event-driven simulation environment and allows to reuse C or C++ code. In SystemC, transactions are implemented by method calls: a process can access data in another module without requiring any time-costly context switch in the simulator [6, 17].

SystemC/TLM models are used for functional testing of embedded software, hardware RTL, or communication protocols. In these models, details about the real hardware can be abstracted; most abstractions contain no timing information at all. As golden reference models, the validation of SystemC/TLM models themselves is a central issue. Formal verification techniques are challenged by the lack of a formal SystemC/TLM semantics, even for timed models [19]. The usual work-around is to give such a semantics by describing the translation of SystemC/TLM models into a formal language well suited for verification.

1.1. Contributions

Comparing the existing semantics is difficult as they are all based on different principles and representations, targeting different levels of abstraction and SystemC subsets. In this paper, we propose to use the *language of temporal ordering specification* (Lotos) and the *construction and analysis of distributed processes* toolbox (CADP) as a uniform framework (formalism and tools) in order to focus on the qualitative and quantitative comparison of two different semantics of untimed SystemC/TLM, namely a nonpreemptive one inspired from [13] and the concurrent one from [16]. The first one mimics the simulation semantics of SystemC by modeling faithfully the SystemC scheduler. The second one focuses on TLM semantics and frees itself from the SystemC simulation semantics, in order to improve the faithfulness with respect to the real system. Indeed, the simulation semantics is based on scheduler specificities that parallel circuits usually do not implement for efficiency reasons and that would prevent realistic executions from being analyzed.

To define these semantics, we describe two translations of untimed models into Lotos. We restrict our presentation to the translation of SystemC and TLM specific features and do not discuss translation of generic C++ code.

Moreover, we compare these semantics on a benchmark extracted from [18]; this will also allow us some comparison points with this third semantics of SystemC/TLM expressed in Promela.

1.2. Related works

Many SystemC semantics, defined as translations of SystemC features into a formal language, have already been published. Generally, the choice of the target language is motivated by the tools available for this language and their performances according to the kind of properties to prove. Moreover, to different levels of abstraction can correspond different subsets of SystemC.

For example, [3, 7] consider verification of hardware SystemC descriptions at the RTL and gate levels. Another semantics for low-level descriptions given in [1] is dedicated to the development of correct-by-construction SystemC programs thanks to the B method. Some other works target TLM, such as [14] where SystemC programs are manually translated into finite state machines. On the contrary, the tool chain [13] translates automatically TLM models into synchronous automata with variables; moreover, it provides some simple abstraction techniques (e.g. abstract address representation). More abstraction techniques are described in [12] together with a translation into labeled Kripke structures.

As TLM models are inherently asynchronous [6], dedicated formalisms coming with optimized tools for asynchrony have been investigated. For instance, Lotos [10], an asynchronous process algebra based on rendez-vous mechanism, for which the CADP [5] toolbox provides enumerative techniques for verification, has been used to validate the STBus interconnect [20]. More recently, Petri nets [11] and Promela [18] have been used to encode SystemC programs. The approach in [16] further acknowledges that asynchrony of TLM models should not be restrained by the simulation semantics of SystemC and proposes a distinctive concurrent semantics of TLM/SystemC descriptions.

1.3. Structure of the paper

After an overview of SystemC, TLM, Lotos and CADP in Section 2, we present a first encoding corresponding to the preemptive semantics of SystemC/TLM in Section 3. We propose a second encoding for the concurrent semantics of TLM in Section 4. Some experiments are specific to each encoding; these are presented at the end of Sections 3 and 4. We compare both encodings in Section 5, in order to check their correctness, and to evaluate their scalability. We recall the highlights of our work and conclude with Section 6.

2. System modeling and verification

In this section, we give a quick overview of the languages, libraries, and tools used in this work.

2.1. SystemC and the TLM library

SystemC is a C++ library that provides classes and macros to describe the architecture (`sc_module`, `sc_port`...) of heterogeneous systems and their behavior thanks to processes (`sc_thread`...) and synchronization mechanisms (`sc_event`...). The *static architecture* is built by executing the *elaboration phase*, which instantiates modules and binds their ports.

Next, the SystemC simulator *schedules* the SystemC processes. A SystemC process is either *eligible* or *running* or *waiting* for a SystemC event. There is at most one running process at a time. A process moves from eligible to running when it is elected by the scheduler. The elected process explicitly suspends itself when executing a `wait` instruction. If the running process notifies an event, then all processes waiting for this event move from waiting to eligible; note that SystemC events are not persistent and so the execution of a `notify` before a `wait` can lead to a deadlock.

In the present paper, we consider only the untimed *asynchronous subset* of SystemC, which is used for the design of functional models of Systems-on-Chip at the system-level (TLM). Fixed durations and δ-cycles should not be used in pure functional models as their use may hide some realistic behaviors [8, 9].

We call SystemC *transition* an atomic section of code from the SystemC scheduler point of view. A transition may execute zero, one or more accesses to shared objects (e.g. events and shared variables).

The TLM library built upon SystemC provides a *transaction* mechanism that allows a process of an *initiator* module to call methods exported by a *target* module. Names and attributes of exported methods depend on the *protocol*. In general, there are at least methods `read` and `write`, but the protocol for transactions modeling interrupts can be reduced to only one method without argument. An example is given by Figure 4 (Section 3.2).

2.2. Lotos and the CADP toolbox

Process algebras allow to specify the observable behavior of distributed systems with *terms* combining behaviors and algebraic operators. Formal reasoning about the behavior of systems is then possible by applying algebraic laws. In the standard process algebra Lotos [10], systems are a set of interacting and communicating concurrent behaviors. The following introduces the syntax and semantics of Lotos.

In Lotos, data values, data operations, and data structures are defined by *abstract data types*. Types are defined by *sorts*, operations on sorts and equations describing the properties of the operations. For instance, booleans and natural numbers are in the standard type library of Lotos.

Interaction and communication between two or more behaviors involve instantaneous *rendez-vous* synchronizations on *gates*. Term "G `;` B" expresses the rendez-vous on gate G followed by behavior B. Data is exchanged during rendez-vous through *offers* allowing either emission of a value V (`!`V) or reception in a variable X of sort S (`?`X`:`S). A rendez-vous on a gate with offers only occurs if all the participating behaviors present the same number of offers, with compatible sorts for receptions and the same value for matching emissions. "B_1 `|[`$G_1, \ldots, G_n$`]|` B_2" is the *parallel composition* of B_1 and B_2 synchronizing on the gates $G_1, \ldots, G_n$; pure interleaving "B_1 `|||` B_2" is the special case where there is no gate to synchronize on. Term "`hide` $G_1, \ldots, G_n$ `in` B" makes the gates $G_1, \ldots, G_n$ appearing in B unobservable and unavailable for synchronization with other behaviors. "B_1 `[]` B_2" is a *choice* and behaves either like B_1 or like B_2. "B_1 `>> accept` X_1`:`$S_1, \ldots, X_n$`:`S_n `in` B_2" is a *sequence* where behavior B_1, on successful termination, can use the operator `exit(`$V_1, \ldots, V_n$`)` to pass values $V_1, \ldots, V_n$ of sorts $S_1, \ldots, S_n$ to B_2 through variables $X_1, \ldots, X_n$. In CADP, the sequence operator involves a rendez-vous on an implicit termination gate between processes B_1 and B_2. "`[`E`]->` B" is a *guard* conditioning the execution of B to the truth value of boolean expression E. Term "`let` X`:`S`=`V `in` B" defines variable X of sort S initialized to value V for use in behavior B. Finally, "`process` P `[`$G_1, \ldots, G_n$`](`X_1`:`$S_1, \ldots, X_n$`:`S_n`):` E `:=` B `endproc`" encapsulates a behavior B in a recursive *process* P, where E is either `exit(`$S_1, \ldots, S_n$`)` or `noexit` and defines the sorts of the values returned on successful termination.

The semantics of a Lotos specification is formally defined by a *state graph*, also called an LTS (labeled transition system) – i.e. a set of states, and transitions labeled by gates and offers between states. CADP [5] includes a compiler from Lotos to LTS and many tools exploiting the LTS for simulation, model checking of modal μ-calculus formulae, equivalence checking, test generation, and performance evaluation.

However, methods based on LTS face the state space explosion problem. CADP addresses this problem with several techniques, among which *on-the-fly* and *compositional* approaches have been used in this paper. On-the-fly verification allows to explore the parts of an LTS relevant to the verified formula, without completely generating the LTS. In a compositional approach, the different parts of a system are generated separately, possibly with an interface reproducing the constraints of the environment. Then, each part is reduced according to equivalence relations, and recomposed with the other reduced parts using parallel and hiding operators. This usually leads to an LTS smaller than the one generated directly from the entire system. Moreover, the approach can also be used in combination with on-the-fly verification.

The tools of CADP can easily be combined in concise verification scenarios written in the *script verification language* (SVL) [4]. SVL offers high-level operators hiding the intricacies of format conversions and of command-line tools and options. For compositional verification, several strategies are predefined:

- "leaf reduction" applies a given reduction to the leaves of parallel composition operators in a behavior;
- "root leaf reduction" applies leaf reduction and then reduces also the result;
- "node reduction" is similar to leaf reduction with the addition that the given reduction is also applied to the result of each parallel composition operator.

By way of illustration, for each predefined compositional reduction strategy, Figure 1 presents the reduced parts (denoted by *red* operators) of `(A |[...]| B) |[...]| C`, where `A`, `B` and `C` are behaviors without parallel operator.

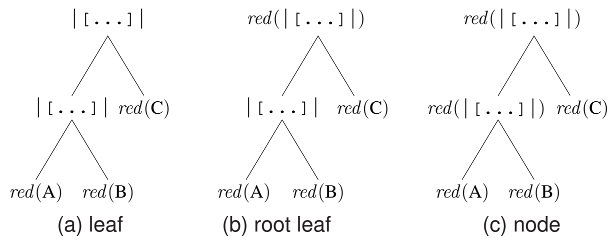

Figure 1. Predefined reduction strategies

3. Lotos encoding of SystemC/TLM

In the following, the word "thread" refers to SystemC processes and "process" is reserved for Lotos processes, in order to avoid any ambiguity.

3.1. Presentation of the encoding

SystemC threads and scheduling SystemC threads may contain long sections of atomic code grouping together many transactions. The Lotos language does not provide a native mechanism for long atomic sections. We chose to reuse the idea of a centralized scheduler that has been proposed first by Matthieu Moy in LusSy [13].

In the Lotos model, we define at least one Lotos process for each SystemC thread, plus one global process modeling the scheduler. The scheduler synchronizes the (Lotos processes modeling the SystemC) threads through the gates `elect` and `yield`, as described by Figure 2. One thread identifier (*Pid*) is associated with each thread. The Lotos code corresponding to a SystemC transition must be encapsulated between an `elect` gate, which allows the thread to run, and a `yield` gate, which allows the scheduler to elect another thread.

Our scheduler process implements the same functionality as the shared variable *M* of the Promela encoding described in [18].

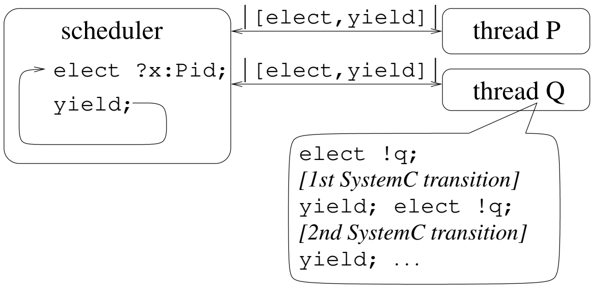

Figure 2. Synchronizing through a scheduler

SystemC events Here, we present an encoding of SystemC events such that there is no restriction on the number of threads that can wait or notify an event.

Each event is represented by a Lotos process. An event saves the list of thread identifiers that are waiting for it. When an event receives a `wait` request from a thread, it adds the thread identifier to its waiting list. When it receives a `notify` request, it enables all the threads of its waiting list and clears the list. The encoding is detailed by Figure 3. Gate `notifyA` corresponds to the call of the notify method in SystemC, whereas gate `notifyZ` corresponds to the return of this method and avoids that the thread execution interleaves with the execution of the Lotos sub-process `enable_all`.

SystemC modules and transactions We give here one possible encoding of transactions. This encoding allows to describe each SystemC module in a separate Lotos process, independently of its context.

A SystemC module contains threads, shared objects (e.g. events) and exported target methods (or transaction methods). Each of these elements is represented in Lotos by a process. All these processes are instantiated inside another Lotos process that represents the SystemC module itself.

A transaction method "`F`" is modeled such as a normal SystemC thread, but it starts with the rendez-vous "`F_request ?x:Pid`", and ends with the rendez-vous and recursive call "`F_response !x; F[...]`". On the initiator side, the call of the transaction "`port.F();`" in SystemC is translated by "`F_request !my_pid; F_response !my_pid;`". Sending the thread identifier is mandatory to execute wait statements inside the transaction, since `wait` and `enable` gates take the thread identifier as offer. Other parameters of the transaction can be added as offers of gate `F_request`, and the return values as offers of gate `F_response`. We give an example of this encoding in the next section.

The behavior of this encoding is not faithful with respect to SystemC semantics in the following case: if a thread is suspended on a wait instruction inside a transaction, then other threads that call the same transaction method will be blocked until the first thread resumes and exits the transaction. [18] suggests to duplicate a transaction method as many times as possible concurrent accesses; that implies to bind statically the number of concurrent calls and may lead to an increase of the model size. This can be done for our Lotos encoding too.

3.2. Presentation of the `chain` benchmark

We evaluate this encoding on the benchmark proposed in [18], whose works are the closest to ours. This benchmark consists of a chain of interrupt transmitter modules, whose length is parametrized by n. Modules communicate through transactions, and threads synchronize with events.

Figure 4 presents the SystemC original benchmark for $n = 1$. To increase n, one adds a transmitter module between the last transmitter and module Sink. There are always $n + 2$ SystemC threads (methods named `initiate`, `compute`, and `complete`) and $n + 1$ events (private attribute `e` of each module except Source). Each target module (Transmitter and Sink) exports a method `f` to the previous initiator module (Source or Transmitter) through a pair of TLM ports, symbolized on Figure 4 by a triangle in a square.

Figure 5 presents the architecture of the Lotos encoding. Each box represents a Lotos process, and the arrows with gate names represent the synchronizations between them. The content of each process is not described on this figure, but follows exactly the rules detailed in Section 3.1.

3.3. Experiments

We present here experiments on this SystemC/TLM semantics that cannot be directly compared with the semantics we present in next section. Comparison of the two semantics is the subject of Section 5.

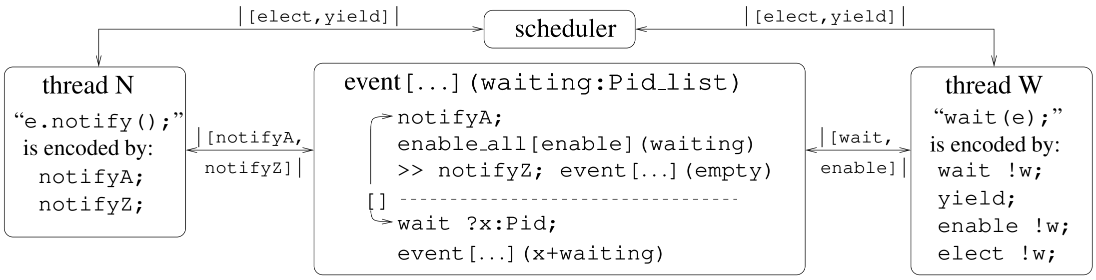

Figure 3. Generic encoding of SystemC events in Lotos with scheduler

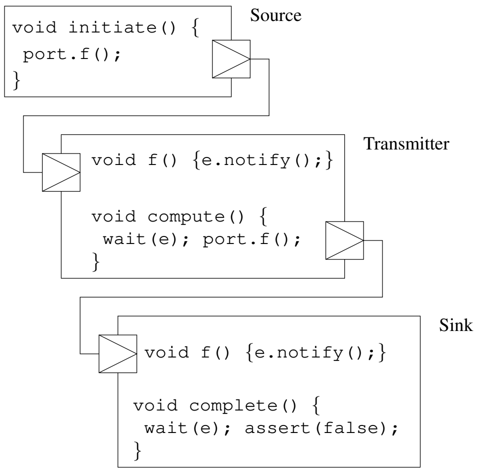

Figure 4. The `chain` benchmark for $n = 1$

False property evaluation We use the *evaluator* tool to check if the transition `assert_false` of module Sink is always accessible. Since the given property is false, *evaluator* returns a diagnostic with enough information to replay the counter-example with the original program and an interactive SystemC scheduler.

True property evaluation Also, we check a property involving the exploration of the whole model state space. This property consists in showing that from any state a given inexistent transition leads to no valid state.

Direct LTS generation Next, we generate the full explicit labeled transition system. Once finished, we minimize the generated LTS using branching equivalence and hiding all but `elect` and `assert_false` gates.

Compositional LTS generation At last, we use the SVL scripting language for compositional verification [4]. The idea is to generate the LTS of subsystems, and to reduce them before connecting them together. When we connect two modules, we hide their communications ($\texttt{f}_i$`_request` and $\texttt{f}_i$`_response` gates). Reductions modulo tau-confluence (faster) or branching equivalence (smaller result) allow to remove most of the internal transitions corresponding to hidden gates.

We tried many scripts; our best SVL script computes successive prefixes of the module chain, according to the iteration rule below (the suffix `.bcg` indicates LTS stored as CADP binary-coded graphs):

```
"Source_Ti.bcg"=
 partial tau-confluence reduction of
 hide fi_request, fi_response in
   (scheduler|[elect,yield]|Ti)
  |[fi_request, fi_response]|
   "Source_Ti-1.bcg"
```

where $\texttt{Source\_T}_0$ is the source module, $\texttt{T}_i$, $1 \leq i \leq n$, is the i-th transmitter module, and $\texttt{T}_{n+1}$ is the sink module. This script is similar to node reduction (cf. Figure 1c) but the process `scheduler` is used to constrain the generation of each composite process $\texttt{T}_i$, thus reducing the composite LTS size. As shown in [2], this use of process `scheduler` as a context constraint is allowed since the process is deterministic, free of internal actions, and part of the context.

Table 1 presents the results ("nc" stands for "not completed" and denotes an operation that exceeded the memory or the tool capacity). All tests have been performed on a 2 GHz AMD opteron with access to 4 Go RAM (maximum addressable) running Linux. The "max. state number" entry gives the size of the largest LTS produced during compositional generation; "state number after minimization" is the size of the LTS minimized for the *branching* equivalence.

Globally, the results are nearly as good as the `goto` encoding of [18], whereas the presented encoding is more similar to their `normal` encoding (shared variable to force atomicity). Evaluation of the false property works well, even for n big, e.g. 10.71 s for $n = 20$. The compositional LTS generation with on-the-fly reductions reduces effectively the state number of the biggest intermediate LTS, and so the memory consumption. Compositional generation is slower than direct generation but it allows to generate the full LTS for bigger values of n. We confirmed experimentally with the *bisimulator* tool that the two techniques for LTS generation return equivalent LTS.

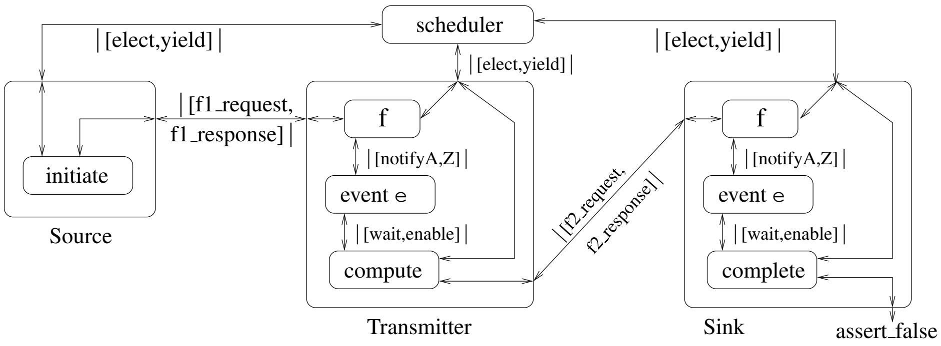

Figure 5. Nonpreemptive Lotos encoding of the `chain` benchmark for $n = 1$

Table 1. Results of the experiments with the nonpreemptive semantics

$n =$	3	7	11	15	17	19
false prop. eval.	1.88 s	2.13 s	3.02 s	4.99 s	6.44 s	8.2 s
true prop. eval.	1.8 s	7.28 s	118.78 s	2,443.77 s	nc	nc
direct LTS gen. state number	2.11 s 455	2.46 s 11,511	4.37 s 249,847	60.3 s 5,046,263	757.79 s 22,282,231	nc nc
compositional gen. max. state number	14.96 s 81	27.02 s 2,305	43.69 s 53,249	185.72 s 1,114,113	976.04 s 4,980,737	8,293.02 s 22,020,097
state number after minimization	48	768	12,288	196,608	786,432	3,145,728

4. Lotos encoding of concurrent TLM

TLM is not tied to SystemC and, in particular, to its simulation semantics based on a nonpreemptive scheduler. On the contrary, the definition of TLM in [6] refers to asynchronous concurrent processes running independently, except for otherwise explicitly defined synchronizations. Moreover, in practice, the nonpreemptive assumption of the SystemC simulation semantics may not hold for the real system hardware implementation and, thus, jeopardize all the verification effort invested in the SystemC/TLM model. Indeed, a nonpreemptive scheduler introduces implicit atomic sections hiding most of the issues regarding concurrent accesses to shared resources. Therefore, potential erroneous behaviors of the real system may not be revealed by SystemC simulation and formal methods complying with the SystemC simulation semantics.

As a consequence, verification of TLM models written in SystemC would benefit from the support of the concurrent semantics of TLM. To this end, it is needed to distinguish between SystemC as a description language for TLM models and SystemC as a simulation kernel. Such a distinction is drawn in [16] which describes a translation from TLM models written in SystemC into Lotos with respect to the concurrent semantics of TLM. In this section, we apply the principles of this translation to the encoding of the `chain` benchmark in Lotos.

4.1. Removal of the scheduler

Once the concept of scheduler removed from the Lotos model, concurrency between processes is only ruled by the semantics of the Lotos parallel operators, which coincides with the TLM semantics (i.e. concurrent execution of independent processes synchronizing explicitly).

SystemC events In the absence of a scheduler, we cannot reproduce the handling of events as done by the SystemC simulation kernel. For instance, a process has now no state "eligible": a process is either waiting for an event or executing. Hence, on notification of the waited event, a waiting process immediately resumes its execution. This is implemented by a rendez-vous on gate `resume` between the event handler and the waiting process, as illustrated in Figure 6. If several processes have to be resumed, the event handler resume them in a nondeterministic order. Moreover, since resumed processes may start interacting with other processes immediately, a notification should be nonblocking so that the execution of the notifying process can interleave freely with other processes without waiting for the last process to be resumed.

Several events can be handled by the same Lotos process, as in Figure 6. This requires one waiting list for each event and an additional offer on gates `notify` and `wait` to identify the involved event.

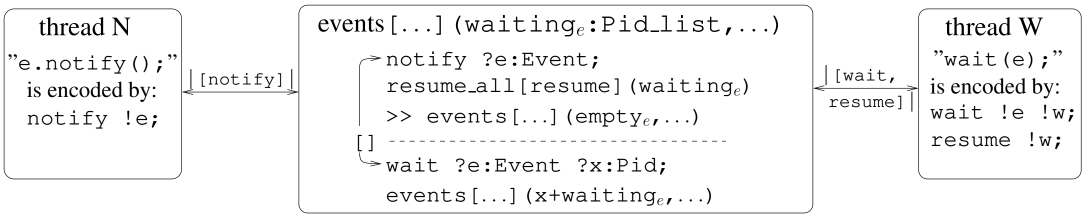

Figure 6. Event handling without scheduler

Locks In [16] was also introduced the idea of using locks to control asynchronism in the Lotos model. A lock allows to specify a resource that processes should access mutually exclusively. This resource could be the whole model, as with the SystemC scheduler, or, with a finer granularity so as to meet specific verification needs, resources could be entire modules or single transactions. The implementation of a lock handler is not very different from that of the scheduler in Section 3.1 which is a special case of a unique global lock acquired by each process before executing and released on wait or termination — this will be exemplified by the experiments in Section 5.1. The lock handler process proposes two rendez-vous: one for acquiring the lock, when a process needs to access to the corresponding resource; and a second one for releasing the lock, so that the resource can be accessed by other processes.

4.2. Transactions as process calls

We apply in this section the alternative translation for transactions described in [16]. This variant is not proper to the encoding without scheduler and could be applied to the previous one with scheduler.

The principle is that a process `F` encoding a transaction method of a target module is called by the initiator of the transaction instead of being encapsulated in the target and synchronized with the initiator. In the initiator, the call is followed by the sequence operator (>>) allowing to pass results to the remaining behavior of the initiator.

Compared to the encoding of Section 3, this avoids the duplication of `F` and of its gates according to the number of initiators, and it also replaces the two rendez-vous on gates `F_request` and `F_response` by one rendez-vous on the implicit termination gate when `F` terminates. As an optimization, most often, this rendez-vous on the termination gate can be removed by inlining in the initiator the code of `F` instead of calling `F`. However, weakening encapsulation may compromise the efficiency of the compositional approach which benefits from grouping in subcomponents processes synchronized together.

For instance, in the `chain` benchmark, the transaction method `void f()` is translated into a process, without parameter or return value, "`f[notify] : exit`", whose sole action is a rendez-vous on gate `notify`. The transaction itself, `port.f()` in SystemC, is translated into a call to process `f`, "`f[notify] >> ...`", in the initiator thread. As a result, the transaction is now part of the behavior of the initiator thread and does not appear anymore inside the target module, as shown in Figure 7. Besides, a single process `events` handles all the events of the system, as described in Section 4.1.

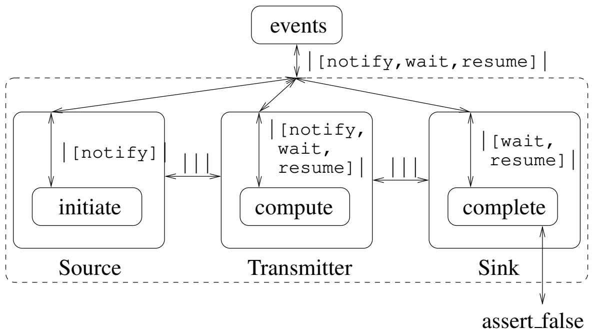

Figure 7. `chain` benchmark encoding in Lotos without scheduler for $n = 1$

4.3. Experiments

We have proceeded to the same experiments as with the nonpreemptive encoding (Section 3.3). The results are summarized in Table 2. The first group of results refers to the encoding of transactions by process calls, whereas in the other, transactions are inlined in the initiator. We observe a clear advantage for models with inlined transactions: size of LTS is greatly reduced, and hence evaluation of nontrivial properties is faster, or indeed possible for $n = 19$.

Compared to the encoding with scheduler, both encodings without scheduler produce smaller LTS by direct generation. This result is due on the one hand to the encoding itself (e.g. transaction translation), and on the other hand to the `chain` benchmark. Indeed, in this benchmark, each thread does only one action in the atomic sections delimited by the scheduler, as a consequence, removing these atomic sections as done by the encoding without scheduler does

Table 2. Results for the encoding without scheduler

	$n =$	3	7	11	15	17	19
called transactions	*false prop. eval.*	1.45 s	1.56 s	1.79 s	2.03 s	2.18 s	2.28 s
	true prop. eval.	1.58 s	5.69 s	101.73 s	2,419.16 s	17,165.4 s	nc
	direct LTS gen.	1.59 s	1.7 s	3.02 s	40.09 s	392.24 s	nc
	state number	288	8,704	204,800	4,325,376	19,398,656	nc
	compositional gen.	12.41 s	19.4 s	28.25 s	83.83 s	340.81 s	2,601.64 s
	max. state number	77	1,281	28,673	589,825	2,621,441	11,534,337
	state number after minimization	47	767	12,287	196,607	786,431	3,145,727
inlined transactions	*false prop. eval.*	1.35 s	1.45 s	1.71 s	1.9 s	2.12 s	2.17 s
	true prop. eval.	1.4 s	2.08 s	11.71 s	166.73 s	686.93 s	3,201.37 s
	direct LTS gen.	1.55 s	1.57 s	1.98 s	4.4 s	15.34 s	89.63 s
	state number	77	1,277	20,477	327,677	1,310,717	5,242,877

not involve a model with more behaviors and should not produce, all things being equal, bigger LTS.

Evaluation of the properties was done on-the-fly and with generation of a diagnostic explaining the truth value of the properties. Times for true property evaluation are increased by asking for a diagnostic (we observed up to more than 30 times): indeed, in this case, the diagnostic is the full LTS of the model. On-the-fly evaluation proved to be faster than verifying the property on the whole LTS, when taking into account the time to generate the LTS. Above all, on-the-fly evaluation still enabled verification of false properties when generation of the LTS had failed.

For compositional generation, we applied the SVL predefined leaf reduction strategy, and then minimized the result for the branching equivalence. We chose to hide all rendez-vous except those corresponding to waits, event notifications, and the observed gate `assert_false`. The SVL script for a behavior B is then:

```
"B.bcg" = branching reduction of
 leaf
 partial tau-confluence reduction of
  hide all but
   assert_false, notify, wait in B
```

The time reported for compositional generation includes the branching minimization too. Compositional generation with the same strategy gives exactly the same results for models with inlined transactions as for models with called transactions. With respect to direct generation, composition leads to smaller LTS with called transactions but not with inlined transactions. As this example shows, an effective compositional approach requires a good expertise to devise a strategy fitting a specific Lotos model otherwise than by trial and error. As regards execution time, compositional generation is slower than direct generation, but the huge size of the LTS can outweigh this better performance of direct generation, e.g. for $n = 17$.

5. Comparison of both encodings

In this section, we compare the encodings with and without scheduler we have presented. The first qualitative comparison aims at showing which encoding is the most meaningful and faithful for a verification task. The second comparison shows how the encoding influences the time needed by the verification tools.

In order to compare the Lotos encodings of the `chain` benchmark, we need to define the common actions that will be observed in all the models and hide all other actions specific to one of them. For instance, gates `elect` and `yield` appear only in the nonpreemptive encoding and have no direct equivalent in the encoding without scheduler. To this end, we add to each thread, just before termination, a rendez-vous on gate `output !pid`. In addition to provide a common observable action, it introduces in the threads of the `chain` benchmark a second action, after the transaction, which leads to different system behaviors according as the threads are interleaved or executed as atomic sections.

We developed three models of the benchmark with outputs. The first model, M_s, is based on the encoding with scheduler; the second one, M_c, is based on the encoding without scheduler where transactions are inlined; the last one, M_l, modifies M_c by adding a global lock that is acquired and released so as to reproduce the behavior of the SystemC scheduler (mutual exclusion of thread executions).

5.1. Inclusion and equivalence of models

Intuitively, an execution with the nonpreemptive semantics is also possible with the concurrent semantics. We use the *bisimulator* tool of CADP to verify this hypothesis. We have shown that, for the branching equivalence, the encoding with scheduler is included in the encoding without scheduler, and not vice versa. This means that, for the `chain` with outputs benchmark TLM model, the behaviors exhibited with the nonpreemptive semantics is a strict subset of the behaviors exhibited by the concurrent semantics.

However, if we introduce a global lock in the encoding without scheduler, gates acquiring and releasing the lock exactly correspond to gates `elect` and `yield` in the encoding with scheduler, as confirmed by the branching equivalence result between the encoding with scheduler and the encoding without scheduler but with a global lock. This result could be shown for the branching equivalence but not for the strong equivalence. This can be explained by the hidden gates only present in the nonpreemptive encoding (e.g. `F_request` and `F_response`) and by the enforced atomicity between a notification and a resume in the nonpreemptive encoding.

Table 3 gives the execution times for the comparison of the LTS obtained for the `chain` with outputs benchmark, when all gates except `output` are hidden, from the Lotos model with scheduler (M_s), without scheduler (M_c), and without scheduler but with a global lock (M_l). The comparison was done on the minimized LTS without generating a diagnostic.

Table 3. Model comparison execution times

$n =$	3	7	11	15
$\mathrm{M}_s \lesssim_{\text{branching}} \mathrm{M}_c$	0.4 s	0.9 s	nc	nc
$\mathrm{M}_s \not\gtrsim_{\text{branching}} \mathrm{M}_c$	0.7 s	6 s	350 s	nc
$\mathrm{M}_s \simeq_{\text{branching}} \mathrm{M}_l$	0.6 s	0.7 s	18.2 s	27,557.8 s

5.2. Influence of encodings on scalability

For each model, Table 4 exposes the size and generation time of the LTS for the direct and compositional approaches when hiding all gates except `output`.

As expected on the benchmark with outputs, the concurrent semantics of M_c exhibits more behaviors than the nonpreemptive one of M_s: this explains the bigger minimized LTS of M_c. However, the better encoding of M_c leads to smaller LTS by direct generation.

In M_l, the addition of the global lock generates bigger LTS than M_c, up to the point this is outweighed by the supplementary behaviors of M_c. But more importantly, direct generation for M_s and M_l produce LTS similar in size.

This means that the encoding without scheduler allows to show more realistic behaviors of the model than the encoding with scheduler, and to reproduce this latter encoding as a particular case, without being more costly as regards direct generation of LTS. Indeed, for $n = 19$, we could produce a LTS only for M_l, not for M_s (because this encoding requires more bytes to store each state).

Compositional generation of M_s seems more effective than that of M_c thanks to a better encapsulation of event and transaction mechanisms (cf. Section 4.2), e.g. it allows to obtain the minimized LTS with $n = 17$ for M_s, but not for M_c. Nevertheless, this has to be moderated by the greater intrinsic complexity of M_c, and by the state number ratio between the minimized LTS and the largest LTS produced during composition, which is close for the two models and not always in favor of M_s, e.g. for $n = 15$. For M_c and M_l, the chosen compositional approach reduces the number of states in the same proportion: the largest composed LTS is around 10% smaller than the directly generated LTS.

6. Conclusion and further work

In this paper, we used the benchmark proposed in [18] to compare two semantics of TLM models written in SystemC. The first semantics is based on the nonpreemptive simulation semantics of SystemC, whereas the second one refers to the concurrent semantics of TLM. We contributed the encoding in Lotos of the nonpreemptive semantics, we proposed inlined transactions as an optimization of the existing concurrent semantics encoding, we defined a variant of the benchmark, and we translated the two versions of the benchmark with each encoding.

Then, we formally showed on the example of the benchmark that the nonpreemptive semantics of SystemC was strictly included in the concurrent one of TLM. We also showed that the concurrent semantics generalize the nonpreemptive one as the former is a particular case of the latter where a global lock is used. Remarkably, this generalization came with a better performance of the CADP tools, even on very large LTS (near 100 million states).

Thus, our encoding of the concurrent semantics of TLM, associated with locks, provides both flexibility in specifying the level of asynchrony between processes desired for verification, and performance since, on the benchmarks, this encoding scales as well, if not better than other encodings yet restricted to the behaviors covered by the SystemC simulation semantics.

This work also shows that CADP and Lotos provide an adequate setting for experimenting with concurrent language semantics. One major benefit of CADP is the numerous integrated tools to generate and handle even very large LTS. Especially, the bisimulator tool is crucial to formally compare models with different semantics. We experimentally verified the effectiveness of the compositional and on-the-fly approaches with SVL and CADP. The former allowed to generate an LTS when direct generation had failed, and the latter to verify properties on models for which generation of an explicit LTS was not possible at all. Compared to the performances obtained with SPIN in [18], CADP seems to scale slightly better when we take advantage of the compositional approach.

Locks combined with the concurrent semantics allow to

Table 4. Results for the LTS generation of the three models

	$n =$	3	7	11	15	17	19
M_s	*direct LTS gen.*	2.24 s	2.68 s	4.57 s	74.62 s	1359.54 s	nc
	state number	485	12,021	258,037	5,177,333	22,806,517	nc
	compositional gen.	16.02 s	29.27 s	49.19 s	291.95 s	2,237.06 s	nc
	max. state number	96	2,560	57,344	1,179,648	5,242,880	nc
	state number after minimization	47	767	12,287	196,607	786,431	nc
M_c	*direct LTS gen.*	1.6 s	1.74 s	3.27 s	50.61 s	445.51 s	nc
	state number	288	8,704	204,800	4,325,376	19,398,656	nc
	compositional gen.	12.07 s	19.09 s	146.93 s	75,741 s	nc	nc
	max. state number	256	7,680	180,224	3,801,088	nc	nc
	state number after minimization	64	1,536	32,768	655,360	nc	nc
M_l	*direct LTS gen.*	1.48 s	1.71 s	2.48 s	37.21 s	451.16 s	8,552.94 s
	state number	412	11,260	253,948	5,242,876	23,330,812	102,760,444
	compositional gen.	11.27 s	17.41 s	29.74 s	275.19 s	1,784.72 s	16,464.3 s
	max. state number	365	9,981	225,277	4,653,053	20,709,373	91,226,109
	state number after minimization	47	767	12,287	196,607	786,431	3,145,727

explore the trade-off between performance and exhaustiveness of the verification. We are currently investigating this approach on a real industrial case study modeled in SystemC and involving more than 20,000 lines of C and C++ code.

References

[1] D. Cansell, D. Méry, and C. Proch. Modelling SystemC scheduler by refinement. In *ISOLA*, Sept. 2005.

[2] S. C. Cheung and J. Kramer. Context constraints for compositional reachability analysis. *ACM Transactions on Software Engineering and Methodology*, 5(4):334–377, 1996.

[3] R. Drechsler and D. Große. Reachability analysis for formal verification of systemc. In *DSD*, pages 337–340. IEEE, Sept. 2002.

[4] H. Garavel and F. Lang. SVL: a scripting language for compositional verification. In *FORTE*, pages 377–392, Aug. 2001.

[5] H. Garavel, F. Lang, R. Mateescu, and W. Serwe. CADP 2006: A toolbox for the construction and analysis of distributed processes. In *CAV*, volume 4590 of *LNCS*, pages 158–163, July 2007.

[6] F. Ghenassia, editor. *Transaction-Level Modeling with SystemC. TLM Concepts and Applications for Embedded Systems*. Springer, June 2005.

[7] D. Große and R. Drechsler. CheckSyC: an efficient property checker for RTL SystemC designs. In *ISCAS*, volume 4, pages 4167–4170, May 2005.

[8] C. Helmstetter. *Validating Models of Systems-on-a-Chip in the Presence of Nondeterministic Schedulings and Loose Timings*. PhD thesis, INPG, 2007.

[9] C. Helmstetter, F. Maraninchi, and L. Maillet-Contoz. Test coverage for loose timing annotations. In *FMICS*, pages 100–115, Aug. 2006.

[10] ISO-8807. Lotos, a formal description technique based on the temporal ordering of observational behaviour, 1989.

[11] D. Karlsson, P. Eles, and Z. Peng. Formal verification of SystemC designs using a petri-net based representation. In *DATE*, pages 1228–1233, Mar. 2006.

[12] D. Kroening and N. Sharygina. Formal verification of SystemC by automatic hardware/software partitioning. In *MEMOCODE*, pages 101–110. IEEE, July 2005.

[13] 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, Sept. 2005.

[14] B. Niemann and C. Haubelt. Formalizing TLM with communicating state machines. In *FDL*, pages 285–292, Sept. 2006.

[15] Open SystemC Initiative. *SystemC v2.1 Language Reference Manual (IEEE Std 1666-2005)*, 2005.

[16] O. Ponsini and W. Serwe. A schedulerless semantics of TLM models written in SystemC via translation into LOTOS. In *FM*, volume 5014 of *LNCS*, May 2008.

[17] OSCI SystemC TLM 2.0, draft 2 for public review, 2007.

[18] C. Traulsen, J. Cornet, M. Moy, and F. Maraninchi. A SystemC/TLM semantics in Promela and its possible applications. In *SPIN Workshop*, pages 204–222, July 2007.

[19] M. Y. Vardi. Formal techniques for SystemC verification. In *DAC*, pages 188–192, June 2007.

[20] P. Wodey, G. Camarroque, R. Hersemeule, and J.-P. Cousin. LOTOS code generation for model checking of STBus based SoC: the STBus interconnect. In *MEMOCODE*, pages 204–213. IEEE, June 2003.