Des Autom Embed Syst (2006) 10:73–104
DOI 10.1007/s10617-006-9044-6

LusSy: An open tool for the analysis of systems-on-a-chip at the transaction level

Matthieu Moy · Florence Maraninchi · Laurent Maillet-Contoz

Received: 15 November 2005 / Revised: 25 April 2006 / Accepted: 28 April 2006

Abstract We describe a toolbox for the analysis of Systems-on-a-chip written in SystemC at the transaction level. The tool is 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.

Keywords SystemC · Formal verification · Model-checking · Semantics · Pinapa · Lussy

1. Introduction

1.1. Using SystemC to model systems-on-a-chip

Performance and quality requirements for embedded systems are increasing quickly. The physical capacity of chips can usually grow fast enough to satisfy those needs, but one of the design flow's bottlenecks is the design productivity (this is often referred to as the "design gap"). New techniques such as component reusability and use of embedded software have to be settled continuously to be able to fill in this gap. These new methodologies raised the need for new modeling and simulation *languages*, since low-level hardware description languages such as VHDL or Verilog would not be simulated fast enough to allow embedded software development or preliminary architectural exploration.

M. Moy (✉)· F. Maraninchi
Verimag, Centre équation - 2, avenue de Vignate, 38610 GIÈRES, France
e-mail: Matthieu.Moy@imag.fr
e-mail: Florence.Maraninchi@imag.fr

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

SystemC [4] has been designed to meet these requirements. A SystemC program has *modules*, *signals*, and other building blocks for the model. It is made of several processes that run *"in parallel"*. The architecture of a SoC, the code describing the activity of its software parts, and the description of its hardware parts, can be given in SystemC. Other classes can be added to allow higher level communication like bus or network protocols, to raise the level of abstraction to the *Transaction Level Modeling* [32, 19].

A number of other approaches have been proposed for the description of heterogeneous hardware/software systems with an emphasis on formal analysis. See, for instance, Metropolis [5]. In this type of approach, the (formal) definition of the description language is part of the game. On the contrary, SystemC has not been defined with formal analysis in mind. It is primarily a simulation and coordination language, aiming at accepting all kinds of hardware or software descriptions in a single simulation.

SystemC is deliberately based on open standards like C and C++, for two reasons: first, it guarantees a fast learning-curve for the engineers of the domain; second, it guarantees that the models of systems developed in SystemC can be exploited even if the tool that was used to build them is no longer available. SystemC is normalized by the Open SystemC Consortium Initiative, involving the major actors of the domain. SystemC is now an IEEE standard (IEEE 1666). It is currently used by major silicon companies like Intel, STMicroelectronics, Philips, Texas Instruments, etc.

Besides this status of a de-facto standard in the hardware industry, SystemC is also gaining interest in other domains where the design of complex systems is based on simulations and virtual prototyping. Being based on a quite high-level general-purpose programming language, it has very few limitations regarding the complexity of the systems to be described. Moreover, it is based on a very clear component view of systems. All these points make it a good candidate as a standard language for virtual prototyping of component-based computer systems. Since SystemC is used to define *reference models* of complex systems, we think that the formal and automatic analysis techniques that can be applied to this language are worth studying, and are likely to find their way in the design flow, to find bugs as earlier as possible.

1.2. Need for new Verification Tools

Verification methods are well established for RTL. Bugs in hardware are known to be extremely costly, and various techniques are applied to find them as efficiently and as soon as possible. The introduction of a new abstraction level implies the creation of a complete development environment, including programming languages, editors, debugging and visualization tools, and verification tools.

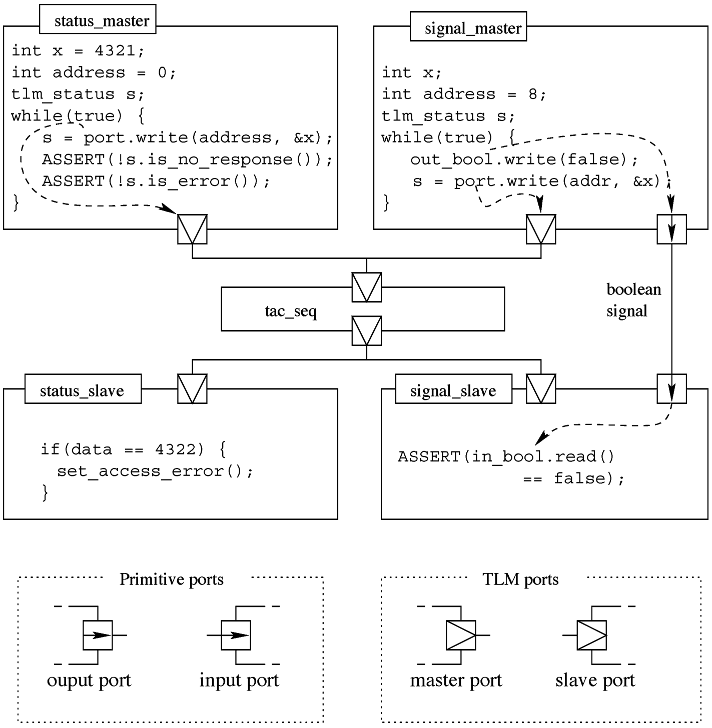
The question of verification tools is a key point for the wide adoption of TLM models in the industry, and is being addressed by a joint project between Verimag and the SPG team of STMicroelectronics. The main problems are: 1. What does it mean to validate properties at the TLM level? Validation can be 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 (timing, consumption, . . .) of SoCs at the TLM level? In this document, we report on the work done for addressing the first item: how to give a formal semantics to SystemC and the additional TLM constructs, and then express and verify properties of a TLM design written in SystemC.

1.3. Example of a SystemC program

Figure 1gives an example of a SystemC program. For clarity, we only show the body of the processes, and the methods 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 (a TLM sequentializer channel developed in STMicroelectronics). The program contains *assertions* for the properties we want to verify. The main function is given in Fig. 2.

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 is such that `status.is_no_response()` is true. If a module is mapped at this address, the status returned is such that `status.is_ok()` is true, unless the method `set_access_error()` has been called during the `WriteAccess` call.

The execution of a SystemC program is as follows. The SystemC kernel executes the `sc_main` function. At the beginning of the execution is the *elaboration* phase. Components



```
int x = 4321;
int address = 0;
tlm_status s;
while(true) {
  s = port.write(address, &x);
  ASSERT(!s.is_no_response());
  ASSERT(!s.is_error());
}
```

```
int x;
int address = 8;
tlm_status s;
while(true) {
   out_bool.write(false);
   s = port.write(addr, &x);
}
```

```
if(data == 4322) {
  set_access_error();
}
```

```
ASSERT(in_bool.read()
           == false);
```

Fig. 1 An Example Transactional Model

```
 1  int sc_main(int argc, char ** argv) {
 2    tac_status_master * stm =
 3      new tac_status_master("status_master");
 4    tac_signal_master * sim =
 5      new tac_signal_master("signal_master");
 6    tac_status_slave * stsl =
 7      new tac_status_slave("status_slave");
 8    tac_signal_slave * sisl =
 9      new tac_signal_slave("signal_slave");
10    tac_channel * channel =
11      new tac_channel("CHANNEL");
12    stm->master_port(channel->slave_port);
13    sim->master_port(channel->slave_port);
14    stsl->slave_port(channel->master_port);
15    sisl->slave_port(channel->master_port);
16    sc_signal<bool> sig;
17    sim->out_bool(sig);
18    sisl->in_bool(sig);
19    sc_start();
20  }
```

Fig. 2 Elaboration of a SystemC Program

are instantiated in the usual way for C++ objects. Elaboration ends with a call to the function `sc_start()` that hands the control back to the SystemC kernel (line 19 of Fig. 2). The last part of the execution is the simulation of the program's behavior where the SystemC kernel executes the processes one by one, with a non-preemptive scheduling policy.

1.4. Summary of the paper

The main contributions of this work are 1) a method to extract syntax and architecture information from a SystemC program, and it's implementation, PINAPA; 2) an executable, formal semantics for SystemC models, including additional TLM constructs; 3) a working connection to verification tools. This tool chain called LUSSY already allows us to prove properties (generic, or expressed directly in C++) on small platforms. Some work is still to be done to be able to scale up, for example using a component-based approach, but we already provide the building blocks.

The paper is organized as follows: section 2 presents the architecture of LUSSY; Sections 3 and 4 will detail the two main components: PINAPA and BISE; Section 5 presents briefly the application of the method to an example; Section 6 is the conclusion.

2. Overview of LUSSY

2.1. Candidate Tools for the Verification of SoCs

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 [28]. However, these tools are tailored for the RTL, exhibiting a clear notion of clock that is absent in TLM models. 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 is limited to the subset of SystemC that allows to write RTL designs. It cannot be used for real TLM designs (See [15] for instance).

Now, since SystemC is mainly a C++ library, one could think that we need 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 algorithms; 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. Moreover, they usually do not take parallelism into account. For instance, CBMC [12, 13] can apply bounded model-checking techniques on pure C models, but does not deal with parallelism, or with infinite loops. SLAM [6] uses clever abstractions and refinement techniques, but also focuses on sequential programs.

VeriSoft [20] 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 behavior. 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). Other works propose improvements for run-time verification: some by providing tools to check properties during a simulation (for example [34]), others by generating sets of test cases with a better code coverage ([7], [17]). In general, those simulation-based techniques scale up better than formal verification, but give weaker guarantees on the reliability of the platform.

The closest related work is to be found in Java model-checking, which also takes a scheduler specification into account. The first version of the Java Path Finder model-checker [23] 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 [24] 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 usually present in SoC models.

2.2. Our Verification Approach

We advocate an approach able to exploit all the particularities of a real TLM design written in general SystemC. 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 behavioral information automatically 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. This is very much in the spirit of the *action language* [10]. For the moment LusSy connects this intermediate representation to model checkers, abstract interpreters, and a SAT engine. These 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; the implementation is already mature enough to accept a large subset of SystemC. Code for *not yet implemented* constructs is being added progressively.

2.3. Expressing Properties

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

- Check that a global dead-lock never occurs. We consider that a global dead-lock occurs when the SystemC scheduler enters the "time elapse" phase while no process is waiting for time. The particular notion of scheduling of SystemC makes a global dead-lock equivalent to the reachability of a state in the automaton of the scheduler.
- Check that a process never terminates. This should always be the case except for test benches. This corresponds to the case when a process goes to the "sleeping" state without any wake-up condition, that is: no `next_trigger` have been called in `SC_METHOD`, and the last statement of the function is never reached for `SC_THREAD`.
- Check that a synchronous signal is never written on twice during the same δ-cycle (a δ-cycle is a step of execution of the system running all the eligible processes, in zero time). 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.

We can also express user-defined properties. We may check that some portions of code are mutually exclusive by specifying the beginnings and ends of the critical sections. 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.

2.4. The LUSSY Tool Chain

The tool chain is presented in Fig. 3.Starting from a SystemC program's source code, the front-end, PINAPA, extracts an abstract representation comprising both the architecture and the syntax-related information. BISE uses the output of PINAPA to generate the intermediate representation HPIOM (expressing the semantics of SystemC). BIRTH performs some HPIOM to HPIOM transformations. The translation from HPIOM to any synchronous language is then rather straightforward.

LUSSY is the composition of PINAPA, BISE, BIRTH and the various back-ends.

The following gives a short presentation of each element of the tool chain. The main components will be detailed in the next sections. A complete description of all the components can be found in [29].

2.4.1. PINAPA*: Pinapa Is Not A PArser*

PINAPA [31] is a SystemC front-end based on GCC and on the SystemC library. Its role is similar to the one of a compiler front-end for a traditional programming language, but the way it works is very different since we are not dealing with a real programming language, but with a library built on top of C++.

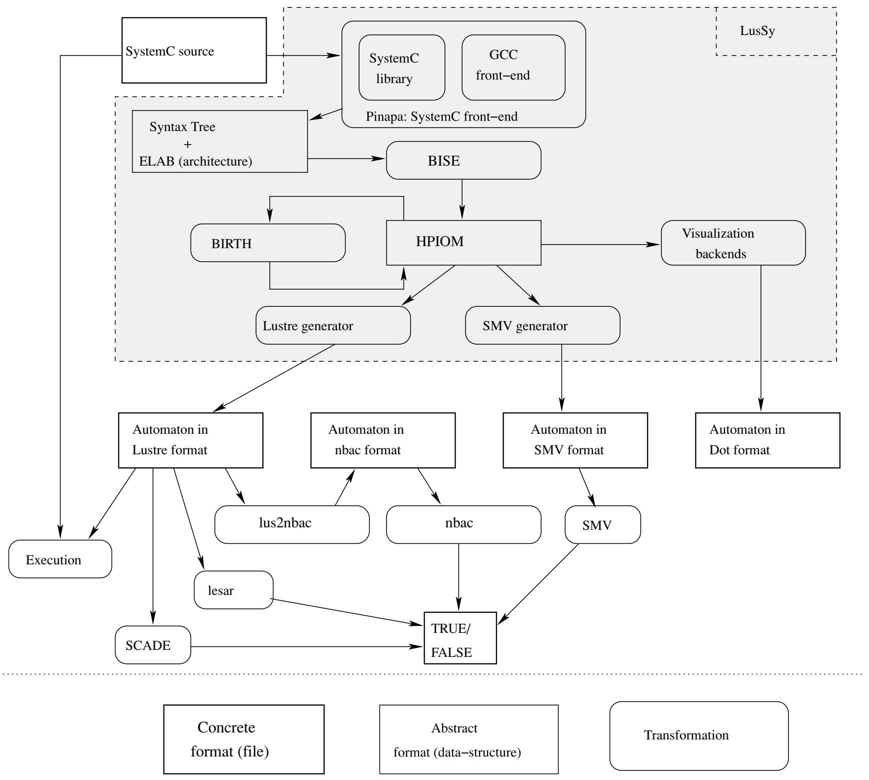


Fig. 3 The LUSSY Tool Chain

2.4.2. BISE*: Back-end Independent Semantic Extractor*

BISE [30] takes the output of PINAPA as input. It defines the data structure HPIOM (for Heterogeneous Parallel Input/Output Machines), an intermediate formalism of communicating, synchronous automata. It generates a system of HPIOM automata whose semantics is a conservative abstraction of the input program, with as few abstractions as possible.

2.4.3. BIRTH*: Back-end Independent Reductions and Transformations of* HPIOM

BIRTH implements some HPIOM to HPIOM transformations: they are optimizations (traditional optimization techniques like live variable analysis), abstractions, or simplifications (expression of high-level HPIOM constructs using lower-level constructs to ease the task of the back-ends).

2.4.4. LUSTRE *and* SMV *back-ends*

We currently have a LUSTRE [8] and an SMV [28] back-end which allow us to use SMV, LESAR [22], NBAC [25] and PROVER PLUG-IN^{TM} for SCADE^{TM} [33] to carry the actual proof.

2.4.5. Visualization back-end

The visualization back-end was mainly written for debugging purposes. From the HPIOM automata, it generates one `dot` file for each automaton, and one to represent the connection between automata in the system. The tool DOTTY from the graphviz package provides an interactive visualization tool.

3. PINAPA: Extracting architecture and behavior information from SystemC models

3.1. Introduction

The first step of the analysis it to extract the information from the SystemC program. This section will present our tool, PINAPA, that is able to extract the syntax and architecture information from the program. We will see how the case of SystemC is different from traditional programming languages, and present our approach and its implementation.

A tool like PINAPA is compulsory for anybody who wants to extract information from realistic SoCs designs: it is able to extract all the information from a piece of SystemC code, with very few limitations. It is open source and available to public.

3.1.1. Static and dynamic information in SystemC

SystemC, like several programming languages or run-time environments, is used for describing: 1) the architecture of a system and then 2) the activity of the elements in this system. The architecture, although it is built by the execution of some piece of code (the so called "elaboration" phase), is not really *dynamic*, and will not change during the simulation of the program activity. It is described in a general-purpose programming language because of the expressivity of such languages, compared to the dedicated pseudo-languages of "configuration files".

The originality is that SystemC, although often referred to as a *language*, is not actually a language, but a library for C++. Execution of a SystemC program is "trivial", since it can be compiled with any supported C++ compiler. But simulation is not the only thing one may want to do with a language. We are particularly interested in the connection to formal verification tools, but also in visualization tools, automatic generation of documentation, program linting, ...

3.1.2. PINAPA*: requirements*

This section presents PINAPA (For **P**inapa **I**s **N**ot **A** **PA**rser), a SystemC front-end. Our requirements when we started writing PINAPA were the following (they also apply for a general use):

1. As few *a priori* limitations as possible. We cannot make any assumption about a well-defined subset of SystemC used in the programs we want to analyze, and we don't want the tool to require any manual annotation.
2. The tool must give precise information on all parts of the program: architecture, software parts, hardware parts. Abstractions may be done in the back-end if necessary, but the front-end must not lose information.

3. PINAPA should maximize code reuse, because using an ordinary C++ front-end when possible avoids creating C++ dialects, and using the reference implementation of SystemC also helps complying with the SystemC specifications.
4. The programs we want to manipulate use some high level Transaction Level Modeling constructs, that are not yet standardized by the SystemC consortium. The tool must be able to manage those constructs.

3.1.3. Contributions

PINAPA satisfies all the abovementioned requirements. The contributions are the following: 1) a general principle for building front-ends of "simulation" languages in which part of the system architecture that has to be extracted statically is actually built by the execution of some piece of code; 2) an open source implementation of this principle for full SystemC. It has been tested on the TLM model of the Example AMBA SYstem (EASY) [3] from ARM written in SystemC by STMicroelectronics, whose complexity is representative of the designs written in SystemC; 3) working connections to analysis tools.

When fed with a SystemC program, PINAPA executes the elaboration phase of the program, parses it with GCC, and outputs a data structure usable through the GCC and SystemC APIs, plus some additional PINAPA-specific functions.

3.2. Existing SystemC Tools

Several other tools manipulate SystemC programs. Some present themselves as SystemC front-ends, but none of them meet our requirements.

SystemPerl [35] is a perl library containing, among other tools, a netlist extractor for SystemC (a *netlist* is a description of the connections between modules). It uses a simple grammar-based parser and will therefore not be able to deal with complex code in the constructors of the program, and does not extract any information from the body of the processes. This does not satisfy requirement 2 above.

Synopsys^{TM} developed a SystemC front-end that has successfully been included in products like CoCentric SystemC Compiler and CoCentric System Studio [38][37]. It parses the constructors and the main function, as well as the body of the modules with the EDG [16] C++ front-end, and infers the structure of the program from the syntax tree of the constructors. SystemCXML [27] seems to use the same approach, using doxygen's [39] C++ front-end, but the implementation details are not published as of now. Using this technique, to be able to parse any SystemC program (requirement 1), one must be able to compute the state of any program at the end of the execution of the constructors, knowing their bodies. In other words, the tool must contain a re-implementation of a C++ interpreter (which does not satisfy requirement 3).

The University of Bremen developed a SystemC front-end called ParSyC [18]. The approach is similar to the one of Synopsys^{TM}, except that the grammar is written from scratch (including both SystemC and C++ constructs) instead of reusing an existing C++ front-end. It has important limitations regarding the complexity of the elaboration phase. For example, "`for`" loops have to be unrolled, which is not possible if the bounds are not constant. sc2v [40] is also a SystemC synthesizer, built with the same approach. KaSCPar recently came into the picture, with a grammar-based parser (using JavaCC), and an XML output. To be complete, this approach needs to include all the C++ syntax (to parse the program) and semantics (to interpret the constructors).

Some lint tools such as AccurateC [1] also manipulate SystemC code. AccurateC can check rules both in the code (this is an extension of a C++ lint tool) and in the netlist. However, it does not need the link between the behavior and the netlist (unfortunately, the internal structure of AccurateC has not been published at time of writing).

Some simulation tools provide an alternative to the reference simulator, with additional features like VHDL or Verilog cosimulation. For simulation, these tools do not need information about the body of the processes in uncompiled form, so, their requirements are different from ours. One particular case is NC-SystemC [11] from Cadence: it also provides source-level debugging, using the EDG C++ front-end. The approach is therefore similar to ours, since the tool has to deal with both syntax and architecture information. However, this tool is focused on debugging, and the front-end is anyway not available to the public.

3.3. PINAPA principles, limitations and uses

3.3.1. Principles of PINAPA

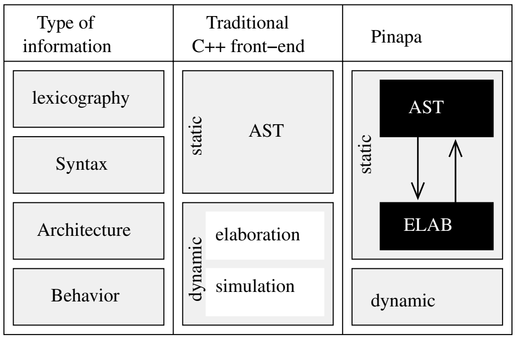
The methodology for writing or generating front-ends for various kinds of languages has been studied extensively (see for example [2]). Such general techniques are used indirectly in our tool since we are using a general C++ front-end, but are not sufficient to get all the necessary information from a SystemC program. Typically, they cannot extract the information about the SoC architecture, which is built by executing the first phase of the SystemC program.

At first, it may appear meaningless to write a front-end for a library, but the case of SystemC is particular. To understand what we mean by "SystemC front-end", we need to examine the notions of *static* and *dynamic* aspects of a SystemC program.

Observe Fig. 4.On the left are the kinds of information present in a SystemC program. From the point of view of a C++ front-end, lexicography and syntax are static and used to build the AST (Abstract Syntax Tree), while the architecture and the behavior are visible during the execution. From the PINAPA point of view, the *static* information extends to include the architecture. The architecture will be present in the memory at the end of the elaboration phase. The dynamic part is reduced to the simulation phase. The static part is made of: the AST obtained by reusing a standard C++ front-end on the SystemC program; the architecture-related information (that we call ELAB) that stays in memory at the end of the elaboration phase.

Figure 5describes the data-flow of PINAPA. The AST is obtained by parsing the program with a traditional C++ front-end (right-hand side of the figure), and ELAB is obtained by compiling and executing the elaboration phase (left-hand side of the figure).

Fig. 4 Static and Dynamic information in a SystemC program



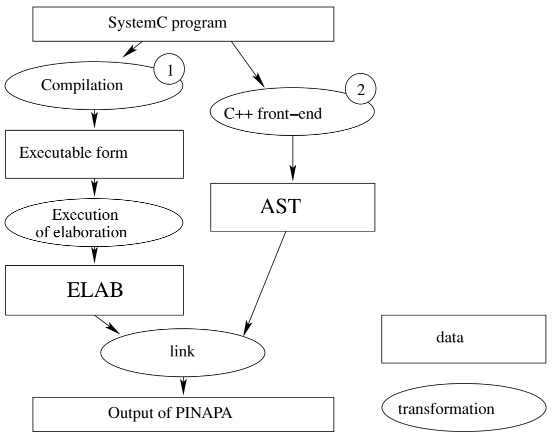


Fig. 5 Data-flow of PINAPA

Note that the source code is parsed by a C++ front-end twice: first, to compile the elaboration phase (step (1) in Fig. 5), and then to get the AST, which, in our case, contains some address offset information (offset of a data-member of a class compared to the address of the class), dependent on the binary interface (ABI) (step (2)). The address offsets obtained in both cases must be consistent. This means the C++ front-ends of (1) and (2) must be ABI-compatible, that is, the layout of the data-members of a class must be calculated exactly in the same way (in particular, the two front-ends may be the same compiler). We currently use GCC (GNU Compiler Collection) version 3.4.1 for both. This means that the program will not be parseable by PINAPA if it does not compile with this precise version of GCC. This may seems to be too strong a limitation, but see the comments at the end of the section.

Unlike other existing approaches, PINAPA has no limitation regarding the complexity of the code of the constructors used to build the architecture, because it does not interpret them; it compiles and executes them. For example, a program reading a configuration file or the command line arguments to determine the number of modules to instantiate can be parsed correctly by PINAPA. In the example of Fig. 2, for instance, the initial values of some data-members depend on command-line arguments, but they will be extracted correctly by PINAPA.

Moreover PINAPA (like the front-end of Synopsys^{TM}) uses a real C++ front-end and will therefore correctly parse any code that would have been parsed successfully by the C++ front-end. The limitations regarding the C++ language itself are therefore minor (limited to "details" such as the `export` keyword not managed by GCC). The use of macros in the source code is not a problem: the macros will be expanded by the C++ preprocessor. Whether the code uses the macro or its expanded version doesn't have any influence on PINAPA. For example, whether the user writes

```
SC_MODULE(name) {...}
```

or

```
struct name : public sc_module {...}
```

does not have any influence on PINAPA (the second form may indeed have to be used in the case of multiple inheritance). PINAPA would deal as well with user-defined macros. Any tool using a dedicated grammar for SystemC would have to include all the grammar and typing rules of the C++ standard in the tool to have a correct parser.

The main task of PINAPA is to establish links between the AST and ELAB (last step in Fig. 5). The idea behind these links is illustrated by the following: the SystemC processes perform actions of the form `port.write (...)`, and these instructions are present in the AST. The elaboration phase creates instances of modules and connects ports, building an architecture that is present in ELAB. The relationship between an instruction `port.write (...)` in the AST, and the actual data structure describing this port in ELAB, has to be established by PINAPA. In practice, PINAPA installs pointers in both directions between the AST and ELAB.

3.3.2. Limitations

While PINAPA has no limitations (except the ones of GCC) regarding the AST (we use a C++ front-end) or ELAB (we let a C++ compiled code *execute* the constructors), it does have limitations due to the way we establish the links between the AST and ELAB.

It is not always possible to establish these links. If a process uses a pointer to a SystemC port or an array of ports, then, the actual object pointed to by this pointer cannot be known statically. In some cases, advanced static analysis techniques like abstract interpretation would allow to get more information statically, but the subset of SystemC managed by the tool would be very hard to define. In practice, those constructs are usually not considered as good programming practice and did not appear in the programs used as input for PINAPA up to now.

PINAPA simply does not manage references and pointers to SystemC objects (the pointers to ports will appear in the output of PINAPA, both in AST and in ELAB, but the objects in the AST will not be linked to the corresponding ones in ELAB). For arrays of SystemC objects, if the index is a constant, then, the actual object is known statically, and PINAPA decorates the AST referring to the port with a pointer to this object. Otherwise PINAPA decorates the AST with the index in the array (which is itself an AST) and a pointer to the first element of the array. In any case, we could reduce the case of arbitrary array indexes to the case of constant indexes by transforming the code to eliminate non-constant indexes, while preserving the semantics of the program. This can be done with loop unrolling (which converts the loop index into a constant), or transformations like turning `ports[i]` into

```
i==0 ? ports[0] : (i==1 ? ports[1] : (abort(),ports[1]))
```

PINAPA being open-source, such a transformation can easily be added if needed.

The use of templates can sometimes be problematic: for a program using templates, The AST contains the expanded templates, and ELAB contains instances of templates class. The template parameter is not necessarily known at the time the back-end is written, so the code of the back-end has to manipulate pointers to objects whose type is not known. The same remark applies to PINAPA itself. In practice, the management of templates made the task harder, but never impossible for PINAPA and our back-end LUSSY (we had to move relevant data or methods from template classes to non-template base classes in SystemC and the TLM library).

Finally, the limitation on SystemC designs being analyzable only if they can be compiled with a precise version of GCC deserves a comment. Recall that our point of view is that the elaboration phase of a SystemC description should be compiled and then executed, instead of being interpreted by some (potentially limited) C++ interpreter. Since we perform some low-level pointer assignments in linking the AST with ELAB, this implies that a SystemC description cannot be analyzed with any C++ compiler, and this is indeed a limitation. However, the approaches based on a dedicated C++ interpreter have much more limitations. Moreover, we use a widely accepted tool (GCC); we have to face some limitations of a code we did not write ourselves, but that we could fix ourselves if needed, since GCC is free software.

3.3.3. Other possible approaches

3.3.3.1. Using a C++ Interpreter An interesting option would be to modify an existing C++ interpreter like UnderC [14]. A C++ interpreter contains a C++ front-end, and the environment to execute the elaboration phase. Ideally, the C++ interpreter should be 100% compliant with the C++ standard, and do the interpretation at the AST level (not on an intermediate byte-code representation, which is unfortunately the case of UnderC) to ease the link between the AST and the run-time information. We are not aware of any such interpreter.

3.3.3.2. Avoiding the need for a SystemC front-end The problem solved by our approach is the expressivity of the language used to describe the program's architecture. Another approach would be to eliminate the problem instead of solving it, by using a less expressive language.

In particular, the SPIRIT [36] XML Schema can be used to describe the architecture of the program. There are ongoing works to extend it to support TLM constructs. Extracting the structure of the program would then consist in parsing an XML file, and extracting the body of the processes would still have to be done with a C++ front-end. Simulation of the program would also be possible, by generating C++ from XML and compiling it as usual. This approach is not applicable today since we need to deal with existing SystemC programs.

3.3.4. PINAPA*: Current and Future Uses*

We currently use PINAPA as a front-end for our formal verification tool LUSSY [30]. Starting from the abstract representation of the program provided by PINAPA, we generate an intermediate representation (a set of communicating automata) which is itself dumped in a text format used as input for a traditional model-checker.

STMicroelectronics is currently developing a visualization tool for SystemC using PINAPA: reading a SystemC program, it generates another representation usable by a visualization tool (using the SPIRIT format). This is a very simple use of PINAPA since it only uses the ELAB part of the information extracted, but it is being extended to provide more advanced visualization including static information about process communication and synchronization. Our medium-term plans include the development of a lint tool for SystemC and our TLM methodology. The tool has to be able to identify both the architectural and the language constructs, which is exactly the scope of PINAPA.

PINAPA is also successfully being used by a research project for compositional verification of transactional models of Systems-on-a-Chip, led by the POP ART team of INRIA Rhône-Alpes (France), in co-operation with STMicroelectronics: the idea is to apply LUSSY's approach with a compositional verification tool called PROMETHEUS [21]. The tool, SC2PROM, reuses our front-end, PINAPA, and generates its own intermediate format, which is based on an asynchronous formalism with priorities. This approach should allow splitting large proofs into smaller ones.

3.4. Implementation of PINAPA

PINAPA can be divided into three main tasks: 1) get the ELAB information by executing the elaboration phase; 2) get the AST of the process bodies using GCC; 3) make the link between

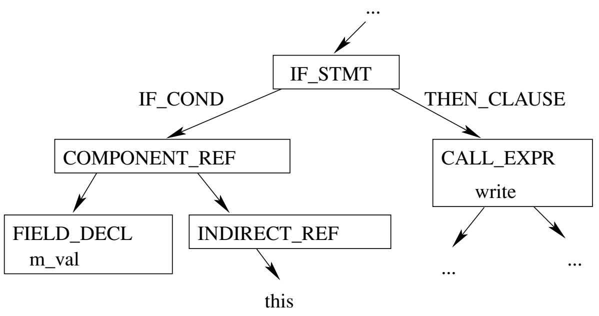


Fig. 6 Abstract Syntax Tree for an `if` statement

the results of phases 1 and 2. Phases 1 and 2 are just software reuse. For phase 1, fortunately, SystemC keeps a list of most objects in a global variable, it is easy to examine them.

In Fig. 1 page 75, each graphical element corresponds to an object in ELAB, which contains:

process handlers. The process handler gives the following information: name of the function, name of the class containing it, type of process (`SC_THREAD` or `SC_METHOD`), and pointer to the executable code of the function. It also contains the list of events the process may be waiting for by default after suspending itself. This list is called the *static sensitivity list*. The static sensitivity list just provides a default argument for `wait` and `next_trigger` statements, so a process with a static sensitivity can use `wait()` (with no argument) or omit the `next_trigger` statement, but this is mainly syntactic sugar.

SystemC objects. Each SystemC object (port, module, ...) contains the necessary information about the binding.

The AST represents the bodies of the processes. For example, the `if` statement line 9 in Fig. 2 would be represented as in Fig. 6.PINAPA will make the link between the AST of the port `port` and its instances in ELAB.

We want to link AST and ELAB to each other, as shown in Fig. 7:each process handler will be linked to the corresponding AST, and each occurrence of a SystemC object in the AST will be linked to its instances (one instance per instance of the process, see below).

Concretely, PINAPA first launches the elaboration of the program. We use a slightly modified version of SystemC, in which we redefined the function `sc_start()` called by the program at the end of elaboration. Instead of launching the simulation, our version of SystemC launches a C++ front-end. A few other minor modifications have been necessary. Usually, they were as simple as adding a `friend` keyword or a data member to a class. For example, SystemC did not keep the name of the class for each module, but the modified SystemC does.

In phase 2, GCC parses the functions one by one. We actually ignore many of them, since we are only interested in the body of processes. We get an abstract representation of the source code of the processes in the form of an Abstract Syntax Tree (AST).

Then, the actual job of PINAPA begins: we have to make the link between this AST and ELAB. Sections 3.4.1, 3.4.2 and 3.4.3 detail some interesting problems raised by this phase. Section 3.4.5 summarizes the architecture of the tool.

3.4.1. Links from ELAB to AST

The first step is to make the link from ELAB to the AST. There is not much to do: for each process handler, we look for the AST of a method with no argument whose class name and function name match the ones in the process handler, and we add a pointer to this AST in it.

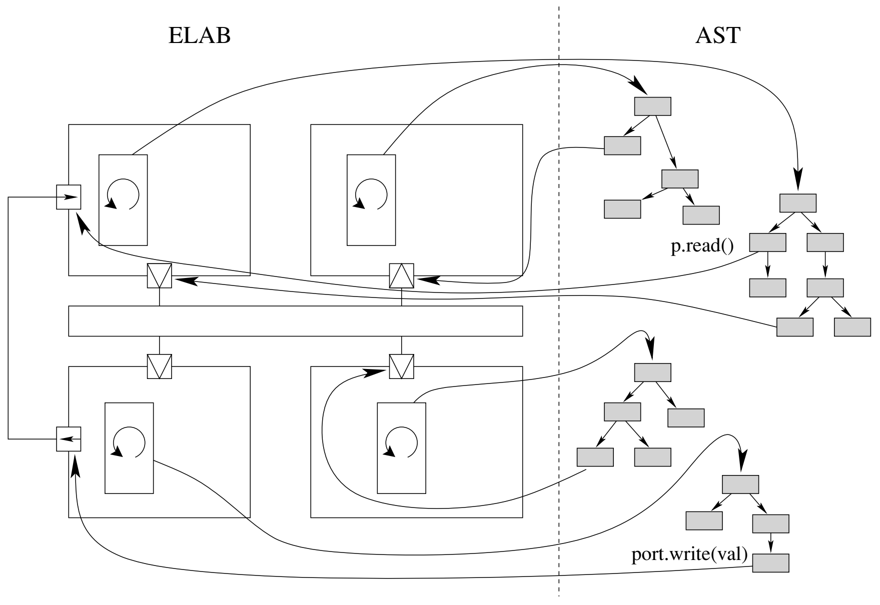


Fig. 7 Link between AST and ELAB

In the example above, there are two process handlers for `module1::code1` (one for each instance of `module1`), and each of them points to the AST of function `module1::code1` declared at line 8.

3.4.2. Links from AST to ELAB

The link from the AST to ELAB is a bit more complex. Each instruction in the AST corresponding to a function or object of the SystemC library must be considered as a SystemC primitive and requires a special treatment.

3.4.2.1. SystemC functions SystemC function calls (in the process bodies) are recognized by their name and list of arguments. We add a decoration to the tree saying that this function is a SystemC function (and which one it is).

For `wait` statements, we also add a representation of the list of sensitivity (information saying when the process will wake up) for this statement, either based on the arguments of the `wait`, or on the static sensitivity list (built during elaboration) for a `wait` with no argument.

3.4.2.2. SystemC objects SystemC objects require much more work. In the AST, we get an abstract representation of the classes, but in ELAB, we have *instances* of these classes. These instances are built once and for all during elaboration. Unless the program uses pointers to SystemC objects, a variable containing such an object will therefore always contain the same object.

Since a module may be instantiated more than once, the same element in the AST may refer to several objects in ELAB. However, for a given process, an element in the AST only corresponds to one object in SystemC. The link is a hash table: (AST, process handler) ⟶ SystemC object.

We describe below two methods to get a pointer to an object in ELAB from its AST and process handler, and how we applied them in the case of GCC. Depending on the information present in the AST, either one, the other, or both can be applicable using another C++ front-end, depending on the information provided by this front-end.

An example: SystemC communication ports In the case of GCC, SystemC communication ports correspond to a situation where the name of the object does not appear in the AST. This is due to the way GCC represents a member function call in the AST: for example, when the user writes `port.write(x);` in a process body, if `port` is a member of the current class, this is equivalent to `this->port.write(x);` Which is itself converted into `write(this->port, x);` by GCC's front-end. Now, here is the bad joke: this code is converted into `write(*(this +` *offset_of_port*`), x);` where *offset_of_port* is a literal numerical constant. At this point, we are still in the GCC front-end, but we have lost an important information: the name of the port.

So, we only have the offset of the port being examined, and we want to get its instance in ELAB. Since the compiler used for the C++ front-end and the one used to compile the program are ABI-compatible, the solution is the following: for each instance of the process, we can get a pointer to the instance of the class containing the process (this information was already in the original SystemC's process handlers). If we add the offset we got from the AST to the value of this pointer, we get a pointer to the instance of the port.

Other objects The same approach is used for other SystemC objects like `sc_event`.

3.4.3. C++ Classes data members

It is often the case that a data member of a class is initialized during elaboration, and we would like PINAPA to be able to extract this information from the program. PINAPA provides an option to read the value of these data members at the end of elaboration.

The problem is that in this case, the address offset does not appear in the AST in the output of the front-end of GCC. The approach of section 3.4.2.2 is therefore not applicable. We could compute the offset from the AST (GCC does this anyway, later in the compilation flow), but we choose a different approach, that does not require this computation.

Since we have the name of the data member and the name of the class it is a member of, we can write a piece of C++ code that would read the value of this data member. The C++ language is not flexible enough to execute dynamically this piece of code, but never mind: we can write it in a file, compile it (run `g++` as an external program), load it dynamically (`dlopen`, `dlsym`, ...), and execute it. It will be executed in the environment ELAB. An example of generated code follows:

```
#include "preprocessed_sc_source.cpp"

namespace pinapa {
struct get_value {
  static bool
  function_to_get_value_0(sc_module * arg) {
    return (static_cast<module1 *>(arg))->m_val;
  }
  [...]
};
[...]
} // namespace pinapa
```

In the current implementation, the return value is converted into an AST representing the value of the constant, which is attached as a decoration to the AST of the program.

There is a limitation here because the return value of the generated function has the same type as the data-member that we are examining, which can be any type. To be able to call this function from PINAPA, we have to know the return type statically. Concretely, this means we need to write a piece of code in PINAPA for each data-type we want to manage. In a future version, it would be interesting to implement the conversion from a concrete value to an AST in the generated code itself. The return value of the function would always be an AST, and this would remove the above limitation. In other words, code generation can be generic on the type of the variable, whereas function calls can not.

3.4.4. A note on module hierarchy

SystemC provides a notion of hierarchical component. This means that an object (module, port, ...) can be instantiated as a sub-component of another one. While this is essential to organize the design, it has no influence on the semantics. For completeness, PINAPA keeps a pointer `parent` in each object, pointing to the containing object, but other components of LUSSY will not use it.

3.4.5. Function call graph

The resulting function call graph in PINAPA is somewhat complex (Fig. 8),but will be made clearer by the end of this section.

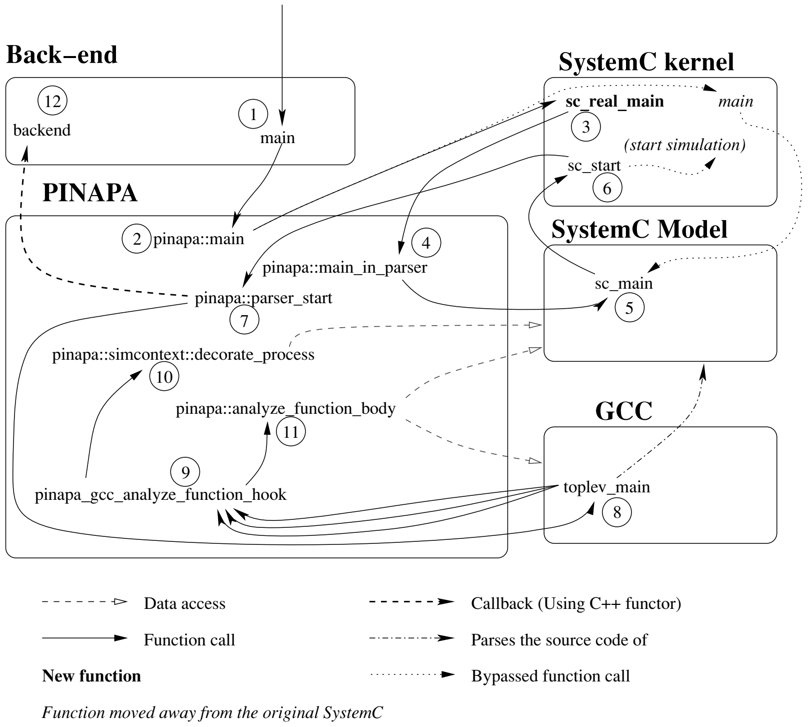


Fig. 8 Architecture of PINAPA

In the original version of SystemC, the main function is in the SystemC library itself (it actually does not do much more than displaying a copyright message and calling the `sc_main` function). We have removed it from the library, considering that the main function should be written by the user (i.e., the programmer of the back-end). This is the item (1) of Fig. 8.

The call to the `sc_main` function (and therefore the call to the PINAPA's main function) must not return, because the elaboration phase may have allocated objects on the stack. We use therefore a callback mechanism (using a C++ functor), so the main function of the back-end should look like:

```
int main(int argc, char ** argv) {
  // The backend is in my_callback::operator()
  pinapa::st_backend *callback = new my_callback();
  // ...
  pinapa::main(..., callback);
}
```

From the `pinapa::main` function (2), we call the function that was originally the `main` function in the SystemC kernel (3), which in turn calls the `pinapa::main_in_parser` function (4), which dynamically loads and executes the user's code (5) to elaborate the program. The call to `sc_start` (6) that originally started the simulation is bypassed and calls `pinapa::parser_start` (7).

The elaboration has now been executed. We call the main function of the GCC compiler (8). We have modified GCC to call the function `pinapa_gcc_analyze_function_hook` (9) in PINAPA for each function it parses (passing the AST of this function as an argument). For each function parsed, `pinapa::simcontext_decorate_process` (10) searches for the corresponding process handler in ELAB and `pinapa::analyze_function_body` (11) runs over the AST to link SystemC primitives to their corresponding object.

3.4.6. Validation

We developed PINAPA incrementally, following our needs for the formal verification back-end LUSSY. Each feature added to PINAPA was validated by at least one example, stimulating both the front-end and the back-end. The correctness of the translation can be ensured by the examination of the model-checker's diagnosis compared to the simulation behavior, and by the visualization tools connected to LUSSY.

3.5. Summary for PINAPA

We presented PINAPA, a front-end for SystemC. Unlike traditional compiler front-ends, it executes a part of the program before parsing it, and the main work presented in this section is the way to make the link between the source code representation and the run-time information.

This technique allowed us to write a SystemC front-end with very few limitations, with a minimal effort. It reuses megabytes of source code from GCC and SystemC, but counts itself less than 4,000 lines of code. The performances are reasonable: most of the time is spent in GCC, so parsing a program with PINAPA takes almost the same time as compiling it with GCC. It already manages the TLM TAC and TLM BASIC extensions of SystemC, and other could be added in the future depending on our needs.

The parser is already operational and used in two formal verification back-ends. It is available under the terms of the GNU Lesser General Public License at `http://greensocs.sourceforge.net/pinapa/`. More details about the implementation can also be found in the online documentation.

4. BISE: Semantics of SystemC and TLM constructs in terms of automata

4.1. Introduction

We have presented the first step of the extraction, and our implementation of it, PINAPA. This section will illustrate the transformation of a SystemC program, parsed by PINAPA, into the intermediate representation HPIOM, which is a simple formalism of communicating synchronous automata. This transformation is implemented in the component BISE of the tool LUSSY.

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 LUSTRE back-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 BISE 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;

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 the source 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.).

4.2. Limitations

4.2.1. Limitations of LUSSY *due to* HPIOM *Design*

The choice of a limited expressivity for HPIOM has some consequences on the possibilities of LUSSY. Since we have no support for dynamic data-structures in HPIOM, any SystemC program using dynamic data-structures or non-terminal recursive calls will have to be approximated during the conversion. We manage to perform the approximation in a way that preserves the properties we want to verify.

4.2.2. *State explosion*

The main and well known issue when it comes to formal verification is the state explosion phenomenon. A large platform given as input to LusSy would be parsed by Pinapa, and transformed into hpiom by Bise without any problem, but the generated hpiom could not be used directly to perform model-checking. As it is, LusSy allows the verification of small platforms as a whole, but requires other techniques for large platforms. There are ongoing works to include abstract interpretation techniques, and a compositional approach, which should allow to deal with reasonably large platforms in the future.

4.3. hpiom: Heterogeneous parallel input/output machines

hpiom is the intermediate formalism used in the LusSy tool-chain. We have chosen a formalism of communicating synchronous explicit automata, which has several advantages: first, it is easy to manipulate, visualize, and debug (compared to a pure data-flow description for example); second, it is an appropriate representation for many optimizations (like live variables analysis); third, it is easy to build from an imperative language. The synchronous parallel composition allows us to describe complex behaviors as the parallel composition of small automata, without paying the price of additional states in the product, as it would be the case with an asynchronous product. A complete and formal description of hpiom can be found in [29].

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. Figure 9gives an example graphical representation for such an automaton.

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 and values of a message (eg: ?*message*);
- a list of messages emitted. messages can be either pure (present or absent), or valued (absent, or present and carrying a value).
- a set of parallel assignments denoted by $v := e$ where $v \in V$ and e is an expression on V.

Graphically, a pure message emission is denoted by !*message*, and a valued message emission is written !*message*(*value*). For instance, the automaton of Fig. 10has the guard $y > 5$ on the edge t_2 on which it performs an assignment. The pure message m_1 and the valued message m_2 (with value x) are emitted on the edge t_1. A condition on the presence of a message is denoted by ?*message* (true if and only if *message* is present).

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

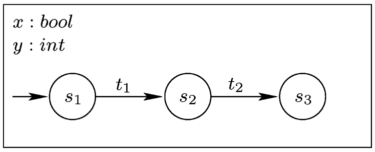


Fig. 9 Simple hpiom Automaton

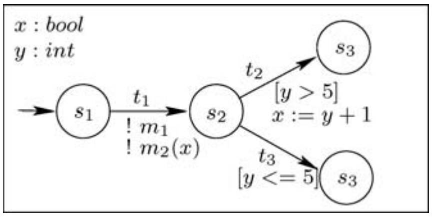


Fig. 10 HPIOM Automaton with Signal Emission, Guard, and Assignment

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 graphical syntax, we usually omit self-loops that have no effect (no emitted message, no assignment).

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 (they cannot intersect since HPIOM automata do not share variables). It expresses the semantics of the synchronous broadcast of signals in all synchronous languages, on which the reader can find more details in [26]. The synchronous broadcast used here is known to raise so-called "causality" problems [9]. The automata we build in HPIOM do not exhibit causality problems because we carefully avoid loops in the communication patterns (an example loop would appear with an automaton waiting for a in order to emit b, in parallel with an automaton waiting for b in order to emit a).

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. We also have the notion of explicitly unknown values, i.e., variables whose value is non-deterministic. They will be compiled into inputs of the system when converted to a deterministic language like Lustre. Semantically speaking, there is no difference between unconstrained input and non-determinism. This mechanism is used whenever a SystemC design exhibits non-determinism: the initial values of uninitialized signals, the choice of a process to be run by the scheduler, etc.

4.4. Semantics of SystemC into HPIOM

4.4.1. 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 4.4.2). 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.

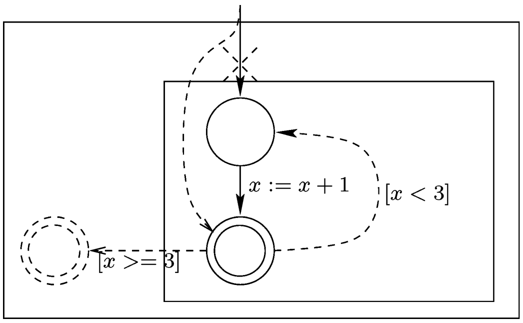
4.4.2. Expressivity of HPIOM *and abstractions*

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 values. 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 addresses 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* $R_1, \ldots, R_n$. With each R_i, we associate a Boolean variable b_i. An address variable x is then encoded by a valuation of the vector $b_1 \ldots b_n$. 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`



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

Fig. 11 Control flow for a while loop

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 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.

4.4.3. 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 Fig. 11as 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.

4.4.4. 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 Fig. 12and 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 (Fig. 13gives the automaton for an `SC_THREAD`. The automaton for an `SC_METHOD` is similar): 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 states only if the second one is in state "running".

The scheduler itself is represented by an additional automaton (Fig. 14).It starts in a state "selecting_process". At that moment, all the processes are eligible. The SystemC

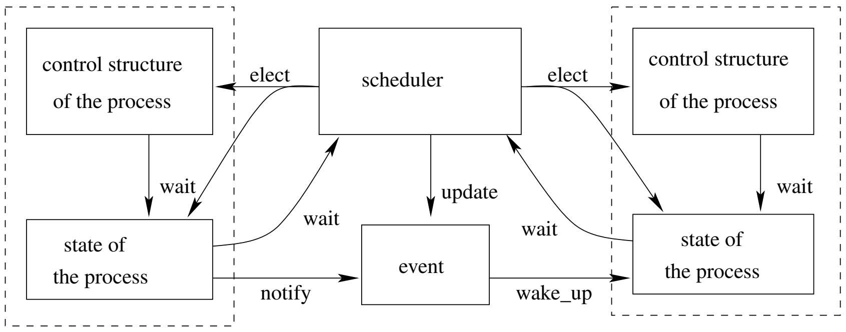
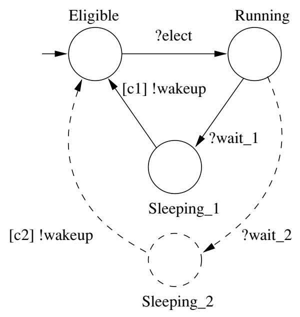


Fig. 12 Global view of the communication between the automata in HPIOM



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.

Fig. 13 State of a SystemC process (SC_THREAD)

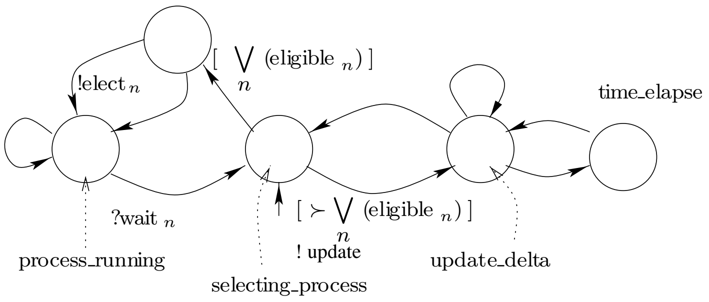
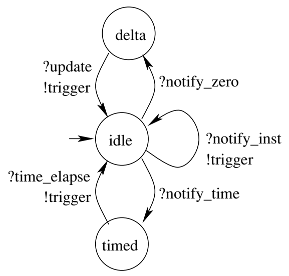


Fig. 14 Pattern of the SystemC scheduler.



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.

Fig. 15 Pattern for an sc_event

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 scheduler, we prove it for any possible implementation. The election corresponds to the transitions emitting `elect_n` on Fig. 14. 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.

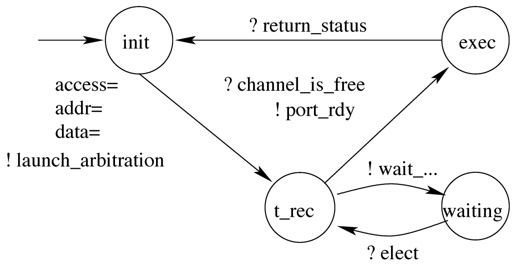
The low-level synchronization primitive in SystemC is called an `sc_event`. C++ objects of type `sc_event`, like other SystemC objects, are instantiated only during the elaboration phase. During the simulation, the operations available for an `sc_event` are:
— `notify()`: the event is triggered immediately,
— `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 Fig. 15.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.

4.4.5. 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". 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:



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.

Fig. 16 Wait for channel availability

4.4.5.1. 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 (Fig. 16).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.

4.4.5.2. Select transaction and resolve the address The TAC itself is modeled by the complex automaton of Fig. 17.It loops in the initial state until it receives a transaction. When

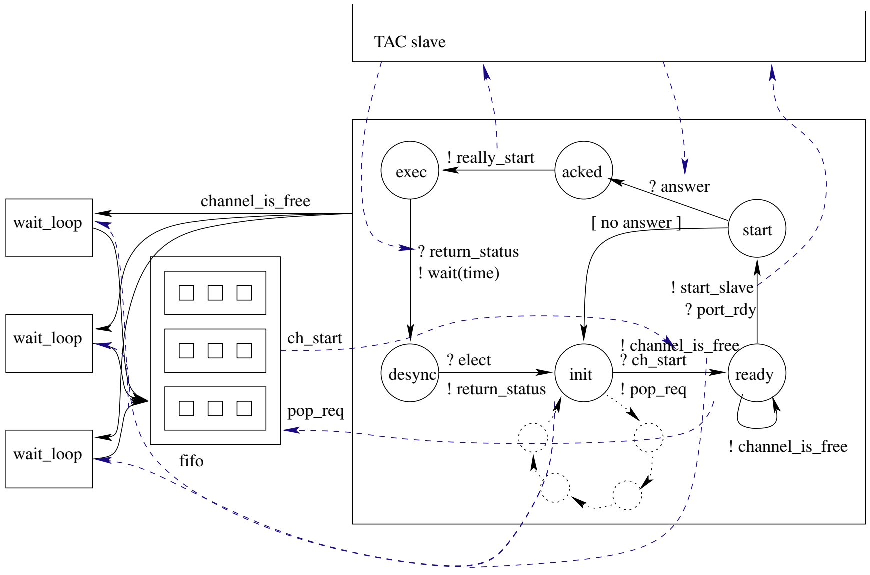


Fig. 17 Pattern for a tac_seq

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 platform (Fig. 1 page 75), 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.

4.4.5.3. 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 Fig. 17). The scheme is a bit simplified here, since the channel actually has to communicate with several instances of slave modules.

4.4.5.4. Simulate a `wait` *to allow other processes to execute* If a slave module answered, then the automaton of Fig. 17 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.

4.5. Summary for BISE

With the help of PINAPA, BISE is able to transform a SystemC model into a set of HPIOM automata. We manage a large subset of SystemC, and some TLM constructs that are not (yet) part of SystemC. Many constructs have not been implemented by lack of time, but could be added easily.

A direct application of this is the connection to model-checking tools. HPIOM being simple and well defined, a code generator for a particular text format can be implemented in a few hundreds of lines of code. We currently have a LUSTRE and an SMV back-end, which allows us to use the complete LUSTRE and SCADE tool chains, as well as the SMV symbolic model checker. The following section provides an example use of the tool chain, for the verification of a small system.

5. Applying LUSSY to the example

Let us come back to the example from section 1.3. PINAPA extracts all the information correctly, BISE generates the HPIOM model, and we can generate LUSTRE or SMV code. In the SMV back-end, we made a complete abstraction of integer values, since SMV's management of integers uses an n-bit exact encoding, which is too costly to be applicable on non-trivial systems. LESAR would anyway abstract them internally. NBAC uses abstract interpretation combined with clever dynamic partitioning, and is able to prove data-dependent properties (noticeably slower than pure Boolean proofs, however). We tried those three tools on the example platform and tried to prove the assertions in the program.

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, for example by adding a wait statement:

```
while (true) {
  out_bool.write(false);
  wait(SC_ZERO_TIME); // let the signal update its value
  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 and SMV, 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. If we remove this data-dependent assertion, then SMV and LESAR can also successfully prove the other one.

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.

6. Conclusion

6.1. Summary

We have presented our approach and tools for the analysis of SystemC transactional models. These models appear very early in the design flow, and one of their use is to be a reference model for further developments. Their reliability is therefore very important. This paper is particularly interested in their validation with formal techniques, connecting SystemC to existing verification tools.

Starting from the source code of a TLM design written in SystemC, 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 and SMV languages. The implementation is operational and the faithfulness 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 several 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 is compulsory if we want the tool to remain open, and novel verification engines to be connected to it. The LUSSY chain of tools works very well on small programs. We are currently working on the optimization of the translation chain, mainly in order to minimize the number of variables involved in the HPIOM model. This is a key point, since we mainly use symbolic tools. This should allow to use LUSSY on larger programs. But we are convinced that formal verification for a SoC taken as a whole is impossible, because of the complexity of such mixed hardware/software objects. Our general goal is to use LUSSY as a basic tool in a verification methodology based on the intrinsic component structure of SoCs. We comment on this point in the following subsection.

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 notifications, and unfortunately the current implementation of the TAC channels lead to scheduler-dependent behaviors. Global variables or arbitrary 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.

6.2. Perspectives

The general objective of this project, in co-operation between STMicroelectronics and Verimag, is to provide a complete development and verification environment for Transaction Level Models written in SystemC.

Since 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. Those tools can either use the output of PINAPA or the HPIOM produced by BISE.

The main perspectives of the work described in this paper are: 1) a formalization of the various levels of abstraction that appear in the design flow of SoCs, concerning time; 2) improving testing methods by using the elements of the LUSSY tool-chain; 3) using software-oriented verification tools in LUSSY; 4) verification methods that exploit the component structure of systems-on-a-chip.

We comment on these points below.

Timed and untimed layers in the design flow Some work is still to be done to get a better understanding of the timed and untimed layers inside the TLM levels of abstraction, and improve the methodologies to manage timing in TLM platforms accordingly. Jérôme Cornet (Verimag and STMicroelectronics) started working on the subject in 2004, and already developed abstract models to represent the possible executions of a timed system. The final goal is a methodology supported by development and possibly validation tools, to add the "timing" layer to a pure functional model. This should be done without modifying the functionality of the reference untimed model.

Improving the testing methods The current validation techniques used in production are simple executions, with a relatively basic oracle. These run-time methods could be greatly improved. The work carried out by Claude Helmstetter (Verimag and STMicroelectronics) aims at improving the coverage of the test-benches while avoiding redundant tests as much as possible. It is based on an instrumented simulation detecting suspicious executions at run-time. The algorithms are currently designed, and are partially implemented, but require an instrumentation of the platform to be analyzed. This manual instrumentation could be made automatic with a tool able to parse the platform and insert the necessary pieces of code in the right places. PINAPA can be a very helpful basis for such an automatic instrumentation tool. The run-time algorithms could also be improved by working in combination with a static code analyzer to reduce "false positive" answers when several processes access a shared resource. The other components of LUSSY can be helpful for this purpose.

Using software-oriented verification tools Since HPIOM preserves some of the potentially complex algorithms of SystemC code, powerful software verification techniques could be used (invariant extraction, predicate abstraction, etc.). We consider extending HPIOM to manage dynamic data-structures, in order to connect LUSSY to some software verification tools able to exploit this information.

Towards component-based verification methods We are currently applying the LUSSY toolchain to the TLM model (provided by STMicroelectronics) of the EASY [3] platform. The EASY platform is a small-size system-on-a-chip. In LUSSY, all the steps of the translation are performed correctly. Up to now, the generated SMV and LUSTRE codes are too large to be provable as a whole, which makes EASY an interesting case-study for component-based verification methods.

Exploiting the intrinsic component-structure of systems-on-a-chip described at the TLM level can be done mainly in two ways: either we perform aggressive automatic abstractions on the models of the components, before we try to prove something on the system as a whole; or we ask the designers to provide the appropriate abstractions for their components. The two methods can be used together in a complete verification environment.

Automatic abstractions can be based on the structure of the source code, since PINAPA provides all the information. It may need to extend HPIOM to support annotations coming from the structure of the program, and usable by dedicated verification techniques. For example, HPIOM automata for the individual processes in SystemC use transitions as "microsteps" to represent the computation between two synchronization points, and also to represent these synchronizations steps. However, since the scheduler is non-preemptive, the parts of the control flow between explicit synchronization points are atomic, with respect to the parallel composition of the processes. It would be easy to use PINAPA in order to identify the parts of the automaton associated to a process that constitute atomic reactions of this process. Of course, since processes are described with a general programming language, an atomic reaction can be any algorithm, with loops and conditional structures. This means that a sub-automaton of any shape has to be abstracted by *one* transition. The difficulty is to label this transition with a kind of *summary* of the computations that happen inside the component, and that would be sufficient for the global view and analysis of the system. We will consider abstract interpretation techniques for determining these "summaries".

If we consider the other approach, in which we ask the designer to provide local specifications, a promising approach is the systematic use of *contracts* for some of the components. They can be used for at least two purposes: verify, component by component, that each part of the platform meets its specifications (either by formal verification or by testing), and abstract a component, replacing it by its specification for a global proof. 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 C code to be executed! The C code 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. Actis Design, LLC. 'AccurateC^{TM} Rule Checker', 2005. `http://www.actisdesign.com/`.
2. Aho, A. V., R. Sethi, and J. D. Ullman. *Compilers—Principles, Techniques, and Tools*. Reading, MA, USA: Addison-Wesley, 1986.
3. ARM Limited. 'AHB Example Amba SYstem technical reference manual', 1999. `http://www.arm.com/pdfs/DDI0170A.zip`.
4. Aynsley, J., D. Long, S. Vandeputte, and S. Swan. 'SystemC v2.0.1 Language Reference Manual', 2003. Open SystemC Initiative. `http://www.systemc.org/`.
5. Balarin, F., L. Lavagno, C. Passerone, A. L. Sangiovanni-Vincentelli, M. Sgroi, and Y. Watanabe. 'Modeling and Designing Heterogeneous Systems'. In: *Concurrency and Hardware Design, Advances in Petri Nets*. pp. 228–273, 2002.
6. Ball, T. and S. K. Rajamani. 'Boolean Programs: A Model and Process for Software Analysis'. Technical report, Microsoft Research, 2000.
7. Baray, F., P. Codognet, D. Diaz, and H. Michel. 'Code-based Test Generation for Validation of Functional Processor Descriptions'. In: *TACAS*. pp. 569–584, 2003.
8. Bergerand, J.-L., P. Caspi, N. Halbwachs, D. Pilaud, and E. Pilaud. 'Outline of a Real Time Data-Flow Language'. In: *Real Time Systems Symposium*. San Diego, pp. 33–42, 1985.
9. Berry, G. 'The Foundations of Esterel'. In: G. Plotkin, C. Stirling and M. Tofte (eds.), *Proof, Language and Interaction: Essays in Honour of Robin Milner*, pp. 425–454, 2000.
10. Bultan, T. 'Action Language: A Specification Language for Model Checking Reactive Systems'. pp. 335–344, 2000.
11. Cadence Design Systems. 'NC-SystemC', 2003. `http://www.cadence.com/products/functionalver/nc-systemc/`.
12. Clarke, E. and D. Kroening. 'Hardware Verification using ANSI-C Programs as a Reference'. In: *Proceedings of ASP-DAC 2003*, pp. 308–311, 2003.
13. Clarke, E., D. Kroening, and F. Lerda. 'A Tool for Checking ANSI-C Programs'. In: K. Jensen and A. Podelski (eds.). *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2004)*, Vol. 2988 of *Lecture Notes in Computer Science*. pp. 168–176, 2004.
14. Donovan, S. 'The UnderC Development Project', 2001–2002. `http://home.mweb.co.za/sd/sdonovan/underc.html`.
15. Drechsler, R. and D. Große. 'Formal Verification of LTL Formulas for SystemC Designs', 2003. `http://www.informatik.uni-bremen.de/grp/ag-ram/doc/konf/iscas03verificationsystemc.pdf`.
16. Edison Design Group. 'Compiler Front Ends', 1996–2005. `http://www.edg.com/`.
17. Ferrandi, F., M. Rendine, and D. Sciuto. 'Functional Verification for SystemC Descriptions Using Constraint Solving'. In: *DATE*. pp. 744–751, 2002.
18. Fey, G., D. Große, T. Cassens, C. Genz, T. Warode, and R. Drechsler. 'ParSyC: An Efficient SystemC Parser'. In: *Synthesis And System Integration of Mixed Information technologies*, 2004. `http://www.informatik.uni-bremen.de/agram/doc/work/04sasimiparsyc.pdf`.
19. Ghenassia, F. (ed.). *Transaction-Level Modeling with SystemC. TLM Concepts and Applications for Embedded Systems*. Springer. ISBN 0-387-26232-6, 2005.
20. Godefroid, P. 'Model checking for programming languages using VeriSoft'. In: ACM (ed.): *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*. New York, NY, USA, pp. 174–186, 1997.
21. Goessler, G. and A. Sangiovanni-Vincentelli. 'Prometheus—A Compositional Modeling Tool for Real-Time Systems'. In: *EMSOFT*, 2001.
22. Halbwachs, N., 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*, pp. 785–793, 1992.
23. Havelund, K. 'Java PathFinder: A Translator from Java to Promela'. p. 152, 1999.
24. Havelund, K., S. Park, and W. Visser. 'Java PathFinder—Second Generation of a Java Model Checker', 2000.

25. Jeannet, B. ‘Dynamic Partitioning In Linear Relation Analysis. Application To The Verification Of Reactive Systems’. *Formal Methods in System Design* **23**(1), 5–37, 2003.
26. Maraninchi, F. and Y. Rémond. ‘Argos: an Automaton-Based Synchronous Language’. *Computer Languages* (27), 61–92, 2001.
27. Matthaikutty, D., D. Berner, H. Patel, and S. Shukla. ‘SystemCXML’, 2005. `http://systemcxml.sourceforge.net/`.
28. McMillan, K. L. ‘Symbolic Model Checking’. Ph.D. thesis, Boston, 1993.
29. Moy, M. ‘Techniques and Tools for the Verification of Systems-on-a-Chip at the Transaction Level’. Ph.D. thesis, INPG, Grenoble, France, 2005.
30. Moy, M., F. Maraninchi, and L. Maillet-Contoz. ‘LusSy: A Toolbox for the Analysis of Systems-on-a-Chip at the Transactional Level’. In: *International Conference on Application of Concurrency to System Design*. pp. 26–35, 2005a.
31. Moy, M., F. Maraninchi, and L. Maillet-Contoz. ‘Pinapa: An Extraction Tool for SystemC descriptions of Systems-on-a-Chip’. In: *EMSOFT*. pp. 317–324, 2005b.
32. Müler, W., W. Rosentiel, and J. Ruf. *SystemC Methodologies and Applications*, Chapt. 2. Kluwer, 2003.
33. PROVER Technology. ‘Prover Plug-in^{TM} for SCADE^{TM}’, 1989–2005. `http://www.prover.com/products/ppi/sl.xml`.
34. Ruf, J., D. Hoffmann, T. Kropf, and W. Rosenstiel. ‘Simulation-guided property checking based on multivalued ar-automata’. In: *Design, Automation and Test in Europe*, pp. 742–748, 2001.
35. Snyder, W. ‘SystemPerl Home Page’, 2001–2005. `http://www.veripool.com/systemperl.html`.
36. SPIRIT Consortium: 2003. `http://www.spiritconsortium.com/`.
37. synopsys. Discussion with the team developing the front-end of the CoCentric suite before writting Pinapa, 2003.
38. Synopsys Inc. ‘CoCentric System Studio’, 2005. `http://www.synopsys.com/products/cocentricstudio/cocentricstudio.html`.
39. van Heesch, D. ‘Doxygen’, 1997–2005. `http://www.doxygen.org`.
40. Villar, J. C. ‘sc2v: SystemC to Verilog Synthesizable Subset Translator’, 2004–2005. `http://www.opencores.org/projects.cgi/web/sc2v/overview`.