LusSy: A toolbox for the analysis of systems-on-a-chip at the transactional level

Matthieu Moy, Florence Maraninchi, Laurent Maillet-Contoz

LusSy: A Toolbox for the Analysis of Systems-on-a-Chip at the Transactional Level

M. Moy
STMicroelectronics, Verimag
Matthieu.Moy@imag.fr

F. Maraninchi
Verimag, Centre équation - 2, avenue de Vignate, 38610 GIÈRES — France
Florence.Maraninchi@imag.fr

L. Maillet-Contoz
STMicroelectronics, HPC,
System Platform Group.
850 rue Jean Monnet, 38920 CROLLES — France
Laurent.Maillet-Contoz@st.com

Abstract

We describe a toolbox for the analysis of Systems-on-a-chip described in SystemC at the transactional level. The tools are able to extract information from SystemC code, and to build a set of parallel automata that capture the semantics of a SystemC design, including the transaction-level specific constructs. As far as we know, this provides the first executable formal semantics of SystemC. Being implemented as a traditional compiler front-end, it is able to deal with general SystemC designs. The intermediate representation is now connected to existing formal verification tools via appropriate encodings. The toolbox is open and other tools will be used in the future.

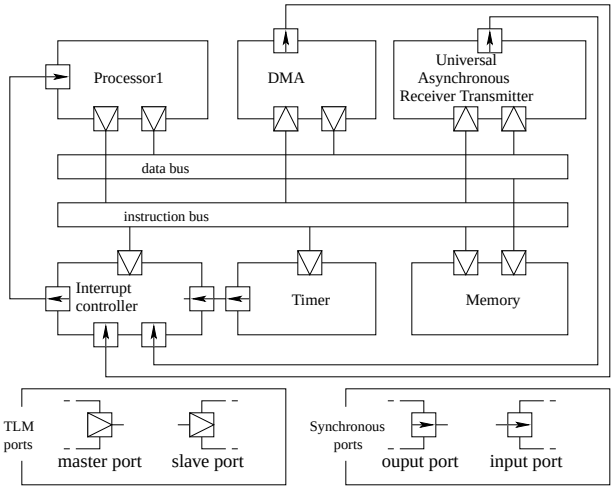


Figure 1. An Example TLM design

1 Introduction

Quality and productivity constraints in the development tools for the design of "systems-on-a-chip" are increasing quickly. Reliability becomes a real issue, as the complexity of designs grows. Verification is known to be the main step of chip design, in terms of manpower.

The Register Transfer Level (RTL) used to be the entry point of the design flow, but the simulation environments for such models do not scale up well. Developing and debugging embedded software for these low level models before getting the physical chip from the factory is no longer possible with a reasonable cost. A new abstraction level, the *Transaction Level Model (TLM)*, is emerging. It uses a component-based approach, in which hardware blocks are modules communicating with so-called *transactions*. This level of abstraction is quite far from the cycle-accurate RTL. The TLM models are used for early development of the embedded software, because the high level of abstraction allows a fast simulation (around 1000 times faster than pure RTL models, and still much faster than cosimulation).

Just to get the flavor of what a TLM model is, observe Figure 1. It shows a simplified view of the typical architecture of a system-on-a-chip, namely the ARM's PrimeXsys wireless platform. The architecture is made of components and several kinds of connections. The main master module is a processor. High level communication is done through the data and instructions buses, and interrupts are managed through synchronous signals. A *transaction* from A to B is the encapsulation of a potentially complex data structure, being transmitted according to a complex protocol that may involve several low-level information transfers in both directions in RTL, but modeled by simple function calls in

TLM. This kind of design is mainly *asynchronous*, in the sense that designers do not rely on a global time scale: a component is developed without knowing whether the other components will have the same clock, and that is why synchronization between them involves general protocols.

As TLM models appear first in the design flow, they become *de facto* reference models for systems-on-a-chip (SoCs). This raises several problems: 1) What does it mean to validate properties at the TLM level? (including the problem that SystemC or similar languages do not have a formal semantics; by validation we mean either simulation or formal verification). 2) Since automatic synthesis from TLM to RTL does not exist (and will not exist soon), how can we *compare* a TLM reference design and a RTL design that is supposed to implement it, and is partly written by hand? 3) How can we express and validate non-functional properties of SoCs at the TLM level?

These questions are key points for the wide adoption of TLM models in the industry, and are being addressed by a joint project between Verimag and the SPG team of STMicroelectronics. In this paper, we report on the work done for addressing the first item: how to give a formal semantics to SystemC and then express and verify properties of a TLM design written in SystemC [14]? We comment on the other points in the conclusion.

Related work

SystemC designs being *circuit* designs, we could think of using one of the verification tools (model-checkers, SAT-solvers, etc.) developed for hardware verification, for instance SMV [13]. However, these tools are tailored for the RTL, exhibiting a clear notion of logical time, while we need to deal with heterogeneous designs. Heterogeneity comes from several places: determinism and non-determinism, synchronous and asynchronous systems, hardware and software components. Moreover, these tools cannot take general SystemC as input.

As far as we know, all the work on verification techniques and tools for SystemC designs are limited to the subset of SystemC that allows to write RTL designs. It cannot be used for real TLM designs (See [5] for instance). [15] treats the SpecC language (similar to SystemC). A date of execution is associated with each instruction of the program, which can then be considered as a dependency graph, on which synchronization properties can be proved. The approach is very limited: since only one date can be associated with an instruction, this does not allow to consider general loops.

Now, since SystemC is mainly a C++ library, one could think that we have to face the same problems as those addressed by general-purpose software model-checking tools. This is not the case. Verifying SystemC designs is, on the one hand *simpler*, because we do not have to deal with general dynamic data structures and general algorithmics; on the other hand *harder*, because we have to take parallelism into account, and to know about the scheduler specification. General software model-checking techniques concentrate on dynamic data structures and general algorithms. They provide sophisticated tools like invariant extraction, loop unrolling, etc., but are not directly usable to exploit the particularities of the SystemC constructs provided as a C++ library. Using these tools for SystemC would need to include the non-deterministic scheduler specification in the tool. Moreover, they usually do not take parallelism into account. For instance, CBMC [3, 4] can apply bounded model-checking techniques on pure C models, but does not deal with parallelism, or with infinite loops. SLAM [1] uses clever abstractions and refinement techniques, but also focuses on sequential programs. VeriSoft [6] can handle parallel processes written in any language. They are executed as black boxes, communicating via calls to operating system primitives. These calls are intercepted to build a model of their parallel behaviour. We cannot exploit such a black-box approach, because we need to extract the transaction-level specific constructs of SystemC, and aim at treating addresses in a specific way (see below).

The closest related work is to be found in Java model-checking, since they take a scheduler specification into account. The first version of the Java Path Finder model-checker [8] used an approach similar to ours, translating Java into the intermediate representation Promela, and using the model checker SPIN to prove the properties. Version 2 [9] checks the byte-code directly, using a dedicated JVM with backtracking capabilities, and lots of other model-checking techniques. However, the techniques dedicated to Java are not directly applicable, neither to SystemC and its scheduler, nor to the modeling of synchronous and asynchronous mechanisms.

Approach and Contributions

We advocate an approach able to exploit all the particularities of a TLM design written in general SystemC. The idea is not to express the TLM concepts manually in yet-another-formalism that can be exploited by verification tools, but to be able to take real SystemC designs into account. We describe a method implemented in a new dedicated tool called LUSSY: based on compiler front-end techniques, it is able to extract architecture and synchronization information from a TLM design written in SystemC with very few abstractions, by exploiting carefully the constructs provided by the library. It builds its own intermediate representation called HPIOM (for *Heterogeneous Parallel Input/Output Machines*) made of communicating parallel machines, able to represent both deterministic and non-deterministic components, synchronous and asynchronous communication protocols, Boolean and numerical data. For

the moment LUSSY connects this intermediate representation to the symbolic model-checker LESAR [7] and to the abstract-interpretation tool NBAC [10]. Both tools provide conservative automatic verification results for safety properties, and may perform their own abstractions on the HPIOM representation, when needed. The current state of the LUSSY implementation is being applied to case-studies provided by STMicroelectronics; it accepts a large subset of SystemC.

Translating SystemC into HPIOM is a way of giving a formal semantics to SystemC. The faithfulness of the translation relies on the executability of HPIOM. The HPIOM obtained may be tested against the official SystemC execution engine (we currently use the LUSTREback-end to execute HPIOM—It allows either user-friendly debugging or efficient compilation). LUSSY is an open tool, allowing other tools (SAT solvers, model-checkers, ...) to be experimented on HPIOM obtained from SystemC. Finally, studying verification methods and tools for TLM designs written in SystemC gives hints on how TLM models should be written to allow easier use of verification tools. This helps in defining libraries at the appropriate level of abstraction, and general guidelines for designers.

The contributions of this paper are: 1) an executable formal semantics for TLM models written in full SystemC, with an operational translation tool; 2) a way of expressing safety properties directly in SystemC; 3) a working connection to verification tools.

Syntactically speaking, LUSSY accepts a very large subset of SystemC, being based on a full C++ front-end; the only limitation is that the use of templates is restricted to a fixed set of parameter types, for which expansion can be performed. The other restrictions are of semantic nature: the SystemC code is accepted by LUSSY, but the semantics is made abstract because the target formalism is less expressive than a general-purpose language. This occurs for all pieces of code that deal with dynamic data structures. We introduce a specific translation for *addresses*, to help identifying the influence of these data on the control. Finally, we deliberately introduce non-determinism in the translation, to reflect the non-determinism of the SystemC specification (choice of the scheduler, uninitialized variables, etc.).

Section 2 explains the TLM level, shows how TLM designs can be written in SystemC, where we need verification, and how a dedicated approach can benefit from the particularities of SystemC TLM designs. We define the intermediate representation HPIOM in section 3, and the semantics of SystemC in HPIOM in section 4. We present the tool LUSSY in section 5, and its use on a simple example in section 6. Section 7 is the conclusion.

2 The TLM level and SystemC

2.1 SystemC constructs and TLM TAC support

A TLM design written in SystemC is based on an *architecture*, i.e. a set of components and connections between them (see Figure 1). Each component has typed connection *ports*, and its behavior is given by a set of communicating *processes* that can be programmed in full C++. SystemC provides a scheduler, and several synchronization mechanisms: The low-level *events*, the synchronous *signals* that trigger an event when their value changes, and higher level, user-defined mechanisms based on abstract communication channels. The architecture may be hierarchical, but this has no influence on the semantics of the model.

A SystemC design that does not use the high level synchronization mechanisms cannot really be considered as a TLM design. Using these high level synchronization mechanisms requires that the abstract classes provided by SystemC be implemented. The SPG team of STMicroelectronics proposed the TAC implementation ("*Transaction Accurate Communication Channel*") as a support for TLM design in SystemC. This proposal is intended to become a standard, and has been donated to OSCI (the "*Open SystemC Initiative*", gathering major actors of the domain). The TAC channels provide transaction serialization (with the `tac_seq` component), channel locking, and even an arbitration policy (in the `tac_arbiter`). In the example of Figure 1, both the data bus and the instruction bus are described using TAC channels.

2.2 Execution and verification

Static and dynamic aspects: the architecture is built by executing the so-called *elaboration phase* of the SystemC program, which performs dynamic object allocations, for all components and connections. Then the set of components is known for all the execution time. The scheduler starts running the processes of the components, according to the algorithm of Figure 2. Initially, all *processes* are eligible. Processes are ran one by one, non-preemptively, and explicitly suspend themselves when reaching a waiting instruction. There are two kinds of waiting instructions: a process may wait for some time to elapse, or for an event to occur. While running, it may send events, or write on signals. These actions are stored in the data structure F, until the end of the so-called *evaluation phase*; then they are taken into account one by one: an event awakes the processes that were waiting for it; a signal write is stored in V for the next reads. When there is no more eligible process at the end of an update phase, the scheduler lets time elapse, awaking the processes that have the earliest deadline. (Note that this algorithm gives only the simpler cases. Uninitialized processes, instantaneous notification for example would add particular cases.)

As far as verification is concerned, the architecture being *static* is a crucial point: the set of processes is known once and for all, and the topology of the connections does not change. When we want to prove properties about the behaviors of SystemC designs, we have to deal with: a fixed set of processes, a fixed architecture, a fixed set of signals and events exchanged, and the scheduler.

Synchronization code vs. complex algorithms: a typical TLM design exhibits a clear distinction between the potentially complex algorithmics of some components, and the code dedicated to synchronization. For instance, a processor may be included in the design, with SystemC code describing the interpreter of its machine language. In this case, the code intented to be run on the processor is provided separately.

In this paper, we will focus on *safety functional synchronization properties* of TLM models (*safety* as opposed to *liveness* [11] and *functional* as opposed to *performance*). If a processor is present in the design, this means treating it in a very abstract way. The program it runs might be checked by other techniques (software model-checking or theorem-proving); the processor component may then be replaced by very simple SystemC describing how it is connected to the other components, and abstracting all its behaviour. The properties that can be checked on such an abstracted TLM design cannot depend on the details of the algorithms run by the processor, but this is good design practise, anyway.

2.3 Expressing properties

Generic properties do not require the use of a specification language. In LUSSY we can express and check the following:

— Check that a global dead-lock never occurs. We consider that a global dead-lock occurs when SystemC scheduler enters the "time elapse" phase while no process is waiting for time. In the example of figure 1, this might happen if the processor is waiting for an interrupt, if the other processes are waiting for transactions to process.

— Check that a process never finishes. This should always be the case except for test benches.

— Check that a synchronous signal is never written on twice during the same δ-cycle. This is a dangerous situation since the final value on the signal will depend on the order of execution, which is most probably dependent on the scheduling policy. In the example, if one process of the interrupt controller raises the interrupt signal while another cancels it, there is a data race.

In order to specify and prove user-defined properties of SystemC designs, we need a specification formalism. The idea in LUSSY is that the user should not have to learn a timed logic language. The property should be written in the same language as the implementation. We may check that some portions of code are mutually exclusive. This is slightly intrusive in the source code since the beginnings and ends of the critical sections have to be specified. Finally the most general safety properties are expressed by assertions in the source code: `ASSERT(condition)`. Technically, the ASSERT macro is defined by:

```
#define ASSERT(X)
       if(!(X)){is_this_reachable();}
```

so the problem of assertion verification is reduced to the problem of code reachability.

```
E: the set of eligible processes
S: a set of tuples (P, e) of sleeping processes
   waiting for events
T: a set of tuples (P, t) of processes
   associated with time events
F: a set of event or signal consequences:
   (EV, e) or (SIG, s, v)
V: a set of tuples (s, v) for signal values
E := { all processes }
loop until the end of simulation
   // one execution of the following loop body is a δ-cycle
   while E ≠ ∅
      // the so-called evaluation phase:
      while E ≠ ∅
         P := one element in E ; E := E − { P }
         run P, while filling F and reading signal values in V,
                        until it stops:
            if P emits an event e: F := F ∪ { (EV, e) }
            if P writes a value v on a signal s:
               F := F − { (s, ...) } ∪ { (s, v) }
         if P stopped on a wait-time (t)
            T := T ∪ { (P, t) }
         if P stopped on a wait-event
            S := S ∪ { (P, e) }
      end
      // the so-called update phase:
      For each element f in F
         if f = (EV, e) then
            for each (P, e) in S :
               E := E ∪ { P }; S := S − { (P, e) }
         if f = (SIG, s, v) then
            V := V − { (s, ...) } ∪ { (s, v) }
      endend
   // let time elapse:
   min = minimum value of the t_i's in the set T = { (P_i, t_i) }
   for each element x=(P_k, min) in T
      T := T - { x } ; E := E ∪ { P_k }
   end
end loop
```

Figure 2. The SystemC scheduler algorithm.

2.4 Didactic example of a TLM design

To illustrate the transformation from SystemC to HPIOM, let us introduce a minimal example (see figure 3). For clarity, we only show the body of the processes, and the meth-

ods called to process transactions in the slave modules. The system contains two master modules and two slave modules. They are connected through a `tac_seq` channel. The program contains *assertions* for the properties we want to verify.

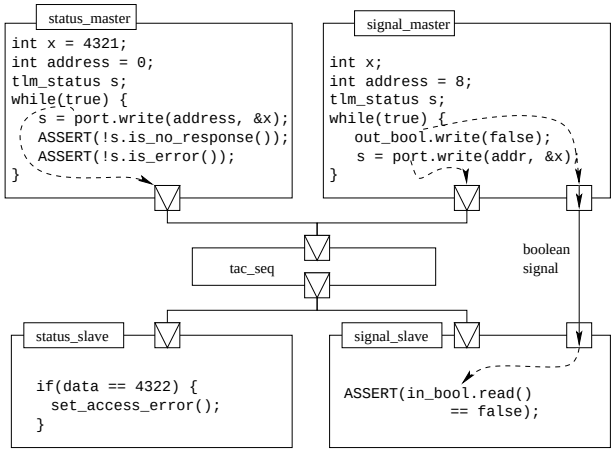


Figure 3. An example transactional system

The target port of the module `status_slave` (resp. `signal_slave`) is mapped at the address 0 (resp. 8): It will receive the transactions initiated by `status_master` (resp. `signal_master`). The call to the method `write` on an initiator port searches for a slave module mapped at the corresponding address, and calls the `WriteAccess` method on it. If no module is mapped at the address written on, then nothing happens, and the status returned verifies `status.is_no_response()`. If a module is mapped at this address, the status returned verifies `status.is_ok()` unless the method `set_access_error()` has been called during the `WriteAccess` call.

In this example, the behavior highly depends on the value of variables representing addresses (the action triggered by the write will be totally different depending on the address on which we write), and may also depend on data (because of the `if` statement) (but there is no complex algorithmics).

3 HPIOM: Heterogeneous Parallel Input/Output Machines

A basic automaton $\mathcal{A}$ is a reactive machine made of a set of *control points*, a set of labeled transitions between control points, and a set of variables V. A *state* of such an automaton is made of a control point and a valuation of the variables. Automata communicate through messages. Transition labels are made of:
— a guard, which is a Boolean expression made of elementary tests on the variables of V (eg: $[x < 3]$) and tests on the presence of a message (eg: ?*message*).
— a list of messages emitted, denoted by !*message*,
— a set of parallel assignments denoted by $v := e$ where $v \in V$ and e is an expression on V.

The messages that appear in the conditions of the transitions (resp. the list of emitted messages) are called *inputs* (resp. *outputs*). Such an automaton reacts to a sequence of inputs by emitting a sequence of outputs, and modifying its internal state. If there is a transition from cp_1 to cp_2 labeled by g_1/e_1a_1 (a guard, a set of emitted messages, a set of assignments) in the automaton, then from a state (cp_1, v_1), provided the condition g_1 is satisfied by the valuation v_1, the automaton can reach a state (cp_2, v_2) where v_2 is obtained from v_1 by executing a_1, emitting the messages in e_1.

All the basic automata are *reactive*: from any state, with any input configuration, the transition of the automata is defined. However, in the concrete syntax, we usually omit self-loops. All the automata are also *deterministic*: from any state, for any input configuration, there is a most one possible transition.

The automata are composed in parallel using the *synchronous product*: a step of the global system involves exactly one step in each of the parallel automata, and is obtained by performing the conjunction of the guards, the union of the emitted messages, and the union of the assignment sets (provided they do not intersect). This product does not perform any synchronization between the parallel components. When two automata have to communicate, they share a message `x` which is an input on one side, and an output on the other side. In their product, `x` is both an input and an output. Then an encapsulation operator is used to force synchronization through `x`, and hide it. It expresses the semantics of the synchronous broadcast of signals in all synchronous languages, on which the reader can find more details in [12]. The intuitive behavior is the following: when one of the parallel components (or several at the same time) sends a message `x`, all the other components that are in the scope of the encapsulation operator for `x` take the corresponding transition (since they are reactive, this transition always exists). This reaction is *synchronous*, meaning that there is only one transition in the resulting automaton. The synchronous broadcast used here is known to raise so-called "causality" problems [2]. The HPIOM models obtained with our translation scheme are free of this kind of problems.

As usual in this kind of formalism, non-determinism is modeled by additional inputs called *oracles*. For instance, a non-deterministic situation that would imply two transitions $(cp, g/e_1a_1, cp_1)$ and $(cp, g/e_2a_2, cp_2)$ is in fact written as $(cp, g \wedge ?i/e_1a_1, cp_1)$ and $(cp, g \wedge \neg ?i/e_2a_2, cp_2)$ where i is an additional input message. This mechanism is used whenever a SystemC design exhibits non-determinism: The initial values of uninitialized signals, the choice of a process to run by the scheduler, etc.

4 Semantics of SystemC into HPIOM

4.1 Principles

The translation into HPIOM does not perform more abstractions than those implied by the expressivity of HPIOM compared to that of SystemC (see section 4.2). Since most interesting properties are undecidable on HPIOM, further abstractions will have to be made, but we let them to specific verification tools connected to HPIOM.

On the other hand, we could translate SystemC processes taking the scheduler and the synchronization primitives into account, but not the TLM constructs, which would then be treated as ordinary C++ code. This would lead us to lose interesting information about the structure and behavior of the design. We have chosen to take TLM constructs into account during the translation, which means giving a direct HPIOM semantics to TLM constructs.

Of course, since SystemC has no formal semantics, a formal proof of the equivalence between a SystemC source file and the corresponding HPIOM representation built by LUSSY is impossible. HPIOM being executable means executions can be compared, but it is also of great importance to give a semantics to SystemC into HPIOM in a simple, well structured and clearly decomposed manner, which we describe here. This leaves room for optimizations.

The main idea is the following: 1) each process in SystemC will be associated with one automaton in HPIOM; 2) the complete HPIOM description of a SystemC design will be made of all these "process" automata, plus specific automata for SystemC and TLM constructs.

The automata representing the body of the processes are extracted from the information obtained with the C++ front-end. Each process gives an automaton representing its control structure. For the SystemC library structures, the method is different: we never parse the SystemC library source code itself. We describe HPIOM patterns, based on the SystemC library specifications: there is an automaton pattern for the scheduler, one for each signal, etc. To generate instances of these patterns, we need to extract additional information from the SystemC design: for the scheduler we need the number of processes in the system, for channels we need the number of connected modules, etc.

4.2 Expressivity of HPIOM and abstractions

HPIOM may be used to encode any statically bounded-memory program. In SystemC, static bounds are guaranteed if: 1) the program does not perform dynamic memory allocation; 2) there are no calls to recursive functions.

The semantics of SystemC into HPIOM abstracts memory allocation primitives and recursive function calls into new input messages with unknown value. We also do that for not yet implemented constructs of SystemC, to get a working connection to verification tools before full SystemC has been taken into account by the front-end. These abstractions are clearly conservative for safety properties: the set of behaviours a SystemC code may exhibit when considering two complex expressions, is a superset of the set of behaviours it can exhibit when considering the detail of these expressions.

Another abstraction (which is optional) is related to the way addresses are dealt with. In SystemC, addresses are simply `int` values. If nothing special is done in the translation, addresses become ordinary variables in HPIOM, and any property related to addresses has to be transmitted to a verification tool able to deal with `int`s. However, in the SystemC source code, it is possible to distinguish adresses from other `int`s. For addresses, we propose an encoding based upon the existence of *address maps*. Indeed, in SystemC, the significant values of the address variables are given by the address maps used to describe the connection between components. Such a map is a partition of N into a finite number of *ranges* R_1, ..., R_n. With each R_i, we associate a Boolean variable b_i. An address variable x is then encoded by a valuation of the vector b_1 ... b_n. A constant value $k \in R_i$ is encoded into $b_i = 1, b_{j \neq i} = 0$. As soon as we manipulate addresses, we may lose information, resulting in encodings where $\exists i \neq j \; . \; b_i = b_j = 1$, meaning the value of x is in range R_i *or* in range R_j. This is conservative for safety properties. It simplifies the proofs a lot, and has proved to be sufficient on the examples we tried (the computation time for the proof fell down from several hours to less than a second on the example platform).

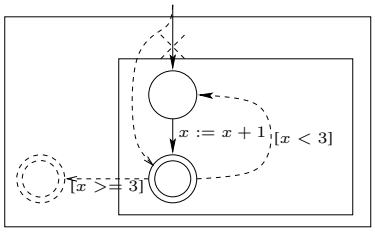
The last abstraction (which is also optional) is related to asynchrony. SystemC is intended to model and simulate *asynchronous* components. Although it provides a construct `wait (t)` where `t` is an amount of time, guidelines specify that this quantitative time `t` should not be used to enforce synchronization (i.e., the designer should not assume that two processes that perform the same `wait (t)` will synchronize when `t` has elapsed). The "time-elapse" phase of the scheduler algorithm (Figure 2) awakes the processes in the order specified by the `wait` parameters. In our translation, they are woken-up non-deterministically (encoding non-determinism with oracles). This means that the HPIOM model has more behaviours than what the SystemC interpreter may exhibit. This conservative abstraction enforces the guideline: if a safety property can be proved on the HPIOM model, then it is true that the `wait` statements have not been used to enforce synchronization.

4.3 Semantics of process code into HPIOM

Compiling imperative code into automata is a well known problem and there is no semantic difficulty here. However, the abstract syntax tree for a C++ contains a lot of particular cases, and a lot of them have to be taken into account if we want to apply our tool to real-world SystemC

designs. Hence this part of the translation represents a significant part of the work. The `while` loop is given in figure 4 as an example.

Abstractions: the translation of C++ code has to perform some abstractions due to the absence of dynamic structures in HPIOM. For instance, we cannot translate C++ code performing memory allocations. Non-recursive function calls can be inlined (at the syntax tree level), but other have to be abstracted away.




```
while (x < 3) {
   x := x + 1;
}
```

Figure 4. Control flow for a while loop

4.4 Semantics of the synchronization primitives and the scheduler

Expressing the semantics of the scheduler by some synchronizations between the HPIOM automata may be done in several ways. The global communication scheme is shown in figure 5 and will be detailed below. The semantics of the SystemC scheduling policy is modeled by one automaton for the scheduler, plus two per process. The fist one represents its control structure (as explained above), and the other one represents its state in the scheduler (figure 6): The process may be either running, ready to run (eligible), or sleeping (blocked in a wait statement for a *SC_THREAD*, or execution over for an *SC_METHOD*.). The synchronization between the two is such that the first automaton (representing the control structure) may change state only if the second one is in state "running".

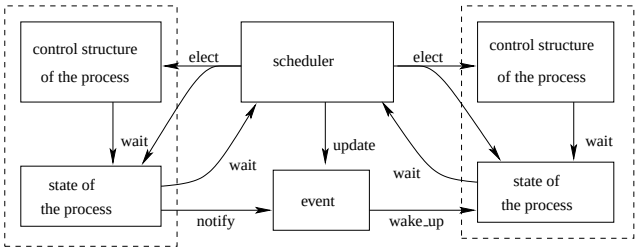
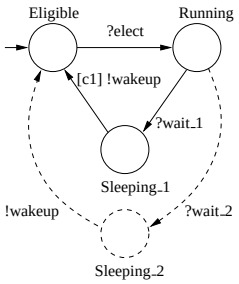


Figure 5. Global view of the communication between the automata in HPIOM

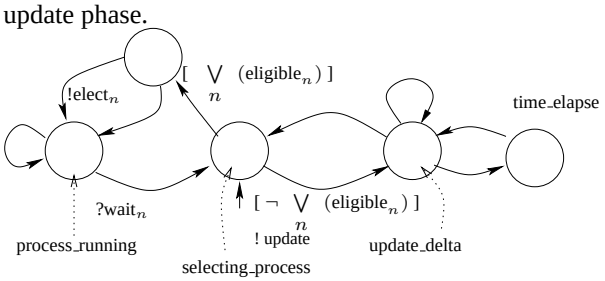
The scheduler itself is represented by an additional automaton (figure 7). It starts in a state "selecting_process". At that moment, all the processes are eligible. The SystemC official definition lets the choice between the eligible processes unspecified. In our model, the scheduler chooses one process non-deterministically: when we prove a property of a SystemC design including this non-deterministic



Synchronizations:
elect: received from the scheduler when the process is chosen,
wakeup: sent to the control structure,
wait_2: received from the control structure when a `wait` statement is reached,
c1 and **c2** correspond to the conditions the process is waiting for in the corresponding "sleeping" state.

Figure 6. State of a SystemC process

scheduler, we prove it for any possible implementation. The election corresponds to the transitions emitting `elect_n` on figure 7. Then the scheduler runs the elected process: in the automaton representing the state of the process, this means taking the transition from "eligible" to "run". When the process has finished its execution (go to "sleep" state), the scheduler selects another one, and so on until there is no more process eligible. Then, the scheduler goes to the update phase.



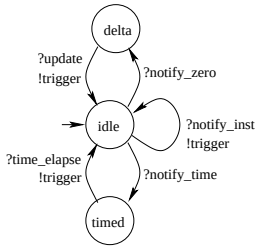
Synchronizations:
elect_n: sent to the corresponding process state automaton and control structure
wait_n: received from the corresponding process state automaton
update: sent to all processes that may have an action to execute during the update phase

Figure 7. Pattern of the SystemC scheduler.

The low-level synchronization primitive in SystemC is called an `sc_event`. C++ objects of type `sc_event`, like other SystemC objects, are instantiated only during the elaboration phase. During the simulation, the operations available for an `sc_event` are:
— `notify()`: the event is triggered immediately,
— `notify(SC_ZERO_TIME)`: the event will be triggered

at the end of the δ-cycle,
— `notify(`*time*`)`: the event is scheduled to be triggered at some date in the future.

We also build one HPIOM automaton for each `sc_event`, according to the pattern of figure 8. It has one initial state plus one state for each kind of delayed notification. The immediate notification is modeled by a single transition. In any case, the transition going back to the initial state is the transition triggering the event. It emits a message that will move processes waiting for that event from the "sleeping" state to the "eligible" state.



Synchronizations:
notify_... messages are received from the control structure when a `notify` statement is encountered,
trigger goes to the process state automaton, and will make the condition to the "eligible" state true,
update and **time_elapse** both come from the scheduler.

Figure 8. Pattern for an sc_event

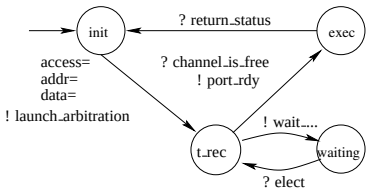
4.5 Direct semantics of TLM constructs

As mentioned previously, although TLM constructs are library components whose code could be translated using the above translation schemes, we advocate a translation in which these constructs are given a direct semantics in HPIOM. This allows to exploit the information they give on the structure of the design. In this section, we sketch the HPIOM encoding of the TAC Channel.

A TAC channel is a bus that may be connected to several *master* modules able to initiate transactions through it, and to several *slave* modules receiving these transactions. Technically, in the SystemC TAC code, initiating a transaction on a TAC, with some address, results in a function being called in the slave corresponding to the address. The TAC may be idle if no transaction is initiated; it may also deal with several transactions "at the same time" (meaning: in the same execution of the main loop of the scheduler algorithm, Figure 2). In this case, a `tac_seq` will treat them in arrival order, a `tac_arbiter` will use an arbitration policy. The principles of the encoding into HPIOM are the following.

Wait for the channel to be available First, for each master port, we create a "waiting" automaton synchronized with the master and with the TAC: it simulates the master process waiting for the TAC to be available (Figure 9). If the TAC is not available when a transaction is initiated by a master, the master should let other processes run. It will become eligible again when the TAC selects its transaction.

Select transaction and resolve the address The TAC itself is modeled by the complex automaton of Figure 10.



Synchronizations:
channel_is_free is received from the channel when it is available.
wait and **elect** are communication with the scheduler to wait until **channel_is_free** is present.
return_status is received when the transaction processing is over.

Figure 9. Wait for channel availability

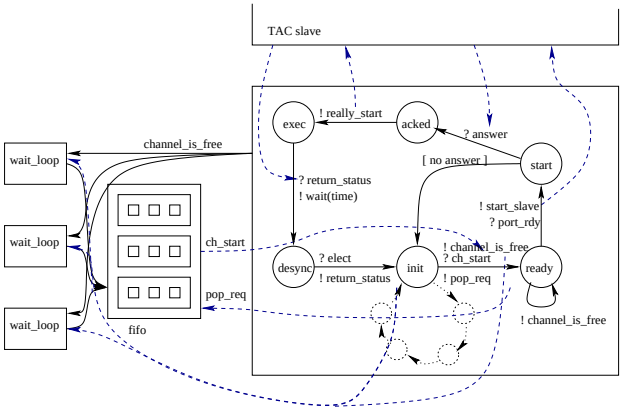


Figure 10. Pattern for a tac_seq

It loops in the initial state until it receives a transaction. When transactions are ready to be executed, values identifying them are entered in a FIFO (we encode finite FIFOs into HPIOM). The automaton of the channel processes them one by one and goes to state "ready". A message is sent to all the automata modeling slaves, and those whose address map matches answer. If the channel gets no answer, then it returns immediately, with a status `is_no_response`. In the example above (figure 3), the first process elected will send the first transaction which will be processed immediately, and the next one will be queued until the transaction is processed.

Execute the corresponding method in the slave module When the transaction has been selected and the slave identified, the body of the corresponding method in the slave module is executed and the status is returned (state "exec" on figure 10). (The scheme is a bit simplified here, since the channel has to communicate with several instances of slave modules.)

Simulate a `wait` to allow other processes to execute If a slave module answered, then the automaton of Figure 10 simulates a `wait` statement on a time duration (state

"desync"). This is included in the protocol to allow other processes to execute (which is necessary because the scheduler is not preemptive). In the example, this means that the second transaction will be processed before the control flow of the first one comes back to the master module.

5 The tool LusSy

The tool LUSSY has an internal structure similar to the one of a compiler (figure 11). The front-end (called Pinapa) extracts information from the system, a second pass compiles it into an intermediate representation called HPIOM, taking the semantics of SystemC into account, and a code generator gives a textual representation for it, which can be used as input by other tools. Our SystemC front-end combines a traditional compiler front-end with a runtime information extractor. We use the front-end of the GCC compiler to get the abstract syntax tree of the body of the processes. But then we need to extract information from the runtime system, right after the execution of the so-called "elaboration phase" of SystemC, that builds the components and the connections between them (typically, which signal a port is connected to). We also need to link this information to the abstract tree.

The HPIOM representation can easily be converted into several formats usable by external tools. We now have a LUSTRE back-end that allows us to use LESAR and NBAC, and two visualization back-ends (one to view the connections between automata and one to view the automata themselves) using the *dot* format of the graphviz (`www.graphviz.org`) package.

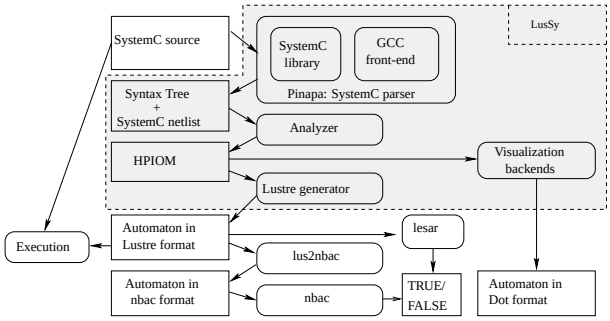


Figure 11. Tool chain

6 Applying LUSSY to the example

Let us come back to the example from section 2.4. In the module `signal_master`, we write a value on the channel at the address 8 after writing the value `false` on a signal. The module `signal_slave`, mapped at this address, will receive the transaction, and check that the value of the signal is `false`. This may seem trivially true, but it is not: the semantics of `sc_signal` says that the value is actually taken into account only at the next δ-cycle. As there is no `wait` statement between the `write` and the `read` statements, the value read is the previous value. During the first iteration of the loop, the value read is the initial value of the signal. In practice, with the current implementation of the SystemC library, the value is initialized to `false`, but it is clear from the SystemC specifications that the initial value is unspecified. So the bug does not appear during simulation, but NBAC can not prove the property. The diagnosis provided when the proof fails gives the condition on the initial value of the signal (`true`), that causes the bug. If we explicitly initialize the signal to `false`, then, the property becomes provable, and if we explicitly initialize it to `true`, then, the property is false and the assertion is actually violated during execution. Now, we have identified the bug, we can fix it by adding a wait statement:

```
while (true) {
  out_bool.write(false);
  wait(SC_ZERO_TIME);
  status = master_port.write(address, x);
}
```

Then, the assertion is verified, and NBAC is able to prove the correctness of the assertions. Now, look at the module `status_master`. It just writes on the channel, and tests the status returned. If we write, at a mapped address, a value not equal to 4322, then the property is true, and provable by NBAC (but not by LESAR, since the property is data-dependent). If we change either the address written to, or the address map to make it write on an unmapped address, then, the first assertion becomes false. If we write the data 4322, then the slave sets the error flag, the second assertion becomes false, and the proof fails.

On this example, we have the complete verification flow. The prover has been able to prove true properties with no manual intervention in less than one second. The model contains 28 automata in parallel, with a sum of 104 states, 196 transitions, 10 numerical variables and 50 Boolean variables. This seems quite a lot in comparison to the size of the source, but the global state-space is not as large as it seems to be, because the system is made of a lot of tightly synchronized small automata. Moreover, we use symbolic tools, for which the number of variables is the key point, and not the number of global states. Optimizations have to address the number of variables first.

7 Conclusion

We have presented our approach and tools for the analysis of SystemC transactional models. Starting from the source code of a SystemC design, we parse it using GCC's C++ front-end and the SystemC library itself, then transform it into a set of automata, and finally dump it in the Lustre language. The implementation is operational and the faithfullness of the translation has been validated on basic examples, by comparing the executions of the generated

Lustre with the executions of the "official" systemC implementation.

We experimented the connection to two different model-checkers that do not perform the same amount of abstractions. The main idea of the approach is to extract as much information as possible from the SystemC design, and let the verification tools perform the abstractions they need.

This work has also helped getting a better understanding of the dangerous constructs of SystemC. We identified several cases of data-race conditions. For example, writing twice on the same signal during the same δ-cycle, most cases of immediate event notification, and unfortunately the current implementation of the TAC channels lead to scheduler-dependent behaviors. Global variables or inter-module function calls are both dangerous and irrelevant to hardware modeling. These remarks led to the first SystemC guidelines. For instance the latter leads to a guideline that requires the use of explicit SystemC or TLM constructs to model communication, and forbids the use of shared memory mechanisms.

Further work: We are currently applying the whole approach to a significant case-study provided by STMicroelectronics, in order to identify optimizations of the encoding. We will also experiment on HPIOM traditional compiler techniques like live variable analysis.

LUSSY being open, we can easily move to another model-checker, by writing a translator from HPIOM to its input format. We are starting experimentations with SMV and SAT solvers. One of the easiest to try is the tool by Prover Technologies. Indeed, Lustre is the basis of the SCADE environment provided by Esterel Technologies, which is equipped with a plug-in by Prover Technologies, providing SAT solving and Presburger arithmetic. Our translation of HPIOM into Lustre can be directly used in SCADE.

As LUSSY provides a formal semantics of SystemC, it can be the basis of a toolbox for the development of systems-on-a-chip at the transactional level, providing tools for all the questions related to TLM design: verification and test at the TLM level, comparison of TLM and RTL levels, analysis of non-functional properties. For instance, the formal semantics can be used as a support for the automatic generation of test sequences, intended to be run on both the TLM design and a corresponding RTL design. This reference semantics is also the necessary starting point for comparing executions at different levels of abstraction.

Since HPIOM preserve the potentially complex algorithms of SystemC code, powerful software verification techniques could be used, (invariant extraction, predicate abstraction, etc.). We could also consider extending HPIOM to manage dynamic data-structures, but this would require efficient support in the provers. A more promising approach is the systematic use of *contracts* for some of the components. We already mentioned the case of processor components: a processor is an interpreter of the binary code, and it has to deal with complex data which is the code C to be executed! C should be abstracted (the values exchanged being replaced by unknown values encoded by inputs) but we need some assumptions about the behavior of the processor concerning the way it synchronizes with other components. This can be described by a contract.

References

[1] T. Ball and S. K. Rajamani. Boolean programs: A model and process for software analysis. Technical report, Microsoft Research, February 2000.

[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] E. Clarke and D. Kroening. Hardware verification using ANSI-C programs as a reference. In *Proceedings of ASP-DAC 2003*, pages 308–311. IEEE Computer Society Press, January 2003.

[4] E. Clarke, D. Kroening, and F. Lerda. A tool for checking ANSI-C programs. In K. Jensen and A. Podelski, editors, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2004)*, volume 2988 of *Lecture Notes in Computer Science*, pages 168–176. Springer, 2004.

[5] R. Drechsler and D. Grosse. Formal verification of LTL formulas for systemc designs. `http://www.informatik.uni-bremen.de/grp/ag-ram/doc/konf/iscas03_verification_systemc.pdf`.

[6] P. Godefroid. Model checking for programming languages using VeriSoft. In ACM, editor, *Conference record of POPL '97, the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages: papers presented at the symposium, Paris, France, 15–17 January 1997*, pages 174–186, New York, NY, USA, 1997. ACM Press.

[7] 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*, Sept. 1992.

[8] K. Havelund. Java pathfinder: A translator from java to promela. Sept. 30 1999.

[9] K. Havelund, S. Park, and W. Visser. Java pathfinder - second generation of a java model checker. May 27 2000.

[10] B. Jeannet. Dynamic partitioning in linear relation analysis. application to the verification of reactive systems. *Formal Methods in System Design*, 23(1):5–37, July 2003.

[11] L. Lamport. Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering*, SE-3(2):125–143, 1977.

[12] F. Maraninchi and Y. Rémond. Argos: an automaton-based synchronous language. *Computer Languages*, (27):61–92, 2001.

[13] K. L. McMillan. *Symbolic Model Checking*. PhD thesis, Boston, 1993.

[14] Open SystemC Initiative. *SystemC v2.0.1 Language Reference Manual*, 2003. `http://www.systemc.org/`.

[15] T. Sakunkonchak and M. Fujita. Verification of synchronization in SpecC description with the use of difference decision diagrams. In *FDL*, 2002.