PinaVM: a SystemC Front-End Based on an Executable Intermediate Representation

Kevin Marquet and Matthieu Moy

Verimag
Centre quation - 2, avenue de Vignate 38610 Gires - FRANCE

April 12, 2010

Abstract:

PinaVM: a SystemC Front-End Based on an Executable Intermediate Representation

Kevin Marquet and Matthieu Moy

Verimag Research Report n^o TR-2010-8

April 12, 2010

PinaVM: a SystemC Front-End Based on an Executable Intermediate Representation

Kevin Marquet and Matthieu Moy

Verimag
Centre quation - 2, avenue de Vignate 38610 Gires - FRANCE

April 12, 2010

Abstract

SystemC is the *de facto* standard for modeling embedded systems. It allows system design at various levels of abstractions, provides typical object-orientation features and incorporates timing and concurrency concepts. A SystemC program is typically processed by a SystemC front-end in order to verify, debug and/or optimize the architecture. Designing a SystemC front-end is a difficult task and existing approaches suffer from limitations. In this paper, we present a new approach that addresses most of these limitations. We detail this approach, based on an executable intermediate representation. We introduce PinaVM, a new, open-source SystemC front-end and implementation of our contributions. We give experimental results on this tool.

Keywords: SystemC, front-end, parser

Reviewers: Florence Maraninchi

Notes:

How to cite this report:

@techreport { verimag-TR-2010-8,
title = { PinaVM: a SystemC Front-End Based on an Executable Intermediate Representation},
author = { Kevin Marquet and Matthieu Moy},
institution = { Verimag Research Report },
number = {TR-2010-8},
year = { 2010},
note = { }
}

1 Introduction

SystemC is a C++ class library which facilitates modeling of systems at many different levels of abstractions ranging from functional description to cycle-accurate modeling. Ability to design at higher abstraction levels is valuable due to increasing complexity of system design. SystemC is a widely accepted system description language and has been approved as a standard by the IEEE consortium [1]. Being a C++ library, SystemC provides typical object-oriented features which make the task of system design easier and faster. It also offers the concepts of timing and concurrency which are essential for hardware modeling.

SystemC is primarily used for simulation. A typical C++ compiler suffices to generate an executable that performs simulation. However, during system design process, a SystemC program may have to be processed for other purposes, e.g. for generating a graphical layout of the system. For such applications, initial processing of SystemC program is done by *front-end* tools.

Writing a SystemC front-end is different from writing a front-end for a *language*. Traditional techniques such as lex/yacc are not sufficient since SystemC is indeed a *library*, that builds an important part of the program at run-time: a SystemC program describes a set of communicating modules, connected together through communication channels, and the layout of the modules and channels is build after the program is started, in a phase called the *elaboration phase*. After the elaboration phase is over, the program calls the function `st_start()`, which starts the simulation. A SystemC front-end must therefore consider the elaboration code in a particular way: the relevant information is not the code itself, but the data-structure it builds, that is, the *architecture*.

There exists a myriad of SystemC front-ends, but we will see that none of them are really satisfactory. Most have severe limitations, and the least limited are either unavailable or hard to install and use. Also, none of the existing tools provide a Static Single Assignment (SSA) [7] form, which is gaining popularity in the compiler's community, and also proved its efficiency for formal verification [4, 11].

The contributions of this paper are the following:

- A novel approach for the development of SystemC front-ends, relying on a *just-in-time compiler* (JIT) compiler to execute fragments of the program code on-the-fly during the analysis,
- The description of a tool called PinaVM applying this approach, that has strictly fewer limitations than each of its predecessors,
- The first SystemC front-end providing a simple SSA form as output.

The tool itself is open-source, and available from `http://gitorious.org/pinavm`.

We first introduce SystemC in section 2. Then, we recall in section 3 the motivations and challenges in the development of a SystemC front-end. Section 4 presents our approach, and our solution to these challenges. Section 5 presents briefly the LLVM compiler infrastructure, on which is based our tool. Section 6 details the technical aspect of our solution, and section 7 gives the experimental results.

2 SystemC

A SystemC program defines an *architecture*, i.e. a set of components and connections between them. Components have a behavior defined by one or several processes. The SystemC library provides different mechanisms allowing synchronization between processes.

Processes explicitly suspend themselves through two kinds of `wait` instructions: a process may wait for some time to elapse, or for an event to occur. So, synchronization of processes is based on the following SystemC specific constructs:

wait(int) Stops executing the current process, yields back the control to the scheduler and makes the current process to wait the given duration.

wait(event) Stops executing the current process, yields back the control to the scheduler and makes the current process to wait for the specified event to occur.

event.notify() Make processes waiting for the specified event eligible. This does not stop the execution of the current process.

As an example, figure 1 shows a SystemC module containing one process waiting for an event, and another notifying it.

```
SC_MODULE(mytop) {
   // SystemC event, instantiated
   // as usual C++ object.
   sc_event e;

   // Process bodies
   void myFctP() {
      ...
      wait(e);
      ...
   }
   void myFctQ() {
      ...
      e.notify();
      ...
   }

   // Constructor
   SC_CTOR(mytop) {
      SC_THREAD(myFctP);
      SC_THREAD(myFctQ);
   }
}
```

Figure 1: A basic SystemC module

Other primitives can be used such as delayed notification, but we do not consider them in this paper as it does not change the complexity of designing a SystemC front-end: we are interested in the syntax of constructs, and the way the syntax refers to the architecture of the platform, but the front-end does not need to be aware of the purely run-time semantics.

Processes may also communicate shared variables through and through ports connected by channels, thanks to the following primitives:

port.read() read in a port

port.write(data) write the given data to the port.

A module can therefore communicate with another by writing data to a port. The connection between ports is made by a *channel* (*i.e.* a class inheriting from the `sc_interface` class defined by SystemC). The SystemC library defines some channels like `sc_signal` (basically a buffer of one place), or `sc_fifo` (a FIFO), but channels can be defined by the programmer. As an example, figure 2 shows two modules, each one defining a process and a port. The two ports are bound together by a `sc_signal` holding an integer value. The two processes communicate an integer value through these ports.

3 SystemC Front-ends

3.1 Motivations

Although a plain C++ compiler can be sufficient to run a program written in SystemC, many applications other than simulation require a dedicated front-end.

These applications include hardware synthesis, which is possible on a restricted subset of SystemC; optimized compilation (for example, Scoot [5] shows great performance improvement by doing some static scheduling, many virtual function calls can be statically resolved at the beginning of simulation, ...); symbolic formal verification, which requires some reasoning about the source code of the program; source code instrumentation (required by some run-time verification [9] and introspection [8] tools); advanced debugging support, and visualization. See [15] for a more comprehensive discussion on the subject.

Our initial motivation for writing PinaVM was to be able to apply the algorithm described in [4, 11] to translate a Static Single Assignment (SSA) intermediate code into a synchronous language for verification.

```
SC_MODULE(reader) {
   sc_in<int> in;

   void read() {... int d = in.read(); ... }

   SC_CTOR(reader) { SC_THREAD(read); }
}

SC_MODULE(writer) {
   sc_out<int> out;

   void write() { ... out.write(42); ... }

   SC_CTOR(writer) { SC_THREAD(write); }
}

int sc_main(int argc, char ** argv)
{
   // Instantiate modules and signal
   Reader readR; Writer writeR;
   sc_signal<int> channel;

   // connect them together through a signal
   readR.in.bind(channel);
   writR.out.bind(channel);

   sc_start(); // start simulation
}
```

Figure 2: Communication between two modules

Indeed, the translation scheme proposed leads to very good performance in the proof engine, but has never been applied to actual SystemC code by lack of a front-end (the implementation uses the SSA form of GCC, but has no knowledge of the architecture, hence cannot really deal with SystemC code). Before PinaVM, no SystemC front-end were based on SSA. Using LLVM was initially a way to get an SSA form as output to the front-end, but revealed to be an excellent choice, since it lead us to a new approach, getting rid of most of the limitations of its predecessors, without introducing new ones.

3.2 Issues In Developing SystemC Front-ends

Developing a frond-end for SystemC involves some non-trivial challenges. First, SystemC being a C++ library, all C++ features must be supported by the front-end. This can be achieved by either writing a parser from scratch using the language grammar (but this is known to be require a tremendous effort), or by using an existing C++ front-end such as GCC, LLVM, EDG etc.

Moreover, while traditional compiler front-ends consider static information to be the ones available at compile-time (lexicography, syntax, typing, ...), a SystemC front-end has to consider the elaboration phase as something static: it builds the architecture of the platform, which is an essential part of the role of a SystemC front-end, and does not change during simulation. The semantics of a program depends on the architecture, as it defines the links between modules and the communications between processes. Consider a code fragment in Figure 2. A regular C++ parser can only parse the program and generate an intermediate representation of the code. Modules `readR` and `writeR` are connected by signal `channel`. A C++ parser does not derive this information: this necessitates the need for some extra processing to determine the interconnections between modules. Some existing solutions use static analysis, but the architecture related information may not be derivable from the static analysis of code, since some components and their interconnections may be generated using dynamic data. As an example, Figure 3 shows a SystemC program where modules are instantiated in a loop.

```
struct module1 : public sc_module {
   ...
}

int sc_main(int argc, char **argv) {
   module1 * m[MAX];
   for(i = 0; i < n; ++i) {
      m[i] = new module1();
   }
}
```

Figure 3: Example illustrating architecture dependent on dynamic information

So, on one hand one has to set up solutions to build the architecture, on the other hand one has to detect the communications between processes in their behaviors. And most difficult part, the link has to be done, in order to establish who is talking to whom in a communication.

Communications through channels provided by the SystemC library are quite easy to detect, as they are mainly performed through a set of functions given in the LRM [1]. In the case of user-defined channels, it is much more difficult, and no existing work is able to handle this case. Even in the simplest case, the hard point is not to detect that a communication happens, but to know which module or process is talking to which; and possibly which data. Those issues are more detailed in [15].

3.3 Related works

The idea of a SystemC front-end is not new: [15] lists 13 existing front-ends, and can serve as a reference for an in-depth comparison.

Several front-ends (including sc2v, KaSCPar, ParSyC, SystemPerl) are based on traditional compilation techniques like lex/yacc, with a dedicated grammar for SystemC. KaSCPar [2] is the most widely used, but our experience with it is discouraging: it crashes on the examples provided in its own distribution, and our questions to the authors remained unanswered. Given the complexity of the C++ language, it is not surprising to see that these tools have considerable limitations with respect to C++ constructs. Other tools reuse an existing C++ front-end (Scoot, SystemCXML are based on limited C++ front-ends, and commercial tools like CoCentric compiler and Semantec Design's front-ends usually use EDG, which is a very complete C++ front-end), but most of them extract the architecture of the platform by doing some static analysis on the elaboration code, and fail as soon as the elaboration code is non-trivial.

Pinapa [18] is our main source of inspiration. One of the key idea is to *execute* the elaboration code, and consider the state of memory at the end of elaboration as the architecture of the platform. The result is a tool developed with little effort, still having very few limitations. It was initially written as part of the LusSy [17] verification chain.

PinaVM is a new tool, borrowing ideas to Pinapa, and introducing new ones, which make it more general and easier to use. In particular, we show the benefit of relying on the LLVM infrastructure and its JIT compiler for the development of a SystemC front-end.

4 PinaVM: Goals and Key Ideas

Our primary motivation for writing PinaVM was to provide means to perform efficient formal and symbolic verification of SystemC programs. We've seen that designing a complete SystemC front-end is a difficult task and existing works suffer from different limitations. PinaVM is our attempt to address these limitations. Another goal is to provide a SystemC-specific optimizing compiler, although we did not experiment in this field yet. We are not concerned by debug and visualization purposes, although PinaVM can serve as a basis for such works.

Basically, PinaVM is SystemC front-end and therefore allows to obtain an abstract representation from a SystemC program. It differs from previous works in the approach (and therefore the subset of SystemC accepted as input), and the intermediate representation provided as output.

We reviewed the limitations of existing SystemC front-ends in [15]. The two main sources of limitations are the complexity of the C++ language, and the difficulty to extract the architecture information. Pinapa was a first attempt to tackle these difficulties, by using a real C++ compiler (GCC), and executing the elaboration phase to retrieve the architecture information. PinaVM keeps what we think are the good ideas behind Pinapa: reuse a C++ front-end, end execute the elaboration, which gives the interesting properties of Pinapa: very good support for C++ advanced features, and support for arbitrarily complex code in the elaboration phase (including code depending on a configuration file, command-line arguments, ...). These two points alone makes the subset of SystemC managed by PinaVM out of the reach of grammar-based front-end like KaSCPar, and of other tools supporting only basic elaboration code. Hence, we focus our comparison on Pinapa, which is the only existing work having a comparable support for complex SystemC code. Still, Pinapa had a number of drawback relative to GCC internal, the way it represents control flow (GCC CFG) and most of all its limited approach to retrieve information into this CFG.

4.1 GCC internals

In particular, it uses the internal API of GCC, which wasn't designed to be modular and reusable as a front-end. Hence, the installation and use of Pinapa is difficult. For example, since a single instance of GCC can't parse two C++ files, Pinapa doesn't manage separate compilation properly.

Using the LLVM [12] compiler infrastructure instead of the internal API of GCC removes the main technical drawback of Pinapa: LLVM is designed to be usable as a library, and we use it as such. Although its API is not fully stable, it is clean, and the migration from a version of LLVM to another is a painless task (we already did it for the migration from version 2.5 to 2.6, and then to 2.7 easily). LLVM provides us many tools that can be used by PinaVM, or together with PinaVM, to perform various tasks in a simple way. Users of SystemC front-ends written from scratch usually have to stick to what the front-end provides, while we keep the convenience of a complete and modern compiler infrastructure available to the user. For example, separate compilation was an open problem with Pinapa, but it's made trivial by the command `llvm-link`. Using a public API also has an interesting consequence: PinaVM does not need to patch LLVM (while Pinapa relied on a patched version of GCC). The user can install LLVM as any other library (with his favorite package manager for example).

4.2 A low-level representation

Also, Pinapa's output format was mostly GCC's abstract syntax tree (AST) decorated with SystemC-specific information. One issue with this is that this AST contains all C++ syntactic sugars. For example, a user of Pinapa would get different ASTs for `x = x + 1;`, `x++;` and `++x;` (GCC 3.4.6 used by Pinapa has 224 different kinds of nodes for C++ ...). A lower-level representation, converting these syntactic sugars into canonical forms is desired. Additionally, experiments have shown the benefits of the Static Single Assignment (SSA) form for the translation to synchronous languages [4, 11]. The intermediate format of LLVM (LLVM bitcode) has all these properties: it is SSA, doesn't have more constructs than needed, and the available constructs are as simple as they can be, while retaining the basic typing information. Technically, it has other advantages like having a human-readable form, a binary file format, and a data-structure representation, with the tools to convert one of these three forms into another.

4.3 The key point: retrieving information in the AST

Unfortunately, the benefits of LLVM and its SSA bitcode also come with a number of challenges. The algorithm of Pinapa to link the architecture information and the AST is not applicable on LLVM's bitcode, and the *approach* has to be re-thought completely. For example, the statement `port.write(42);`, translated into an AST, is a node of type `CALL_EXPR` in GCC's AST. It is easy to find the value `42` in the children of this node, and Pinapa knows where to find the information to compute a pointer to `port` in the tree. This is a limitation of Pinapa as it only allows to retrieve information stored in trees of that particular shape.

However, in LLVM's bitcode, the computation of the arguments is done with a sequence of statements, prior to the call expression. The approach of Pinapa, relying on a fixed form of tree, doesn't apply here.

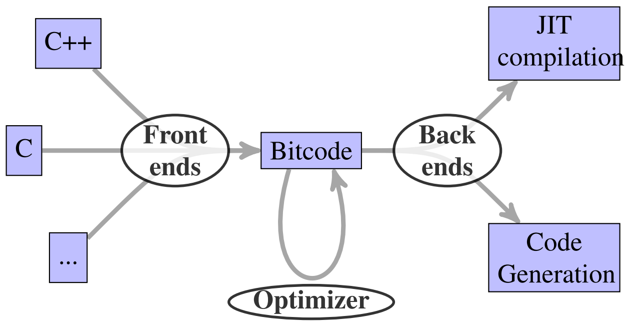

Figure 4: The LLVM Infrastructure

Indeed, during the early stages of development of Pinapa, experiments were made with the SSA branch of GCC (which became GCC 4.0 later), that provides an intermediate representation similar to the one of LLVM. The conclusion was that the SSA form had a lot of benefits, but that its low-level nature made the treatment of SystemC constructs too hard, if at all possible, and the experiment was abandoned.

This paper proposes a solution to this problem. Our proposal is to push the idea of "execution" one step further. We do not only execute the elaboration phase, but also small portions of code that are used to build the arguments of functions. In the example above, `port.write(42);`, the piece of code to build the implicit argument `port` computes a pointer to `port` using a pointer to the current module, **`this`**. Since the later is known after the elaboration phase, we can execute this piece of code with the actual value of **`this`** as input, and get a pointer to the port involved in the `port.write(42);` statement. The key point here is the executability of the bitcode. Technically, we rely on the JIT (Just In Time compilation) capabilities of LLVM for this task.

This bidirectional link between the representation of the source code and the dynamic data-structure can be seen as adding some reflexivity to SystemC: the code of PinaVM, and of its back-end, runs after the elaboration, hence, during the execution of the program it analyzes. The functionality provided by PinaVM are essentially "given a process handle, which contains a pointer to a compiled function, get a pointer to the intermediate representation of this function", and "given a reference to an object in the intermediate representation, get a pointer to the actual object". Indeed, tools targeting the addition of reflexivity features to SystemC use a similar approach [8]. Our solution is partly based on LLVM.

5 LLVM: Low Level Virtual Machine

5.1 The LLVM infrastructure

LLVM [12] is a compiler infrastructure. Its central point is an language-independent Intermediate Representation (the LLVM bitcode, or LLVM IR), generated by front-ends from source languages. The bitcode can be modified by optimization passes, and can be used by various back-ends to either generate code statically, or perform Just-In-Time compilation, or even bitcode interpretation (See Figure 4 for an illustration). Among the qualities of LLVM, we can cite:

- The language-independant intermediate representation is a bitcode in Static Single Assignment (SSA) form.
- The LLVM IR is not an internal format devoted to change with each new version of LLVM. It is a well-documented format that can be exploited by different tools of the LLVM infrastructure (for example, loading a bitcode file is trivial using the LLVM library).
- It is Open Source and supported by Apple, two points which guarantee the durability of this compiler.
- Its license is very permissive, and does not impact the user's code (as opposed to the GPL, used by GCC, used by Pinapa).

C++ can be compiled to LLVM bitcode by two means. First, the LLVM front-end C++, called `clang`, can be used. However, CLang's support for C++ is still incomplete, and is not able to parse complicated programs such as SystemC libraries. Second, a GCC back-end for LLVM bitcode exists: `llvm-gcc`. llvm-gcc can compile any C or C++ program supported by GCC to LLVM bitcode.

5.2 The LLVM bitcode

The LLVM representation is an SSA based bitcodes format (including about 50 different bitcodes). It is rather low-level, but is completely typed: type of variables, functions' parameters, functions' return are directly available. Pointer types exist and allow to represent C pointers very simply. However, the representation keeps no information related to objects at all. It has no notion of inheritance for instance. It is not a blocking problem, although it complicates things a bit.

```
struct Component : public sc_module {
  int example(int x) {
    int res = x + 42;
    return res;
  }
};
```

(a) Source code in C++

```
define linkonce_odr i32
@_ZN9Component7exampleEi
(%struct.Component* %this, i32 %x) nounwind
{
  entry:
     %"alloca point" = bitcast i32 0 to i32
     %0 = add nsw i32 %x, 42
     br label %return

  return:
     ret i32 %0
}
```

(b) corresponding human-readable bitcode file

Figure 5: A small example of LLVM representation

An LLVM bitcode is a set of *basic-blocks*, starting with a *label* (like `entry:`), and ending with a possibly conditional jump (like `br label %return`). Basic-block contain a sequence of instructions. Instruction usually define *registers* (like `%0`, `%x`, ...) as a function of previously defined registers. Each SSA register is defined once, before being used, and is never reassigned. LLVM keeps the link between uses of a register and its definition (use-def chain). An LLVM instruction contains *arguments*, and the definition of each argument is directly pointed to. That means that, in the sequence `%23 = expr; call fct (%23)`, by parsing the parameters of `fct`, we obtain a direct pointer to the instruction defining it (in the case of a register, the instruction pointed to is the instruction allocating it).

There does not exist a single LLVM representation for a given program, and some optimizations can be made on it.

We notably use an LLVM pass (mem2reg) that maximizes the use of registers instead of memory locations, allowing to reduce the use load/store instructions which complicates static analysis of bitcodes.

6 Dynamic compilation to retrieve static information

We now detail our solution for writing a SystemC front-end. Three difficult parts have to be addressed in this approach. First, how to execute the elaboration phase? Second, how to recognize SystemC constructs in the LLVM IR? Last, how to link them together? These questions are dealt with respectively in sections 6.1, 6.2 and 6.3 below. The solution makes intensive use of just-in-time compilation, which is the main novelty of our approach. Last, section 6.5 describes our own representation.

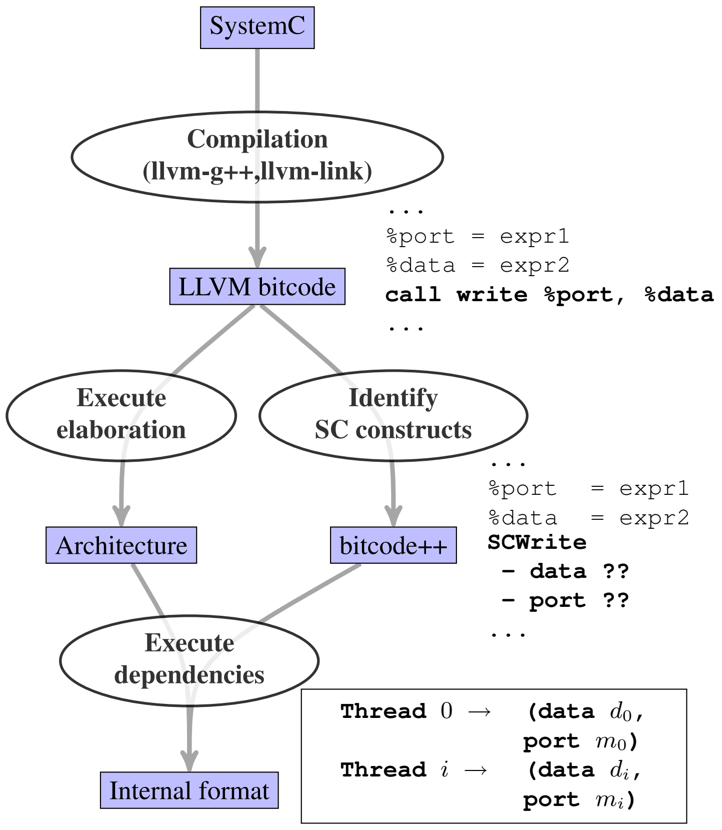

Figure 6: PinaVM: Architecture and Data-Flow

6.1 Executing the Elaboration Phase

The tool PinaVM itself takes an LLVM bitcode file as input. To obtain this file, the user can compile the SystemC program, replacing the usual C++ compiler by `llvm-g++` (in the future, `clang++` should become a viable alternative), and the linker with `llvm-link`.

This bitcode file is loaded by PinaVM, and an execution engine (based on JIT-compilation) executes the elaboration code, hence creating new modules, process handles, ... which are registered in global variables of the SystemC library. We use a slightly modified version of SystemC, in which we inserted a call to a function `pinavm_callback()` within the function `sc_start()`, which normally starts the simulation. Instead of starting simulation, this function comes back in the code of PinaVM, which will then analyze the processes before calling the back-end functions. Figure 6 illustrates the data-flow and the different stages.

In order for this to work, when the user compiles and link his SystemC program (with `llvm-g++` and `llvm-link`), the SystemC library should not be linked with this bitcode file, leaving the SystemC symbols and `pinavm_callback` undefined. Then, during loading, the unresolved symbols it contains are resolved with the defined ones of PinaVM, which contains the complete SystemC library. This way, the global variables of the SystemC library (typically, the list of processes, the list of `SC_MODULE`s, ...) are shared between PinaVM and the loaded program: the list of processes, for example, is filled-in by the program, and later read by PinaVM. When the JIT-ed code calls `pinavm_callback()`, the function is resolved to the function defined in the compiled code of PinaVM.

A consequence of this is that the compiler used to generate the bitcode (i.e. `llvm-g++` currently) has to be ABI compatible with the compiler used to compile PinaVM. Currently, this is not a problem, since `llvm-g++` is based on GCC 4.2, and there have been no ABI change in GCC since version 4.0. In the future, `llvm-g++` will be implemented as a GCC plugin [3], hence, it will be easy to use the same `g++` executable to compile PinaVM and to generate the bitcode. Future changes in the ABI of GCC are therefore not likely to cause any problem (this was indeed a limitation of Pinapa, which can not be compiled with a version of GCC greater or equal to 4.0).

```
% "alloca point" = bitcast i32 0 to i32
% 0 = getelementptr inbounds %"struct.sc_core::sc_inout<int>"* %this, i32 0, i32 0
% 1 = getelementptr inbounds
%     "struct.sc_core::sc_port<sc_core::sc_signal_inout_if<int>,1,SC_ONE_OR_MORE_BOUND>"*
%     0, i32 0, i32 0
% 2 = call %"struct.sc_core::sc_signal_inout_if<int>"*
           @_ZN7sc_core9sc_port_bINS_18sc_signal_inout_ifIiEEEptEv
           (%"struct.sc_core::sc_port_b<sc_core::sc_signal_inout_if<int> >"* %1)
% 3 = getelementptr inbounds %"struct.sc_core::sc_signal_inout_if<int>"* %2, i32 0, i32 1
% 4 = getelementptr inbounds %"struct.sc_core::sc_signal_in_if<bool>"* %3, i32 0, i32 0
% 5 = getelementptr inbounds %"struct.sc_core::sc_interface"* %4, i32 0, i32 0
% 6 = load i32 (...)*** %5, align 4
% 7 = getelementptr inbounds i32 (...)** %6, i32 4
% 8 = load i32 (...)** %7, align 1
% 9 = bitcast i32 (...)* %8 to void (%"struct.sc_core::sc_signal_in_if<bool>"*, i32*)*
call void %9(%"struct.sc_core::sc_signal_in_if<bool>"* %3, i32* %value_)
```

Figure 7: SSA code for sc_in::write(int data)

Another option would be to execute the elaboration phase natively (i.e. compile it with the usual compiler, and link it against PinaVM or load it dynamically, as Pinapa does). Loading a bitcode file and using the JIT compiler have the advantage of requiring only one input file, which is used by PinaVM both to get the architecture and the body of processes. But more importantly, the JIT compiler maintains a bidirectional map between native code and the source bitcode, which allows one to get a bitcode structure from a pointer to function.

One technical difficulty is that the elaboration phase may build objects on the stack, while it is desirable to wrap the front-end into a function call, so that the user of the front-end can call the front-end, and then call the back-end. Currently, objects allocated on the stack are destroyed when the front-end returns, but an alternative would be to launch the elaboration in a separate thread, so that it is launched with its own stack.

6.2 Finding SystemC constructs in LLVM IR

So, on one hand we have an architecture, given by the execution of the elaboration phase. On the other, we have the behavior of processes, given by the compilation of the SystemC program. These behaviors are composed of any C++ code in which some statements correspond to SystemC specific constructs, allowing processes to communicate. In order to establish precisely communications between processes, one first step is to detect these constructs in processes' behavior. These constructs are briefly described in section 2 and are basically synchronization (wait for time, wait for event, notify an event) and communication means (write data to a port, read in a port).

The difficult part here is that a simple line of code in the SystemC program can be compiled to dozens of low-level lines of SSA code in the LLVM IR. Table 7 presents the code generated for a `port.write(int)`. Inlined, such pieces of llvm bitcodes can be hard to isolate. The first thing we do is to compile without inlining, so that the use of SystemC constructs are isolated in a limited set of functions representing. Therefore, the code corresponding to those constructs is compiled once, in the body of a function, and detecting a communication only means to detect calls to these functions.

The LLVM IR includes `call` and `invoke` bitcodes that can be easily found. In addition, although LLVM bitcodes are low-level, all type information remains as well as the (mangled) name of functions. Therefore, constructs are easily found. Constructs currently managed by PinaVM are summed up in table 1, which also gives the name, mangled by GCC, of the corresponding C++ functions. For communication primitives, although one different function exist for each kind of channel and each type of data held by the channel, only the example of accesses to a `sc_signal<int>` is given.

Thereafter, the goal is to detect the calls to that "API". For a `wait(int:t)`, it is rather easy to parse the code and identify a call to function whose mangled name is
`"_ZN7sc_core9sc_module4waitEi"`.

Synchronization	Wait on time	_ZN7sc_core9sc_module4waitEi
	Wait on event	_ZN7sc_core9sc_module4waitERKNS_8sc_eventE
	Notify event	_ZN7sc_core8sc_event6notifyEv
Communication	Write into port	_ZN7sc_core8sc_inoutIiE5writeERKi
	Read into port	_ZNK7sc_core5sc_inIiE4readEv

Table 1: Main SystemC constructs handled by PinaVM

```
%port  = expr
%data  = expr
call myfct %port, %data
```

Figure 8: Getting the value of arguments

However, finding functions calls is not enough, we also need to compute the values given as parameters to functions calls. Indeed, let us consider the instruction `port.write(42)`. Identifying the module called through this instruction requires to know which port is written to. Afterwards, the correspondence between this port and the target is known thanks to the information retrieved at the end of the *elaboration* (see previous section).

In the instruction `port.write(42)`, the compiled code is equivalent to a function call `write(port, 42)`. The value written is not important but in order to identify precisely the communication, it is necessary to identify the `port` parameter. However, obtaining values given as parameters is difficult, as they can be the result of any arbitrary computations. Figure 8 gives a simple example to illustrate this difficulty. Considering that `expr` can be computed in an arbitrarily complex manner, constant propagation made by compilers does not help to figure it out in the general case. We distinguish `port` between the following cases:

- **constant** if the parameter is a constant, it is quite easy to retrieve it.
- **depends on dynamic data** In the general case, it is impossible to obtain the information needed precisely, as values could depend on dynamic content. Different executions of the same piece of code can actually refer to different value of the parameter.
- **only dependant on architecture** If the data we need to know is only dependant on the architecture, it might be possible to get it. Typically, `port.write(42);` is equivalent to `this->port.write(42);` in C++, `port` depends on the value of `this`, which isn't known at compile time, but corresponds to the address of a module, which is known after the elaboration phase. However, this might not be easy, as data can be stored in structures, or accessed through complex control flow. We now detail the algorithm used to retrieve parameters in such a case.

6.3 Linking architecture and CFG

The objective is to retrieve the address of ports and events in SystemC constructs. These are parameters given as parameters to the SystemC function. In the LLVM IR, they are computed, before that call, by a sequence of bitcodes. The key idea of this paper is to analyse these instructions, determine which ones are useful to compute these addresses, and build a new LLVM function containing *only* these instructions and returning the target value. Once built, this function can be executed.

The algorithm used to retrieve parameters' values is given by function `buildFct()`, in figure 9. The principle of this algorithm is detailed below:

- basic blocks are cloned and the association (original block → cloned block) is kept in `ValueMap`. The new blocks are added to the new function when they are created.
- all basic blocks and instructions necessary to compute the target parameter are marked by function `markUsefulInstructions()`. This is done recursively this way:
  - initially, the target instruction is pushed onto the `stack`, and while the stack is not empty, the first instruction is popped.
  - the uses of this value are pushed onto the stack and added to the `usedInstructions` as well as the associated basic block to `usedBlocks`. This is done by function `mark()` and **only** if the instruction is used *before* the call using the target value (because instructions after this one have no impact on the computation of the target value.

    - the same is done for the arguments of the current instruction
- at last, useful instructions are cloned and added to cloned basic blocks. The link between instructions in the cloned blocks is done by function `RemapInstructions` which use for this task the association between original and cloned values stored in the `ValueMap`.

The uses of instructions are directly available in the LLVM representation as it contains use-def chains between them.

The result of this algorithm is therefore a function which takes as parameter a SystemC module and return the value wanted. This function can be natively compiled by the Just-In-Time compiler (JIT) provided by the LLVM, then executed. Or it can be directly interpreted by LLVM. The result of this execution gives the value of the parameter we want (typically the port involved in a `port.write(...)`) or the event in a `wait` or `notify` statement).

As we said earlier, this approach only works for static data. In the case where the result of the function depends on dynamic information, it is not possible to execute the function built. However, this is a more general problem and not a limitation of our approach.

Our implementation in PinaVM suffers from a limitation: our analysis is limited to function bounds. We do not handle the case where data on which depends the computation of the address of a port are given as parameters. In the case, where these data are decidable, an inter-procedural analysis could be used to improve things. Or we could inline the called function. However, this case is rare and existing works have no solution either for such cases.

Although the functioning described above is well-suited to get basic types such as integers involved in a call `wait(t:int)` for instance, this is not enough considering other constructs. Indeed, in a call to `event.notify()`, applying the algorithm will get us the pointer to the event. We need, from this address, to get the module notified through this call. In this goal, we need to access the information about the architecture, computed during the elaboration phase.

6.4 User-defined communication channels

When channels connecting two ports are defined by the program itself and not provided by the SystemC library, things are more complicated. In this case, the function called cannot be detected statically because it is not known in advance. A solution would be to let the user define the name of the function, but it is not working: a user-defined channel inherits from the `sc_interface` class and the defined virtual methods appear as pointer to functions in the LLVM representation. Our approach allows to solve this problem. The idea is as follows:

1. When a call to a function pointer is encountered, we get the type of the first argument, which is the type of the object called.
2. If this is not a subtype of `sc_interface`, do nothing.
3. Else, we build and execute a function giving the address of the function called. This is done exactly the same way as described in previous section.
4. From this address, we get the corresponding compiled code, thanks to the just-in-time compiler which has a map associating LLVM representation of functions (`Function*`) and compiled code.
5. From the representation obtained, we see if it is a read/write communication.

6.5 Intermediate representation

Once SystemC constructs have been identified and linked to the architecture, we build an intermediate representation that is largely based on the LLVM representation. It is composed, as in the LLVM representation, of the CFG, with basic-blocks comprising SSA instructions, but also comprises SystemC constructs. Figure 10 illustrates the representation given by PinaVM on a 1-bit adder. One can see the normal basic blocks in white squares. The circles represent SystemC constructs and contains all information that have been retrieved from the architecture, notably the source ports, channels and target ports for communication.

As the same function can be used by several different processes, a construct contains in fact a set of tuples $(process \rightarrow communication)$.

7 Experimental results

In order to evaluate our approach, we experimented PinaVM on real SystemC examples. We give three types of experimental results. First, and most important point, we expose the ability of PinaVM to handle real SystemC code (and not a subset) compared to existing works. Then, we briefly give experimental results related to resource consumption. At last, we give preliminary results on verification back-ends.

7.1 Capabilities of PinaVM

One of our main goals with PinaVM is to address the limitations of existing solutions concerning the subset of SystemC programs handled. In [15], we described a set of SystemC examples illustrating these limitations and typical cases front-ends, available at [13] should ideally be able to take in charge. We experimented on these examples and were able to handle almost all cases, showing that our approach is more powerful than existing solutions.

Table 2 shows the ability of PinaVM to extract an intermediate formal representation from the given examples, compared to Pinapa. We only compare to Pinapa in this table because, amongst the tools we experimented, it presented the best results, thanks to the execution of the elaboration. For each tool and each example, this table indicates ✓ if the example could be analyzed, ✍ if the example could be analyzed but with (small) adaptation of the test-case, ≈ if it works partially. The concerned case is detailed below, `Easily` if the example could not be analyzed, but could be managed with a small implementation work, `Doable` if the example could not be analyzed, if this is not a theoretical limitation of the approach, but requires a huge implementation to work. And ✘ if the example could not be analyzed and if it is a fundamental limitation of the approach.

	Pinapa	PinaVM
elab-only	✓	✓
elab-easy	✓	✓
elab-easy-int	✓	✓
elab-easy-uint	✓	✓
elab-easy-array	✓	✓
elab-easy-sc_stop	✓	✓
elab-port-bool	✓	✓
elab-pointer	≈	≈
elab-instances	✍	✓
elab-clock	`Easily`	✓
signal	✍	✓
event	✓	✓
fifo	✘	`Easily`
RAM	`Doable`	✓

Table 2: Capabilities of PinaVM compared

In this table, we can see that PinaVM is clearly able to handle a larger subset of SystemC programs than others. The bigger difference is illustrated by the ”fifo” example. In this example, `sc_interfaces` used by modules to communicate are defined by the user and can not be detected statically because functions called are pointers, as detailed in section 6.4.

In the “elab_pointer”, `write()` statements are performed in a loop, and the written module depends on an index. This kind of communications depending on dynamic data are not determinable in the general case. In this case, Pinapa and PinaVM back-ends generate the code computing the recipient of the communication, introducing a potential loss of precision. In PinaVM, we use LLVM features to unroll loops. This improves a bit the solution, but this problem is a more general one, since arbitrary loops cannot be unrolled statically. A lot of work is done around this in the field of real-time systems, especially concerning computation of worst case execution time [19].

In the ”RAM” example, Pinapa experience experimental problems that would require a huge work to solve, although theoretically feasible.

7.2 Existing back-ends

A SystemC front-end should provide a usable intermediate format; In our case, as we mainly target verification tools, we were able to write back-ends to different verification back-ends. A first one to Promela, the input language for the SPIN model-checker; there was no problems implementing this and the translation was shown to be particularly efficient [14] . A second one is being implementing to the abstract interpreter of B. Jeannet *et al.* [10] and we are automating the work described in [6].

7.3 Resources consumption

At last, we evaluate PinaVM experimentally in terms of resources' consumption. We measured the time and memory necessary to analyze examples presented in section 2. Table 3 gives the time needed to compile the example "RAM" with LLVM compared to the time needed by GCC, showing that compiling with LLVM take about the same time as compiling with GCC. The time needed by PinaVM is also given and corrrespond to the time needed to compile. We do not report the values for other examples as they are very similar.

GCC	LLVM	PinaVM
1.9s	1.8s	+1.7s

Table 3: Time needed by our approach

Although those examples are not significant, in terms of complexity, our results shows that the resources needed are very low. In addition, it is to be considered that the complexity in time is linear with the size of the code. In addition, in the benchmark used in our translation to Promela, the time needed to translate to Promela is negligible compared to the time needed by Spin to verify the program.

8 Conclusion

We presented a new approach to the design of SystemC front-ends, allowing to address most of the limitations of existing works, therefore facilitating the construction of formal validation tools of Systems on a Chip. This approach is mainly based on the use of a SSA-based and executable representation.

The proposal has been implemented in an available, open-source tool called PinaVM, based on the compilation framework LLVM. Our experimental results show that PinaVM has a more powerful approach than existing works, allowing a large subset to be processed. They also show that PinaVM'S resources consumption is low and that PinaVM is usable for verification purpose.

We still do not manage all SystemC constructs, and in particular, we did not implement recognition of SystemC/TLM (*Transaction Level Modeling* constructs as of now. However, their syntax is similar to the constructs we already manage, therefore, supporting them is only a matter of implementation now.

Direct perspectives to this work include the design of validation back-ends benefiting from our intermediate representation. Actual verification back-ends of our tool show that our representation is well-suited for verification but does not benefit from its SSA nature. Perspectives include the connection of our intermediate representation to verification tools based on synchronous language (like SMV [16]), as it has already been showed that they are naturally and efficiently translated from SSA code. A second perspective we consider is to connect to simulators, pushing the idea of "execution" one step even further. The idea is to optimize the program thanks to information retrieved by PinaVM, notably concerning inter-processes communications. This is possible, since our intermediate representation is executable.

References

[1] IEEE std 1666 - 2005 IEEE standard SystemC language reference manual. *IEEE Std 1666-2005*. 1, 3.2

[2] KaSCPar - Karlsruhe SystemC parser suite. `http://www.fzi.de/index.php/de/component/content/article/238-ispe-sim/4350-sim-tools-kascpar-examples`. 3.3

[3] DragonEgg - using LLVM as a GCC backend, 2010. `http://dragonegg.llvm.org/`. 6.1

[4] L. Besnard, T. Gautier, M. Moy, J.-P. Talpin, K. Johnson, and F. Maraninchi. Automatic translation of C/C++ parallel code into synchronous formalism using an SSA intermediate form. In *Ninth International Workshop on Automated Verification of Critical Systems (AVOCS'09)*. Electronic Communications of the EASST, September 2009. 1, 3.1, 4.2

[5] N. Blanc, D. Kroening, and N. Sharygina. Scoot: A tool for the analysis of SystemC models. In *TACAS*, pages 467–470, 2008. 3.1

[6] T. Bouhadiba, F. Maraninchi, and G. Funchal. Formal and executable contracts for transaction-level modeling in systemc. In *ACM International Conference on Embedded Sofware (EMSOFT'09)*, Grenoble, France, Oct. 2009. 7.2

[7] R. Cytron, J. Ferrante, B. Rosen, M. Wegman, and F. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(4):451–490, 1991. 1

[8] C. Genz and R. Drechsler. Overcoming limitations of the SystemC data introspection. In *DATE*, 2009. 3.1, 4.3

[9] C. Helmstetter, F. Maraninchi, and L. Maillet Contoz. Full simulation coverage for SystemC transaction-level models of systems-on-a-chip. *Formal Methods in System Design*, 35(Number 2):pages 152–189, 06 2009. 3.1

[10] B. Jeannet. Relational interprocedural verification of concurrent programs. In *Software Engineering and Formal Methods, SEFM'09*. IEEE, Nov. 2009. 7.2

[11] H. Kalla, J.-P. Talpin, D. Berner, and L. Besnard. Automated translation of c/c++ models into a synchronous formalism. In *ECBS '06: Proceedings of the 13th Annual IEEE International Symposium and Workshop on Engineering of Computer Based Systems*, pages 426–436, Washington, DC, USA, 2006. IEEE Computer Society. 1, 3.1, 4.2

[12] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *CGO '04: Proceedings of the international symposium on Code generation and optimization*, page 75, Washington, DC, USA, 2004. IEEE Computer Society. 4.1, 5

[13] K. Marquet and M. Moy. http://greensocs.sourceforge.net/pinapa/download /files/frontends-testcases.tar.gz. 7.1

[14] K. Marquet, M. Moy, and B. Jeannet. An asynchronous semantics of systemc in promela. Technical Report TR-2010-7, Verimag Research Report, 2010. 7.2

[15] K. Marquet, M. Moy, and B. Karkare. A theoretical and experimental review of SystemC front-ends. Technical Report TR-2010-4, Verimag Research Report, 2010. 3.1, 3.2, 3.3, 4, 7.1

[16] K. L. McMillan. The SMV system, Nov. 06 1992. 8

[17] M. Moy, F. Maraninchi, and L. Maillet-Contoz. Lussy: A toolbox for the analysis of systems-on-a-chip at the transactional level. In *ACSD '05: Proceedings of the Fifth International Conference on Application of Concurrency to System Design*, pages 26–35, Washington, DC, USA, 2005. IEEE Computer Society. 3.3

[18] M. Moy, F. Maraninchi, and L. Maillet-Contoz. Pinapa: an extraction tool for SystemC descriptions of systems-on-a-chip. In *EMSOFT '05: Proceedings of the 5th ACM international conference on Embedded software*, pages 317–324. ACM, 2005. 3.3

[19] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström. The worst-case execution-time problem—overview of methods and survey of tools. *ACM Trans. Embed. Comput. Syst.*, 7(3):1–53, 2008. 7.1

```
Stack stack;
Set usedInstructions, usedBlocks;

bool mark(Value* v) {
   if (! usedInstructions.contains(v)) {
      usedInstructions.push(argAsInst);
      stack.push(argAsInst);
   }
   block = v->getparent();
   if (usedBlocks.contains(block))
      used_bb.push(block);
}

void markUsefulInstructions() {
   while (! stack.empty()) {
      Value* value = stack.pop();
      foreach Use use in value->getUses() {
         if (isDefinedBeforeTargetInst(use))
            mark(use);
      }
      foreach Arg arg in value->getUses() {
         if (isDefinedBeforeTargetInst(use))
            mark(use);
      }
   }
}

void cloneBlocks() {
   foreach BasicBlock bb in origFct.getBlocks() {
      if (usedBlocks.contains(bb)) {
         BasicBlock *NewBB = createBasicBlock();
         valueMap.insert(bb, NewBB);
      }
   }
}

void buildFct(Value* targetValue) {
   clones = cloneBlocks();
   Stack.push(targetValue);
   markUsefulInstructions();

   foreach BasicBlock bb in origFct.getBlocks() {
      if (usedBlocks.contains(bb)) {
         foreach Instruction inst in bb {
            if (used_instructions.contains(inst)) {
               clonedInst = inst.clone();
               RemapInstruction(clonedInst, valueMap);
               valueMap.insert(inst, clonedInst);
            }
         }
      }
      createRet();
   }
}
```

Figure 9: Algorithm to retrieve parameters’ values

```
sc_event end;
int x = 0, y = 0;
sc_in<bool> xPort, yPort;
sc_out<bool> carry;

bool carry;

do {
   x = xPort.read();
   wait(42);
} while (x == 0) {

do {
   y = yPort.read();
   wait(42);
} while (y == 0);
carry = x | y;
if (carry == 0)
   carry.write(false);
else
   carry.write(true);
end.notify();
```

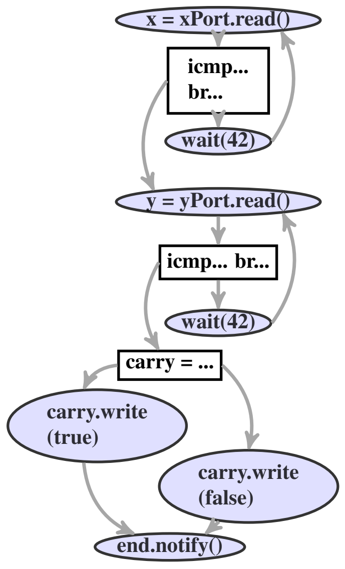

Figure 10: PinaVM intermediate representation