PINAPA: An Extraction Tool for SystemC Descriptions of Systems-on-a-Chip

Matthieu Moy[*] [†] Matthieu.Moy@imag.fr
Florence Maraninchi[*] Florence.Maraninchi@imag.fr
Laurent Maillet-Contoz[†] Laurent.Maillet-Contoz@st.com

ABSTRACT

SystemC is becoming a de-facto standard for the description of complex systems-on-a-chip. It enables system-level descriptions of SoCs: the same language is used for the description of the architecture, software and hardware parts.

A tool like PINAPA is compulsory to work on realistic SoCs designs for anything else than simulation: it is able to extract both architecture and behavior information from SystemC code, with very few limitations. PINAPA can be used as a front-end for various analysis tools, ranging from "superlint" to model-checking. It is open source and available from `http://greensocs.sourceforge.net/pinapa/`. There exists no equivalent tool for SystemC up to now.

Categories and Subject Descriptors

B.6.3 [**Logic Design**]: Design Aids—*Hardware description languages*

General Terms

Languages

Keywords

SystemC, front-end, parser, static, dynamic, elaboration

1. INTRODUCTION

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 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 software development or preliminary architectural exploration.

SystemC [13] has been designed to meet these requirements. It provides *system-level* descriptions of SoCs. A SystemC program has *modules*, *signals* to model the low-level communication and synchronization of the system, and a notion of *simulation time*. SystemC modules and programs are made of several processes that run *"in parallel"*, according to a scheduling policy that is part of the SystemC library specifications. 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* [12]. The architecture of a SoC, the code describing the activity of its software parts, and the description of its hardware parts, can be described in SystemC.

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 [4]. In this type of approach, the definition of the description language is part of the game. The language can be defined formally, and tailored to allow easy connections to validation tools. 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. An IEEE standardization is ongoing. SystemC is currently used by major silicon companies like Intel, STMicroelectronics, Philips, Texas Instruments, ...

Developing analysis tools for SystemC is therefore interesting. SoC case-studies written directly in SystemC can be obtained from industry, and used to validate an analysis approach. The use of any academic formal language as the core of an analysis tool would imply that any case study is first *translated* into this language, before analysis can be per-

[*]Verimag. Centre équation - 2, avenue de Vignate, 38610 GIÈRES — France
[†]STMicroelectronics, HPC, System Platforms Group. 850 rue Jean Monnet, 38920 CROLLES — France

EMSOFT'05, September 19–22, 2005, Jersey City, New Jersey, USA.

formed. If the transformation is manual, then the validity of the results obtained, with respect to the original system, can be questioned.

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

Static and dynamic information in SystemC

SystemC, like several programming languages or runtime 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. In many contexts, one will need to get some static information on the program. This is useful for example to synthesize a lower-level view of the program, to visualize it graphically, to generate some documentation automatically, or to connect to formal verification tools.

Pinapa: requirements

This paper presents PINAPA (For **P**inapa **I**s **N**ot **A** **PA**rser), a SystemC front-end. Our requirements when we started writting 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.

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 useable through GCC and SystemC API, plus some additional PINAPA-specific functions.

Structure of the paper

Section 3 presents SystemC. Section 4 explains the principles of our tool, its limitations, and its uses. Section 5 gives more details about practical problems and their solutions. Section 2 lists existing SystemC tools and compares them with our requirements; it also mentions tools from other application domains using a comparable approach. section 6 is the conclusion.

2. RELATED WORK

2.1 Existing SystemC Tools

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

SystemPerl [15] 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 in section 1.

Synopsys™ developed a SystemC front-end that has successfully been included in products like CoCentric SystemC Compiler and CoCentric System Studio [18][17]. It parses the constructors and the main function, as well as the body of the modules with the EDG [7] C++ front-end, and infers the structure of the program from the syntax tree of the constructors. SystemCXML [10] seems to use the same approach, using doxygen's [19] 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 recently developed a SystemC front-end called ParSyC [8]. The approach is similar to the one of Synopsys™, 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 [20] is also a SystemC synthesizer, built with the same approach. 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 [5] 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.

2.2 Other combinations of static and dynamic analyzers

2.2.1 Reverse engineering

The combination of static and dynamic information extraction is used in other domains. In particular, several reverse engineering techniques use a comparable approach: in [9], the dynamic analysis is used to refine the result of the static analysis and eliminate false positive in design pattern recognition, and in [14], the static and dynamic information are combined to generate UML diagrams. In both cases, the difference is that the dynamic information extracted relates to the behavior of the program, and not to an elaboration phase as we are doing in SystemC.

2.2.2 Graphical User Interfaces

The most similar works are to be found in the domain of Graphical User Interfaces. Most GUI toolkits have this notion of elaboration phase where graphical elements are built and displayed, followed by the behavior of the program which consists in waiting for an event and executing the corresponding action. The difference with hardware modeling is that GUI elements can be created dynamically. Many tools and Integrated Development Environments need to deal with the static part of the interface (in particular, to provide a graphical editor for it). The approach followed by most of them is the one presented in section 4.3.2, defining a dedicated language to describe the interface, and providing a code generator or a dynamic loader.

3. THE SYSTEMC "LANGUAGE"

SystemC provides a set of components (some usable out-of-the-box, and some as base classes to be derived and implemented), and an execution kernel.

Figures 1 gives an example of a SystemC program, used to illustrate the explanations of the following sections. A SystemC program is made of a set of modules. Each module may contain one or more processes. The process code is an ordinary C++ function (like `code1` line 8 and `code2` line 24). In a module, the user declares that such a function is to be used as a process, by using two macros: `SC_THREAD (...)` and `SC_METHOD (...)` (the "threads" execute the code in a infinite loop, while the "methods" are executed periodically in null time). Communication internal to a module can be done in many ways (shared variables, events, etc.), but inter-module communication should be limited to SystemC communication primitives: *ports* and *channels*.

A module contains *ports*, which are the interface to the external world. The ports of different modules are bound

```
1  #include "systemc.h"
2  #include <iostream>
3  #include <vector>
4
5  struct module1 : public sc_module {
6    sc_out<bool> port;
7    bool m_val;
8    void code1 () {
9      if (m_val) {
10       port.write(true);
11     }
12   }
13   SC_HAS_PROCESS(module1);
14   module1(sc_module_name name, bool val)
15     : sc_module(name), m_val(val) {
16     // register "void code1()"
17     // as an SC_THREAD
18     SC_THREAD(code1);
19   }
20 };
21
22 struct module2 : public sc_module {
23   sc_in<bool> ports[2];
24   void code2 () {
25     std::cout << "module2.code2"
26               << std::endl;
27     int x = ports[1].read();
28     for(int i = 0; i < 2; i++) {
29       sc_in<bool> & port = ports[i];
30       if (port.read()) {
31         std::cout << "module2.code2: exit"
32                   << std::endl;
33       }
34       wait(); // wait with no argument.
35               // Use static sensitivity list.
36     }
37   }
38   SC_HAS_PROCESS(module2);
39   module2(sc_module_name name)
40     : sc_module(name) {
41     // register "void code2()"
42     // as an SC_METHOD
43     SC_METHOD(code2);
44     dont_initialize();
45     // static sensitivity list for code2
46     sensitive << ports[0];
47     sensitive << ports[1];
48   }
49 };
50
51 int sc_main(int argc, char ** argv) {
52   bool init1 = true;
53   bool init2 = true;
54   if (argc > 2) {
55     init1 = !strcmp(argv[1], "true");
56     init2 = !strcmp(argv[2], "true");
57   }
58   sc_signal<bool> signal1, signal2;
59   // instantiate modules
60   module1 * instance1_1 =
61     new module1("instance1_1", init1);
62   module1 * instance1_2 =
63     new module1("instance1_2", init2);
64   module2 * instance2 =
65     new module2("instance2");
66   // connect the modules
67   instance1_1->port.bind(signal1);
68   instance1_2->port.bind(signal2);
69   instance2->ports[0].bind(signal1);
70   instance2->ports[1].bind(signal2);
71   sc_start(-1);
72 }
```

Figure 1: Example of a SystemC Program.

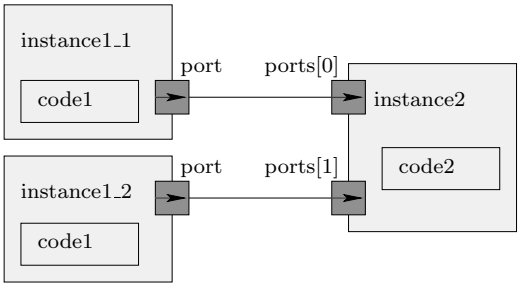

Figure 2: Graphical view of the program

together with communication channels to enable communication. SystemC provides a set of communication interfaces such as `sc_signal` (synchronous signals), and abstract classes to let the user derive his own communication *channels*. In the example, both instances of `module1` write on a signal the value `m_val` that they have been passed as an argument, which is received by `module2`.

The SystemC kernel executes the `sc_main` function. At the beginning of the execution (lines 52 to 70 on Figure 1) is the *elaboration* phase. Components are instantiated in the usual way for C++ objects. In the pieces of code of Figure 1, we have two definitions of modules, one of which is instantiated twice. The program is represented graphically in Figure 2. Elaboration ends with a call to the function `sc_start()` that hands the control back to the SystemC kernel (line 71). 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.

With the objective of developing a tool able to extract architecture and behaviour information from a SystemC program, it is important to note that a SoC designer may write general C++ code. In the example, the actual architecture depends on the command line parameters `init1` and `init2` (see lines 61, 63). Moreover, the example makes use of templates and macros. Those constructs are commonly used in the platforms on which we tried PINAPA.

4. PINAPA PRINCIPLES, LIMITATIONS AND USES

4.1 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 Figure 3. 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 PINAPA's 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 (ELAB) that stays in memory at the end of the elaboration phase;

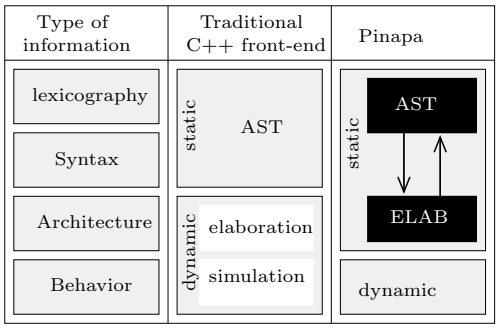

Figure 3: Static and Dynamic information in a SystemC program

Figure 4 describes the dataflow 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).

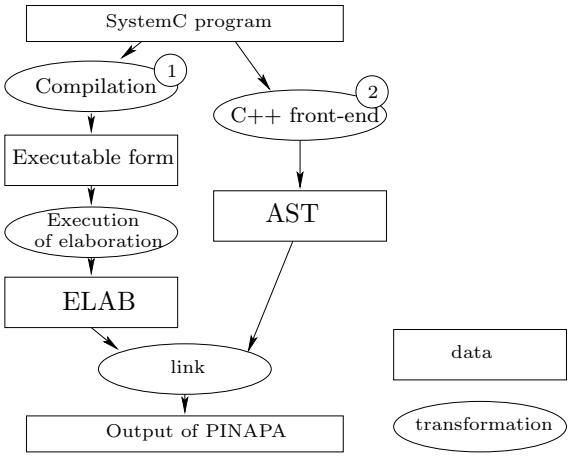

Figure 4: Data-flow of PINAPA

Note that the source code is parsed by a C++ front-end twice: first, to compile the elaboration phase (step (1) in Figure 4), and then to get the AST, which, in our case, contains some address offset information, dependent on the binary interface (ABI) (step (2)). The address offsets obtained in both cases must be consistent, so the C++ front-ends of (1) and (2) must be ABI-compatible (in particular, they can 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.

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 Figure 1, 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™) 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 use the macro or its expanded version doesn't have any influence on PINAPA. 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 Figure 4). 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.

4.2 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. This is the case of `port`, declared line 29 in Figure 1. 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 (this is the case in the instruction `ports[1]` line 27 of the example). 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 index by transforming the code to eliminate non-constant indexes, while preserving the semantics of the program. In the example, the transformation would unroll the `for` loop or transform `ports[i]` into

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

PINAPA being open-source, such 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 know 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-templates base class in SystemC and the TLM library).

4.3 Other possible approaches

4.3.1 Using a C++ Interpreter

An interesting option would be to modify an existing C++ interpreter like UnderC [6]. 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 runtime information. We are not aware of any such interpreter.

4.3.2 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 [16] 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.

4.4 Pinapa: Current and Future Uses

We currently use PINAPA as a front-end for our formal verification tool LUSSY [11]. 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.

We are currently developping a vizualization tool for SystemC using PINAPA: Reading a SystemC program, it will generate another representation useable by a visualization tool (either the `dot` format from graphviz or SPIRIT). This is a very simple use of PINAPA since it only use the ELAB part of the information extracted. Our medium-term plans include the development of a lint tool for SystemC and our TLM methodology. The tool will have 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).

5. 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 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 Figure 2, 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 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.*

SystemC Objects. Each SystemC object (ports, modules, ...) contains the necessary information about the binding. In the example above, the port `port` of `instance1_1` contains a pointer to the signal `signal1`, which itself gives the list of connected ports (`ports[0]`).

The AST represents the bodies of the processes. For example, the `if` statement line 9 in Figure 1 would be represented as in Figure 5. PINAPA will make the link between the AST of the port `port` and its instances in ELAB.

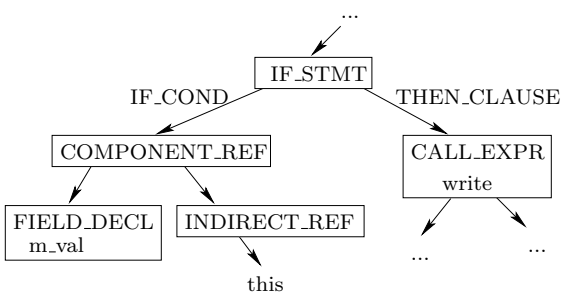

Figure 5: Abstract Syntax Tree for an `if` statement

Concretely, PINAPA first launches the elaboration of the program. We use a slightly modified version of SystemC, in which we redefined in particular 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 5.1, 5.2 and 5.3 below detail some interesting problems raised by this phase. Section 5.4 summarizes the architecture of the tool.

5.1 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, look for the AST of a method with no argument whose class name and function name match the ones in the process handler, and add a pointer to this AST in it.

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.

5.2 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.

5.2.1 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 for a `wait` with no argument. In the example of Figure 1, the `wait` statement line 34 has no argument. The list of sensitivity used is therefore the static list built during elaboration (lines 46 and 47): PINAPA will attach the list `{*ports[0], *ports[1]}` to this statement.

5.2.2 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 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 therefore actually a hash table: (AST, process handler) $\longrightarrow$ SystemC object. For instance, the port referred to line 9 in the example, and in the AST of Figure 5 has two instances, but the pair (AST of the port, process handler for `instance1_2->code1`) uniquely identifies the port `instance1_2->port`.

We describe 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.

5.2.2.1 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 to `write(this->port, x);` by GCC's front-end. Now, here is the bad joke: this code is converted to `write(*(this +` *offset_of_port*`), x);` where *offset_of_port* is a literal numerical constant. At this point, we are still in GCC's 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.

5.2.2.2 Other objects.

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

5.3 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 5.2.2.1 is therefore not applicable. We could compute the offset from the AST (GCC does this anyway, later in the compilation flow), but we chose a different approach, that does not require this computation.

Since we have here both 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. This way, 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 calling can not.

5.4 Function Call Graph

The resulting function call graph in PINAPA is somewhat complex (Figure 6), but will be made clearer by the end of this section.

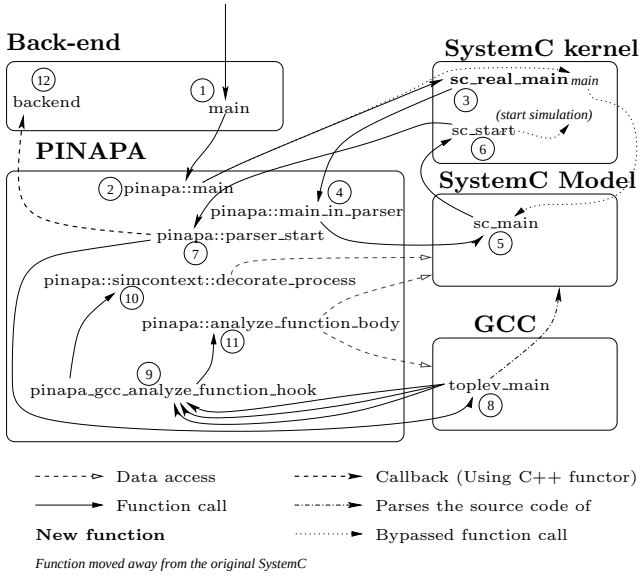

Figure 6: Architecture of PINAPA

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 Figure 6.

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.

5.5 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.

6. CONCLUSION

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 paper is the way to make the link between the source code representation and the runtime 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.

7. REFERENCES

[1] Actis Design, LLC. AccurateC^{TM} Rule Checker. `http://www.actisdesign.com/`.

[2] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers — Principles, Techniques, and Tools*. Addison-Wesley, Reading, MA, USA, 1986.

[3] ARM Limited. AHB Example Amba SYstem technical reference manual, `http://www.arm.com/pdfs/DDI0170A.zip`.

[4] F. Balarin, 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*, pages 228–273. Springer-Verlag, 2002.

[5] Cadence Design Systems. NC-SystemC. `http://www.cadence.com/products/functional_ver/nc-systemc/`.

[6] S. Donovan. The UnderC development project. `http://home.mweb.co.za/sd/sdonovan/underc.html`.

[7] Edison Design Group. Compiler front ends. `http://www.edg.com/`.

[8] G. Fey, D. Groe, 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/04sasimi_parsyc.pdf`.

[9] D. Heuzeroth, T. Holl, and W. Löwe. Combining static and dynamic analyses to detect interaction patterns, 2002. In IDPT, 2002. (submitted to). Welf Löwe and Markus Noga.

[10] D. Matthaikutty, D. Berner, H. Patel, and S. Shukla. SystemCXML. `http://systemcxml.sourceforge.net/`.

[11] M. Moy, 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*, June 2005.

[12] W. Müler, W. Rosentiel, and J. Ruf. *SystemC Methodologies and Applications*, chapter 2. Kluwer, 2003.

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

[14] C. Riva and J. V. Rodriguez. Combining static and dynamic views for architecture reconstruction. In *European Conference on Software Maintenance and Reengineering*, pages 47–56. IEEE Computer Society Press, 2002.

[15] W. Snyder. SystemPerl home page. `http://www.veripool.com/systemperl.html`.

[16] SPIRIT Consortium. `http://www.spiritconsortium.com/`.

[17] 2003. Discussion with the team developing the front-end of the CoCentric suite before writting Pinapa.

[18] Synopsys Inc. CoCentric System Studio. `http://www.synopsys.com/products/cocentric_studio/cocentric_studio.html`.

[19] D. van Heesch. Doxygen. `http://www.doxygen.org`.

[20] J. C. Villar. sc2v: SystemC to verilog synthesizable subset translator. `http://www.opencores.org/projects.cgi/web/sc2v/overview`.