Comparative study of approaches to semantics extraction and virtual prototyping of system-level models

—

Preliminary Version

Loïc Besnard Thierry Gautier Florence Maraninchi Matthieu Moy
Jean-Pierre Talpin

June 25, 2009

1 Introduction

Designing chips is becoming increasingly difficult as the modeling tools and methodologies cannot keep up with the rise in system complexity. In the recent past, formal methods have proved their capability of error detection and prevention in the different phases of system modelling. Formal methods have the ability to verify the corect behavior of a specific functionality for all possible states of the system. In this report, we compare two such methods: one defined at VERIMAG-Grenoble, the other one at IRISA-Rennes. They start from SystemC specifications and extract the SystemC behavior into specific formalisms to study properties.

1.1 The SystemC library

SystemC [11] is a library for C++ that allows the modeling and simulation of complex Systems-on-a-Chip, at a high level of abstraction. It has emerged recently but has already become the de-facto standard for system-level modeling in silicon industry.

One particularity of SystemC is that the same language, C++, is used both to describe the architecture of a system and its behavior. A C++ program can therefore be compiled with any C++ compiler. Figure 1 show the definitions of two modules, `module1` and `module2` with their respective behaviors (`code1` and `code2`, first defined as C++ methods and then registered as SystemC processes). Figure 2 show the elaboration of the platform.

Being able to formally verify a SystemC programs requires several steps:

1. the ability to read SystemC programs, and to produce data-structures, or on-disk format, containing all the information about the source program (typically, under the form of an abstract syntax tree).

2. it requires one to model the semantics of SystemC, including function calls like `wait`, `read`, `write`, ... and the semantics of the SystemC scheduler.

3. But since SystemC includes all the C++ language, it also requires the ability to deal with any arbitrary C++ code.

```
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();
35      }
36    }
37    SC_HAS_PROCESS(module2);
38    module2(sc_module_name name)
39      : sc_module(name) {
40      // register "void code2()"
41      // as an SC_METHOD
42      SC_METHOD(code2);
43      dont_initialize();
44      // static sensitivity list for code2
45      sensitive << ports[0];
46      sensitive << ports[1];
47    }
48  };
```

Figure 1: Example of SystemC Program: Definition of Modules

```
49  int sc_main(int argc, char ** argv) {
50    bool init1 = true;
51    bool init2 = true;
52    if (argc > 2) {
53      init1 = !strcmp(argv[1], "true");
54      init2 = !strcmp(argv[2], "true");
55    }
56    sc_signal<bool> signal1, signal2;
57    // instantiate modules
58    module1 * instance1_1 =
59      new module1("instance1_1", init1);
60    module1 * instance1_2 =
61      new module1("instance1_2", init2);
62    module2 * instance2 =
63      new module2("instance2");
64    // connect the modules
65    instance1_1->port.bind(signal1);
66    instance1_2->port.bind(signal2);
67    instance2->ports[0].bind(signal1);
68    instance2->ports[1].bind(signal2);
69    sc_start(-1);
70  }
```

Figure 2: Example of SystemC Program: Main Function

1.2 Overview of the report

This report compares two approaches based on the translation of SystemC code to a formal language (Lustre or SMV in the case of Verimag's tool, Signal in the case of IRISA).

The approach followed by Verimag focused on points 1 and 2 above, while IRISA has interesting contributions on point 3.

The goal of this joint work is to combine the advantages of both approaches. The problem is actually not as simple as it seems to be. The way to parse SystemC proposed by Verimag, through the tool Pinapa, uses the fact that the intermediate representation of C++ code is an abstract syntax tree, and as opposed to this, the approach followed by IRISA heavily uses the fact that this intermediate representation is "static single assignment" (SSA). The approaches and their limitations are presented in sections 2 and 3.

The first step in this direction is to have a clear idea, with quantitative measurements, of the benefits of SSA in the verification tool-chain. This point is fully addressed by this report (see in particular section 4.1).

Finally, we present possible solutions to take the best of both worlds in the end of section 4. This report only gives a feasibility study, none of them are actually implemented as of now, but they are part of our plans for 2009.

2 State of the Art

2.1 IRISA

A methodology for translating SystemC models into SIGNAL programs has been described in [1]. It is depicted on the figure 3. We abstract here this methodology.

The structural information of the original SystemC code such as the module organization and their connections is directly extracted from the SystemC code. It produces a SIGNAL architecture skeleton with empty boxes. This tool is called SystemCXML [2]. Then the modules are translated one by one with the help of the translation based on a SSA to SIGNAL translation scheme defined in [15, 16]. SIGNAL models are automatically generated from C/C++ component descriptions with the help of the GNU Compiler Collection (GCC) [5] and its static single assignment (SSA) intermediate representation [6, 12]. It is integrated in the GCC compiler since the 4.0 version.

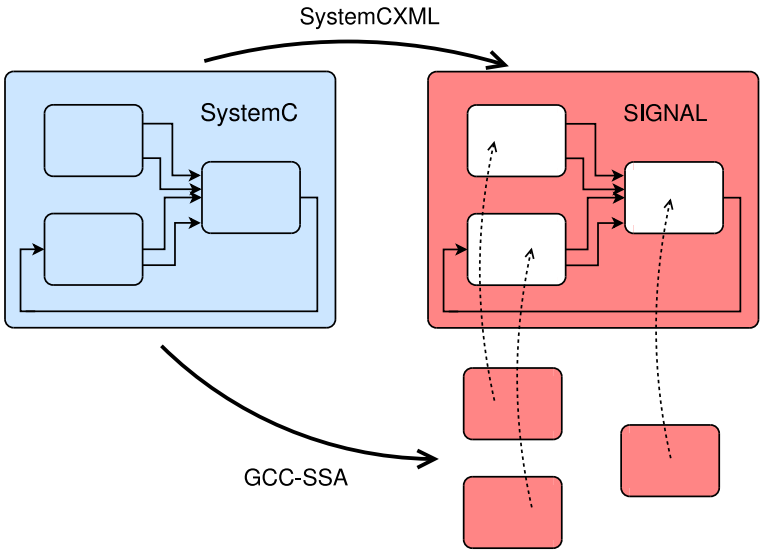

Figure 3: Methodology for translating SystemC models into SIGNAL

The approach to extracting structural design information (SystemCXML tool, see Figure 4) uses a suite of open-source technologies consisting of Doxygen [17], Apache's Xerces-C++ XML [4], in combination with a C++ library to enable validation tasks exploiting this information.

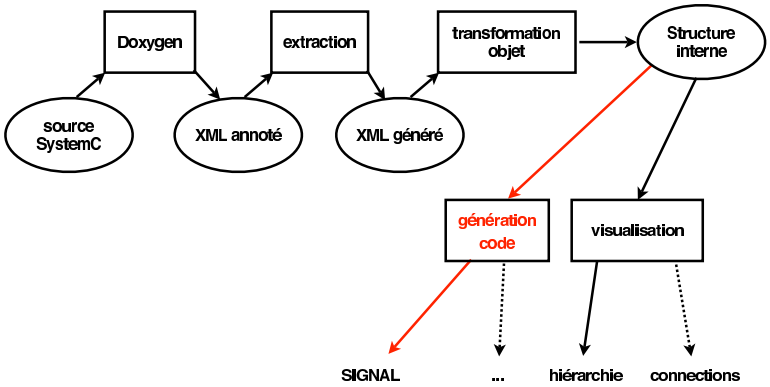

Figure 4: Design Flow for the extraction

The converting of the C/C++ code of the computing part of the SystemC modules to SIGNAL code is composed of three steps (Figure 5).

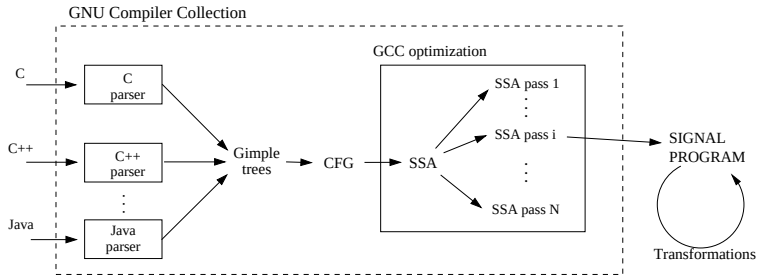

Figure 5: GCC-SSA to SIGNAL

We explain this translation using the following boolean-counter example. This program implements a 2-bits counter using 2 variables b_1, b_2. We start with $(b_1, b_2) = (0, 0)$ and increment the bit vector by one in a loop, until the value $(1, 1)$ is reached. The C code is given in Figure 6.

```
/*
  This program implements an 2-bits counter using 2 variables b1, b2.
  We start with (b1, b2) = (0, 0) and increment the bit vector by one in a loop,
  until the value (1,1) is reached.
  Then, the output out is defined (=1) after the end of the loop.
*/
void compute2( bool *out)
  {
    bool  c0 = true;
    bool  b1, c1;
    bool  b2, c2;

    b1 = false;
    b2 = false;

    while(!(b1 && b2))
      {
        c1 = b1 && c0;
        b1 = (b1 || c0) && ! (b1 && c0);
        c2 = b2 && c1;
        b2 = (b2 || c1) && ! (b2 && c1);
      }
    *out = b1 && b2;
  }
```

Figure 6: C code for the boolean-counter example

1. **Converting C/C++ into SSA:** The first step of the translation scheme consists in converting C/C++ models into SSA form. This step is performed by GCC, where it first translates C/C++ programs into Gimple Trees [8]; these are a high-level intermediate representation derived from GCC parse trees. These trees contain complete control, data, and type information of the original program. Next, GCC translates these trees into a control-flow graph (CFG) which is then transformed into SSA form. This representation is used by GCC for optimizations. The SSA code of the boolean-counter example is given in Figure 7.

```
void compute2 (bool*)(out) {
    bool c2; bool b2; bool c1;...
bb0:
    c0_8 = 1;
    b1_9 = 0;
    b2_10 = 0;
    goto <bb 22> (<L21>);
L0: ;
    D.1743_18 = !b1_1;
    if (D.1743_18) goto <L3>; else goto <L1>;
L1: ;
    D.1748_36 = !c0_8;
    if (D.1748_36) goto <L3>; else goto <L2>;
L2: ;
    iftmp.0_37 = 1;
    goto <bb 5> (<L4>);
L3: ;
    iftmp.0_35 = 0;
    # iftmp.0_3 = PHI <iftmp.0_37(3), iftmp.0_35(4)>;
L4: ;
    c1_19 = iftmp.0_3;
    if (b1_1) goto <L6>; else goto <L5>;
L5: ;
    if (c0_8) goto <L6>; else goto <L9>;
L6: ;
    D.1743_32 = !b1_1;
    if (D.1743_32) goto <L8>; else goto <L7>;
L7: ;
    D.1748_34 = !c0_8;
    if (D.1748_34) goto <L8>; else goto <L9>;
L8: ;
    iftmp.1_33 = 1;
    goto <bb 11> (<L10>);
L9: ;
    iftmp.1_31 = 0;
    # iftmp.1_4 = PHI <iftmp.1_33(9), iftmp.1_31(10)>;
L10:;
    b1_20 = iftmp.1_4;
    D.1744_21 = !b2_2;
    if (D.1744_21) goto <L13>; else goto <L11>;
L11:;
    D.1757_29 = !c1_19;
    if (D.1757_29) goto <L13>; else goto <L12>;
L12:;
    iftmp.2_30 = 1;
    goto <bb 15> (<L14>);
L13:;
    iftmp.2_28 = 0;
    # iftmp.2_5 = PHI <iftmp.2_30(13), iftmp.2_28(14)>;
L14:;
    c2_22 = iftmp.2_5;
    if (b2_2) goto <L16>; else goto <L15>;
L15:;
    if (c1_19) goto <L16>; else goto <L19>;
L16:;
    D.1744_25 = !b2_2;
    if (D.1744_25) goto <L18>; else goto <L17>;
L17:;
    D.1757_27 = !c1_19;
    if (D.1757_27) goto <L18>; else goto <L19>;
L18:;
    iftmp.3_26 = 1;
    goto <bb 21> (<L20>);
L19:;
    iftmp.3_24 = 0;
    # iftmp.3_6 = PHI <iftmp.3_26(19), iftmp.3_24(20)>;
L20:;
    b2_23 = iftmp.3_6;
    # b2_2 = PHI <b2_10(0), b2_23(21)>;
    # b1_1 = PHI <b1_9(0), b1_20(21)>;
L21:;
    D.1743_11 = !b1_1;
    if (D.1743_11) goto <L0>; else goto <L22>;
L22:;
    D.1744_12 = !b2_2;
    if (D.1744_12) goto <L0>; else goto <L23>;
L23:;
    D.1743_13 = !b1_1;
    if (D.1743_13) goto <L26>; else goto <L24>;
L24:;
    D.1744_16 = !b2_2;
    if (D.1744_16) goto <L26>; else goto <L25>;
L25:;
    iftmp.4_17 = 1;
    goto <bb 28> (<L27>);
L26:;
    iftmp.4_15 = 0;
    # iftmp.4_7 = PHI <iftmp.4_17(26), iftmp.4_15(27)>;
L27:;
    *out_14 = iftmp.4_7;
    return;
}
```

Figure 7: GCC-SSA code for the boolean-counter example

2. **Converting SSA into SIGNAL:** The next step of the translation scheme consists in converting SSA into SIGNAL processes. The translation scheme is implemented in the GCC front end. The output of this step is a SIGNAL program, **syntactically correct but not semantically**. The implementation of the SIGNAL generation is inserted in the GCC source tree as an *additional Front end optimization pass*. GCC currently features over 50 optimization passes. We can choose to use all of these by inserting our pass at the very end, but it may also make sense to exclude some of the optimizations since depending on the code, they may result in less performing code. The result of this step for the boolean-counter example is given in Figure 8. This SIGNAL program is an other view of the SSA code without any

transformation. So, the connexion to another C compiler with an SSA internal representation would be easily possible. This code is composed of a set of labeled blocks. A block is composed of a set of PHI definitions, a set of computations, and a branching.

```
 % ------------------------------------------------
    This file was automatically generated by c2signal
    translator:
    It is the FIRST STEP of the translation C to Signal:
    it is not a well formed Signal program.
    Manual editing not recommended.
    ------------------------------------------------%
process compute2 = (  ?
                      ! boolean out; )
  (| bb_0:: (|(| |)
             | (| c0_8 := 1
                | b1_9 := 0
                | b2_10 := 0
                |)
             | (| GOTO(L21)|)
             |)
   | L0:: (|(| |)
          | (| D_1743_18 := not b1_1 |)
          | (| IF (D_1743_18, GOTO(L3), GOTO(L1) ) |)
          |)
   | L1:: (|(| |)
          | (| D_1748_36 := not c0_8 |)
          | (| IF (D_1748_36, GOTO(L3), GOTO(L2) ) |)
          |)
   | L2:: (| (| |)
           | (| iftmp_0_37 := 1 |)
           | (| GOTO(L4)|)
           |)
   | L3:: (|(| |)
          | (| iftmp_0_35 := 0 |)
          | (| |)
          |)
   | L4:: (| (| iftmp_0_3 := PHI(iftmp_0_37, iftmp_0_35) |)
           | (| c1_19 := iftmp_0_3 |)
           | (| IF (b1_1, GOTO(L6), GOTO(L5) ) |)
           |)
   | L5:: (| (| |)
           | (| |)
           | (| IF (c0_8, GOTO(L6), GOTO(L9) ) |)
           |)
   | L6:: (| (| |)
           | (| D_1743_32 := not b1_1 |)
           | (| IF (D_1743_32, GOTO(L8), GOTO(L7) ) |)
           |)
   | L7:: (| (| |)
           | (| D_1748_34 := not c0_8 |)
           | (| IF (D_1748_34, GOTO(L8), GOTO(L9) ) |)
           |)
   | L8:: (|(| |)
          | (| iftmp_1_33 := 1 |)
          | (| GOTO(L10)|)
          |)
   | L9:: (| (| |)
           | (| iftmp_1_31 := 0 |)
           | (| |)
          |)
   | L10:: (| (| iftmp_1_4 := PHI(iftmp_1_33, iftmp_1_31) |)
            | (| b1_20 := iftmp_1_4
               | D_1744_21 := not b2_2 |)
            | (| IF (D_1744_21, GOTO(L13), GOTO(L11) ) |)
            |)
   | L11:: (| (| |)
            | (| D_1757_29 := not c1_19 |)
            | (| IF (D_1757_29, GOTO(L13), GOTO(L12) ) |)
            |)
   | L12:: (| (| |)
            | (| iftmp_2_30 := 1 |)
            | (| GOTO(L14)|)
            |)
   | L13:: (| (| |)
            | (| iftmp_2_28 := 0 |)
            | (| |)
            |)
   | L14::(| (| iftmp_2_5 := PHI(iftmp_2_30, iftmp_2_28) |)
          | (| c2_22 := iftmp_2_5 |)
          | (| IF (b2_2, GOTO(L16), GOTO(L15) ) |)
          |)
   | L15::(| (| |)
          | (| |)
          | (| IF (c1_19, GOTO(L16), GOTO(L19) ) |)
          |)
   | L16::(| (| |)
          | (| D_1744_25 := not b2_2 |)
          | (| IF (D_1744_25, GOTO(L18), GOTO(L17) ) |)
          |)
   | L17:: (| (| |)
            | (| D_1757_27 := not c1_19 |)
            | (| IF (D_1757_27, GOTO(L18), GOTO(L19) ) |)
            |)
   | L18:: (| (| |)
            | (| iftmp_3_26 := 1 |)
            | (| GOTO(L20) |)
            |)
   | L19:: (| (| |)
            | (| iftmp_3_24 := 0 |)
            | (| |)
           |)
   | L20:: (| (| iftmp_3_6 := PHI(iftmp_3_26, iftmp_3_24) |)
            | (| b2_23 := iftmp_3_6 |)
            | (| |)
            |)
   | L21:: (| (| b2_2 := PHI(b2_10, b2_23)
               | b1_1 := PHI(b1_9, b1_20)
              |)
            | (| D_1743_11 := not b1_1 |)
            | (| IF (D_1743_11, GOTO(L0), GOTO(L22) ) |)
            |)
   | L22:: (| (| |)
            | (| D_1744_12 := not b2_2 |)
            | (| IF (D_1744_12, GOTO(L0), GOTO(L23) )
              |)
            |)
   | L23:: (| (| |)
            | (| D_1743_13 := not b1_1 |)
            | (| IF (D_1743_13, GOTO(L26), GOTO(L24) ) |)
            |)
   | L24:: (| (| |)
            | (| D_1744_16 := not b2_2 |)
            | (| IF (D_1744_16, GOTO(L26), GOTO(L25) ) |)
            |)
   | L25:: (| (| |)
            | (| iftmp_4_17 := 1 |)
            | (| GOTO(L27)|)
            |)
   | L26:: (| (| |)
            | (| iftmp_4_15 := 0 |)
            | (| |)
            |)
   | L27:: (| (| iftmp_4_7 := PHI(iftmp_4_17, iftmp_4_15) |)
            | (| POINTER(out) := iftmp_4_7 |)
            | (| |)
            |)
  |)
 where
   %  Local declarations%
 end;
```


Figure 8: SIGNAL code **before transformations** for the boolean-counter example

3. **Transforming the SIGNAL program:** The next step consists in the definition of (i) the control induced by the references to the labels in the branching statements; (ii) the memories induced by the loops and the control. The control is given first to the first block (through the signal `bb_0` in the example). The result of this step for the boolean-counter example is given in Figure 9.

```
process compute2 =
   ( ! boolean out;
   )
 (|
% Memories induced by the while loop%
    (| Z_b1_1 := b1_1$1
     | Z_c0_8 := c0_8$1
     | Z_b2_2 := b2_2$1
     |)
% Synchronisation of the memories %
  | b1_1 ^= c0_8 ^= b2_2 ^= L0 ^= bb_0
  | % Code of the first block (labeled bb_0)%
    (| c0_8 := (1 when bb_0) default Z_c0_8
     | b1_9 := 0 when bb_0
     | b2_10 := 0 when bb_0
     |)
    % Code of the second block (labeled L0)%
  | D_1743_18 := (not Z_b1_1) when L0

  | D_1748_36 := (not Z_c0_8) when L1
  | iftmp_0_37 := 1 when L2
  | iftmp_0_35 := 0 when L3
    % Translation of (from labeled block L4)
      iftmp_0_3 := PHI(iftmp_0_37, iftmp_0_35) %
  | (| (| iftmp_0_3 := iftmp_0_35 default iftmp_0_37 |)

     | (| c1_19 := iftmp_0_3 when L4 |)
     |)
  | D_1743_32 := (not Z_b1_1) when L6
  | D_1748_34 := (not Z_c0_8) when L7
  | iftmp_1_33 := 1 when L8
  | iftmp_1_31 := 0 when L9
  | (| (| iftmp_1_4 := iftmp_1_31 default iftmp_1_33 |)
     | (| b1_20 := iftmp_1_4 when L10
        | D_1744_21 := (not Z_b2_2) when L10
        |)
     |)
  | D_1757_29 := (not c1_19) when L11
  | iftmp_2_30 := 1 when L12
  | iftmp_2_28 := 0 when L13
  | (| (| iftmp_2_5 := iftmp_2_28 default iftmp_2_30 |)
     | (| c2_22 := iftmp_2_5 when L14 |)
     |)
  | D_1744_25 := (not Z_b2_2) when L16
  | D_1757_27 := (not c1_19) when L17
  | iftmp_3_26 := 1 when L18
  | iftmp_3_24 := 0 when L19
  | (| (| iftmp_3_6 := iftmp_3_24 default iftmp_3_26 |)
     | (| b2_23 := iftmp_3_6 when L20 |)
     |)
  | (| (| b2_2 := b2_23 default (b2_10 default Z_b2_2)
        | b1_1 := b1_20 default (b1_9 default Z_b1_1)
        |)
     | (| D_1743_11 := (not b1_1) when L21 |)
     |)
  | D_1744_12 := (not b2_2) when L22
  | D_1743_13 := (not b1_1) when L23
  | D_1744_16 := (not b2_2) when L24
  | iftmp_4_17 := 1 when L25
  | iftmp_4_15 := 0 when L26
  | (| (| iftmp_4_7 := iftmp_4_15 default iftmp_4_17 |)
     | (| out_14 := iftmp_4_7 when L27 |)
     |)
  | (| out := (out_14 cell L27) when L27 |)
```

```
% Labels defining the control of the application
  One by SSA block. %

(| bb_0 := (not (^bb_0))$1 init true
 | next_L0 := (D_1744_12 when L22) default (D_1743_11 when L21)
             default false
 | L0 := next_L0 $ 1 init false
 | L1 := (not D_1743_18) when L0
 | L2 := (not D_1748_36) when L1
 | L3 := (D_1748_36 when L1) default (D_1743_18 when L0)
 | L4 := (true when L3) default (true when L2)
 | L5 := (not Z_b1_1) when L4
 | L6 := (Z_c0_8 when L5) default (Z_b1_1 when L4)
 | L7 := (not D_1743_32) when L6
 | L8 := (D_1748_34 when L7) default (D_1743_32 when L6)
 | L9 := ((not D_1748_34) when L7) default ((not Z_c0_8) when L5)
 | L10 := (true when L9) default (true when L8)
 | L11 := (not D_1744_21) when L10
 | L12 := (not D_1757_29) when L11
 | L13 := (D_1757_29 when L11) default (D_1744_21 when L10)
 | L14 := (true when L13) default (true when L12)
 | L15 := (not Z_b2_2) when L14
 | L16 := (c1_19 when L15) default (Z_b2_2 when L14)
 | L17 := (not D_1744_25) when L16
 | L18 := (D_1757_27 when L17) default (D_1744_25 when L16)
 | L19 := ((not D_1757_27) when L17) default ((not c1_19) when L1
 | L20 := (true when L19) default (true when L18)
 | L21 := (true when L20) default (true when bb_0)
 | L22 := (not D_1743_11) when L21
 | L23 := (not D_1744_12) when L22
 | L24 := (not D_1743_13) when L23
 | L25 := (not D_1744_16) when L24
 | L26 := (D_1744_16 when L24) default (D_1743_13 when L23)
 | L27 := (true when L26) default (true when L25)
 |)
 |)
 where %Local Declarations %
    Z_b1_1,Z_c0_8,Z_b2_2;
    ...
 end;
```

Figure 9: SIGNAL code for the boolean-counter example

Then one can use the Polychrony toolset [3], based on SIGNAL, that provides a formal framework:

- to validate a design at different levels,
- to refine descriptions in a top-down approach,
- to abstract properties needed for black-box composition,
- to assemble predefined components (bottom-up with COTS).

It constitutes a development environment for critical systems, from abstract specification until deployment on distributed systems. It relies on the application of formal methods, allowed by the representation of a system, at the different steps of its development, in the SIGNAL polychronous semantic model.

Polychrony is a set of tools composed of:

- A SIGNAL batch compiler providing a set of functionalities viewed as a set of services for, e.g., program transformations, optimizations, formal verification, abstraction, separate compilation, mapping, code generation, simulation, temporal profiling...
- A Graphical User Interface (editor + interactive access to compiling functionalities)
- The SIGALI tool [18, 7], an associated formal system for formal verification and controller synthesis.

The SIGNAL compiler checks for **static** problems such as contradictory clock constraints, cycles, null clocks, exclusive clocks. In order to check **dynamic** properties of the system, the SIGNAL companion model checker SIGALI can be used. It is an interactive tool specialized on algebraic reasoning in $\mathbb{Z}/3\mathbb{Z}$ logic. SIGALI transforms SIGNAL programs into sets of dynamic polynomial equations that basically describe an automaton. It can analyze this automaton and prove properties such as liveness, reachability, and deadlock. The fact that it is solely reasoning on a $\mathbb{Z}/3\mathbb{Z}$ logic constrains the conditions to the Boolean data type (true, false, absent). This is practical in the sense that true numerical verification very soon would result in state spaces that are no longer manageable, however it requires, depending on the nature of the underlying model, major or minor modifications prior to formal verification.

For many properties numerical values are not needed at all and can be abstracted away thus speeding up verification. When verification of numerical manipulations is sought, an abstraction to boolean values can be performed, that suffices in most cases to satisfy the needs.

Example: To prove the reachability of the last line of code of the boolean-counter, one can add an observer in the generated SIGNAL program (See Figure 10).

```
process compute2 =
     ( ! boolean out, OBS;
     )
  (| % Defining an observer of the signal out synchronized with the master clock.%
  (| OBS := ^out default (OBS$1 init false) | OBS ^= bb_0 |)
     % Memories induced by the while loop%
    (| Z_b1_1 := b1_1$1
     | Z_c0_8 := c0_8$1
     | Z_b2_2 := b2_2$1
     |)
  ...
```

Figure 10: The SIGNAL program with an observer

To prove the property, the SIGNAL compiler is then called, with the option that generates the control part of the program for the SIGALI tool.

The SIGALI session that proves the property is given in Figure 11.

```
> sigali
 *--------------------------------------*
 * Sigali - version 2.3 (Dec 2005)      *
 *--------------------------------------*

Sigali : read("Scompute2.z3z");  Loading the file generated by the Signal compiler.
                               ;  it expresses the synchronizations of the program.
------------------------------------------------------------------------------
Sigali : read("Creat_SDP.lib"); Polynomial Dynamical System Building
------------------------------------------------------------------------------
Sigali : read("Bibli.lib"); Loading Sigali library
------------------------------------------------------------------------------
Sigali :  subset(initial(S), AF(B_True(S, OBS))); Proving the property
------------------------------------------------------------------------------
True     ; result of the proof.
------------------------------------------------------------------------------
Sigali :
```

Figure 11: A SIGALI session proving that OBS always becomes true.

2.2 The Verimag tool-chain: LusSy

Verimag started working on the analysis and automatic proof of SystemC code in 2002, in cooperation with STMicroelectronics. The first Ph.D thesis [9] dealt with the connection of SystemC to synchronous languages (Lustre and SMV). The tool-chain developed is called LusSy, and implements the complete transformation of SystemC code into synchronous languages, without human intervention.

LusSy is able to deal with a large subset of the SystemC language specification and with some dedicated TLM channels like TAC [13] and BASIC (never released outside STMicroelectronics).

The architecture of LusSy is similar to the one of a compiler: First, the code is transformed into a data-structure with Pinapa, then, the interpretation of the semantics of SystemC takes place in the component BISE, which generates a set of communicating automata called HPIOM, before the code generation which is mostly a pretty-printer.

2.2.1 Pinapa

The first step performed by the tool-chain is to read the SystemC code. This is done by the tool Pinapa [10], which is the equivalent of a compiler front-end, applied to SystemC. But the information needed for further analysis are not limited to the usual static information (lexicography, syntax, typing, ...), but have to include the architecture, which is built at run-time. Pinapa reuses the front-end of GCC to get the Abstract Syntax Tree (AST), actually runs the elaboration of the platform, and its main job it then to make the link between the AST and the architecture.

2.2.2 BISE

The next step, performed by the component BISE is to generate the intermediate representation HPIOM. HPIOM is a simple formalism of synchronous, communicating automata. The translation for normal C++ code uses one transition for each individual action in the code. In other words, the generated HPIOM automata correspond directly to the control-flow graph. The generated HPIOM code for the two-bits counter example (Figure 6 above) is given in Figure 13.

2.2.3 Code generators

From the HPIOM code, it is relatively easy to generate code for a model-checker for synchronous languages. We can generate Lustre or SMV code. The translation of the two bit counter in Lustre is given in figure 14.

3 Limitations of the Approaches

3.1 Pros and Cons of the SSA Form

The SSA (Static Single Assignment) form is widely used in modern compilers. As the name suggests, it is a form in which variables are assigned only once (multiple assignments in the source code are usually transformed into assignments to temporary variables in SSA). One other particularity is that SSA code is a sequence of elementary instructions (usually, three-addresses code like `x=operator(y,z)`).

The main benefits are:

- It makes manipulation of SSA much simpler, one doesn't have to deal with all the constructs of the source language.
- Most interesting transformations have already been done by the compiler.

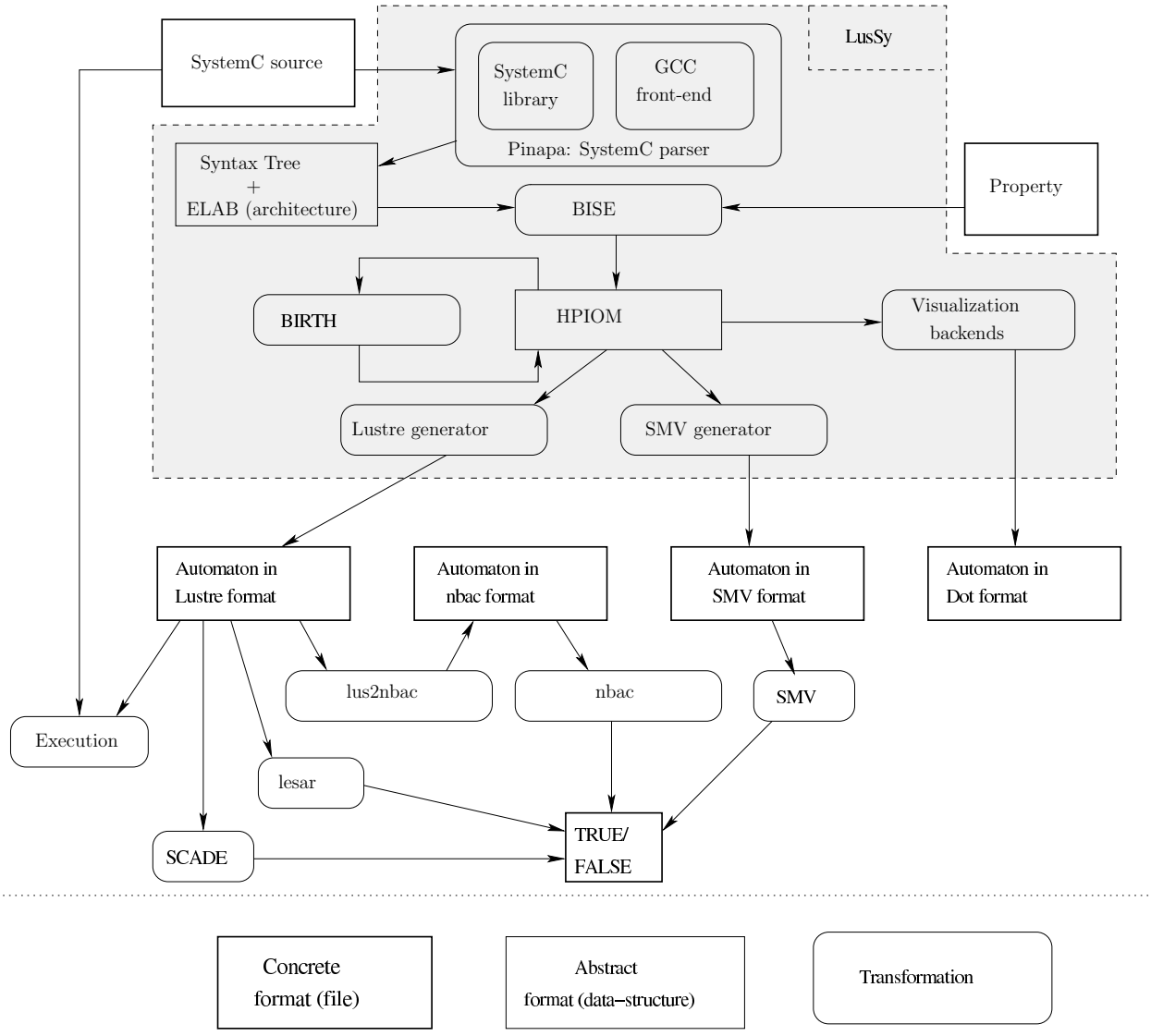

Figure 12: The LusSy tool chain

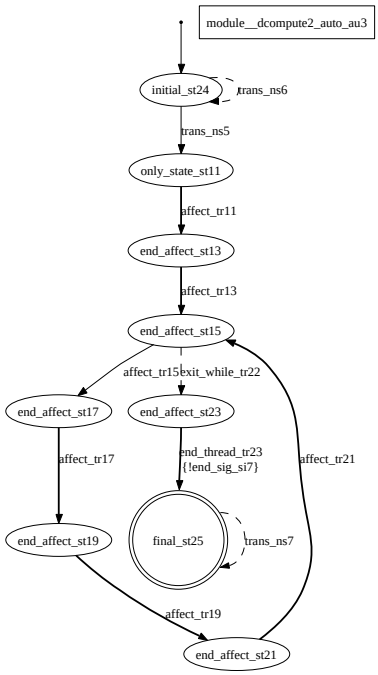

Figure 13: Generated HPIOM automaton for the two-bits counter example

- Since variables are assigned once only in a block, it makes it easy to generate code for a synchronous language, where one clock tick corresponds to at least one basic block in the control-flow graph of the program.

On the other hand

- A sequence of elementary constructs like SSA makes it hard to identify high-level constructs which would map directly to the target language (e.g. arrays are already expanded into addresses manipulation)

- Makes it hard to recognize constructs for which one would need direct semantics. In particular, SystemC constructs are expanded into a sequence of instructions which would require advanced pattern-matching to be extracted, so implementing Pinapa on top of SSA would be very difficult. This point is detailed further in section 4.4.2.

- The transformation from abstract syntax tree to SSA usually creates a lot of temporary variables. Without further optimization, this explosion of the number of variables can be an important penalty for the proof engine.

3.2 Pros and Cons of the HPIOM Form

HPIOM is the intermediate format of the LusSy toolchain. An HPIOM system is a set of automata, each automaton being a set of control points, linked by transitions. Transitions can be guarded by conditions, and can trigger assignments to variables (Boolean or integer). HPIOM was designed to be expressive enough while remaining as simple as possible.

```
node module__dcompute2_auto_au3 (elect_si2: bool)
  returns (only_state_st11, end_affect_st13, end_affect_st15, end_affect_st17,
           end_affect_st19, end_affect_st21, end_affect_st23, initial_st24, final_st25: bool;
           end_sig_si7: bool;
           );
var
  c0_va4_va6 : bool; c1_va3_va9 : bool;
  b1_va1_va7 : bool; b2_va2_va8 : bool;
  affect_tr11: bool; affect_tr13: bool;
  affect_tr15: bool; affect_tr17: bool;
  affect_tr19: bool; affect_tr21: bool;
  exit_while_tr22: bool;
  trans_ns5: bool; trans_ns6: bool;
  end_thread_tr23: bool;
  trans_ns7: bool;
let
  -- Transitions:
  affect_tr11 = false -> (pre(only_state_st11) and true);  -- > end_affect_st13
  affect_tr13 = false -> (pre(end_affect_st13) and true);  -- > end_affect_st15
  affect_tr15 = false -> (pre(end_affect_st15) and ((not (pre b1_va1_va7))
                          or (not (pre b2_va2_va8))));  -- > end_affect_st17
  affect_tr17 = false -> (pre(end_affect_st17) and true);  -- > end_affect_st19
  affect_tr19 = false -> (pre(end_affect_st19) and true);  -- > end_affect_st21
  affect_tr21 = false -> (pre(end_affect_st21) and true);  -- > end_affect_st15
  exit_while_tr22 = false -> (pre(end_affect_st15) and (not ((not (pre b1_va1_va7))
                          or (not (pre b2_va2_va8)))));  -- > end_affect_st23
  trans_ns5 = false -> (pre(initial_st24) and elect_si2);  -- > only_state_st11
  trans_ns6 = false -> (pre(initial_st24) and (not elect_si2));  -- > initial_st24
  end_thread_tr23 = false -> (pre(end_affect_st23) and true);  -- > final_st25
  trans_ns7 = false -> (pre(final_st25) and true);  -- > final_st25

  -- States:
  only_state_st11 = false -> (trans_ns5);
  end_affect_st13 = false -> (affect_tr11);
  end_affect_st15 = false -> (affect_tr13 or affect_tr21);
  end_affect_st17 = false -> (affect_tr15);
  end_affect_st19 = false -> (affect_tr17);
  end_affect_st21 = false -> (affect_tr19);
  end_affect_st23 = false -> (exit_while_tr22);
  initial_st24 = true -> (trans_ns6);
  -- final
  final_st25 = not(only_state_st11 or end_affect_st13 or end_affect_st15 or
                   end_affect_st17 or end_affect_st19 or end_affect_st21 or
                   end_affect_st23 or initial_st24);
  -- Pure signals
  end_sig_si7 = false -> end_thread_tr23;

  -- Variables
  c0_va4_va6 = true -> (pre (c0_va4_va6));
  c1_va3_va9 = true -> (if affect_tr15
                        then (((pre b1_va1_va7) and (pre c0_va4_va6)))
                        else pre (c1_va3_va9));
  b1_va1_va7 = true -> (if affect_tr11 then (false) else
    if affect_tr17 then ((((pre b1_va1_va7) or (pre c0_va4_va6)) and
                          ((not (pre b1_va1_va7)) or (not (pre c0_va4_va6)))))
    else pre (b1_va1_va7));
  b2_va2_va8 = true -> (if affect_tr13 then (false) else
    if affect_tr21 then ((((pre b2_va2_va8) or (pre c1_va3_va9)) and
                          ((not (pre b2_va2_va8)) or (not (pre c1_va3_va9)))))
    else pre (b2_va2_va8));
tel.
```


Figure 14: Generated Lustre code for the two-bits counter example

- It is synchronous, and uses a single clock,
- Nothing can be created dynamically,
- All the possible communications have to be defined statically.

The main benefits of using HPIOM as intermediate format are:

- It is easy to add new back-ends to the tool, as soon as the back-end can encode synchrony.
- The transformation from the abstract syntax tree is reasonably simple.

But the main identified drawbacks are:

- High-level constructs also have been broken down into simpler ones in HPIOM. Therefore, the transformations performed on HPIOM can not benefit from SystemC particularities (like non-preemptiveness).
- The naive encoding generates far too many transitions (one C instruction per transition as a maximum).

4 Possible cross-experiments

4.1 Benchmark

Before trying anything to combine the advantages of both approaches, we want to get a quantitative estimation of the gain of using SSA on plain imperative code. This benchmark is not meant to be comprehensive, but should give an idea of the gain to be expected.

4.1.1 Benchmark programs

To be able to perform a comparison between the two programs, we have to remain in the intersection of what the tools are able to do. That means the code should be plain C++ code, without SystemC constructs, and therefore without parallelism. Integer variables would be problematic too, since their management is too different depending on the proof engine (lesar abstracts them, SMV breaks them down into boolean, SIGALI solely works on boolean data type). To make the proof non-trivial, we use loops and/or non-determinism (a non-deterministic condition is simply an uninitialized Boolean variable used once in the program).

The properties checked on the program are always program termination (i.e. reachability of the last line of code), which is a typical safety property (any other safety could reduce to it modulo an observer). We are not interested in safety properties here.

We use two benchmark programs, each of them having two variants:

boolean-counter This program implements an 8-bits counter using 8 variables $b_1, b_2, \ldots, b_8$. We start with $(b_1, b_2, \ldots, b_8) = (0, 0, \ldots, 0)$ and increment the bit vector by one in a loop, until the value $(1, 1, \ldots, 1)$ is reached.

The non-deterministic variant called `boolean-counter-nd` does the same, but instead of incrementing by 1 at each step, it increments with a non-deterministic value in $\{0, 1\}$.

long-code This program is an arbitrary piece of code using Boolean variables. As above, we manipulate a bit-vector using 8 explicit Boolean variables. We define macros to perform bit shift and bit rotations on them, and use them to produce a long piece of code, with non-deterministic branching, but no loop. The program is designed so that the property is violated iff all non-deterministic conditions are true (which the counter-example has to state).

The variant forces on of the non-deterministic condition to be true, and the prover can then check that the property is true.

4.1.2 Results

The HPIOM part was benchmarked on a dual-core machine with processor Intel(R) Pentium(R) D CPU 3.40GHz. The IRISA part was benchmarked on an Intel Core 2 CPU T7600 2,33Ghz. We measure user time, in seconds. The columns titles are to be interpreted as follows

Lesar The HPIOM system is dumped into the Lustre language, and the lesar model-checker is used to prove the property.

Lesar (opt) Same as above, but turning the HPIOM optimizer on (live variable analysis, minimization of the number of variables for non-determinism, and trivial Boolean optimizations like `x&&true==x`.

SMV This also uses HPIOM, but dumps the system into the SMV language and uses the SMV model-checker to perform the proof.

SMV (opt) Same, with the HPIOM optimizer turned on.

SIGALI the technique has been explained in section 2.1. The tool chain is the following: C → SSA → SIGNAL → SIGALI. For the translation SIGNAL → SIGALI, the SIGNAL compiler is used: the clock synthesis is applied to the SIGNAL generated program. It reduces the number of constraints.

	HPIOM				SIGNAL
Program	Lesar	Lesar (opt)	SMV	SMV (opt)	SIGALI
boolean-counter	6.5	2.5	2.7	2.18	0.090
boolean-counter-nd	22	0.13	2.5	2.0	0.015
long-code (true)	4.8	1.6	0.34	0.18	0.014
long-code (false)	5.3	2.2	0.7	0.15	0.011

The next table gives numbers about the HPIOM automaton generated, regardless of the performance of a back-end proof engine. “States” and “Trans.” means the number of states (control-points) and transitions of the automaton. “CE Length” means the length of the shortest Counter-Example for the system. We do not give the numbers for the variants as they would be identical when applicable.

As presented before, SIGALI is a model-checking tool which manipulates Polynomial Dynamical Systems (that can be seen as an implicit representation of an automaton) as intermediate models. The techniques used consist in manipulating the system of equations instead of the sets of solution,

which avoids the enumeration of the state space. Each set of states is uniquely characterized by a polynomial and the operations on sets can be equivalently performed on the associated polynomials. We give

1. the number of theorical states (Th. States): for N delay signals in the SIGNAL/SIGALI program, it is equal to 2^N.
2. the number of reachable states (R. States),
3. the number of transitions.

SIGALI does not provide counter-example for the system when the proof fails.

	HPIOM			SIGNAL			
Program	States	Trans.	CE length	State variables	Th. States	R. States	Trans.
boolean-counter	26	29	4096	12	4096	257	257
long-code	193	208	176	2	4	3	1536

The results of this benchmark are to be taken with care, since the number of test-case is relatively low, but this clearly validates the fact that using SSA within the tool-chain is a valid approach. The number of state variables remain very low in the Signal code generated from SSA, which means the explosion of the number of variables in SSA can be solved efficiently.

4.2 SIGNAL Backend for HPIOM?

A naive interpretation of the above benchmark would be that SIGALI is a better proof engine than Lesar and SMV. Since the LusSy tool-chain was built to be open to other proofs engine, one could think of writing a SIGNAL back-end for LusSy (i.e. a code generator from HPIOM to SIGNAL).

But we believe that the strong point of the SIGNAL transformation is not the proof engine itself, but the transformation from SSA. Indeed, writing a signal back-end for LusSy would probably mean combining the *weaknesses* of both tools: the unoptimized transformation from C++ to HPIOM, without using SMV which is one of the most efficient proof engine for synchronous programs.

We have therefore decided not to go further in this direction.

4.3 Embed SSA into PINAPA's AST manipulation

At this point, it becomes clear that a tool-chain combining the advantages of Verimag's and IRISA's approaches would have to write an equivalent of Pinapa, but producing an SSA form instead of an abstract syntax tree (AST).

Since recognizing SystemC primitives inside an SSA form is hard, one solution could be to use the AST $\rightarrow$ SSA transformation of a compiler on well-identified pieces of code. That would allow keeping the high-level constructs unmodified, and graft pieces of SSA obtained for portions of code independent from SystemC. Unfortunately, this approach doesn't solve the problem of arrays/iterators, which have already been broken down into pointer arithmetic in GCC-SSA.

4.4 LLVM's SSA form: an improvement over GCC ?

4.4.1 LLVM: Low Level Virtual Machine

LLVM is a compiler infrastructure based on an intermediate code (LLVM code). It provides essentially

- Front ends for various languages, generating LLVM code (for C++, the choices are clang, which supports C correctly, but with only very partial support for C++ as of writting, and llvm-g++, which is a patch for GCC to generate LLVM code),
- A static optimizer (called "opt"),
- A virtual machine with jit compilation and optimizations,
- A native code generator.

We are interested only in the front-end, the intermediate code, and to some extend the optimizer.

The intermediate code is mostly a typed assembly-like language, with the particularity of being SSA-based. For example, the following code is valid LLVM code:

```
define i32 @main() {
       %some-variable = add i32 1, 1   ; integer with the value 1+1
       %some-other-variable = add i32 %some-variable, 1
       ret i32 %some-other-variable
}
```

while the next one is not (the assembler complains with "Redefinition of value 'some-variable' of type 'i32' "):

```
define i32 @main() {
       %some-variable = add i32 1, 1   ; integer with the value 1+1
       %some-variable = add i32 %some-variable, 1
       ret i32 %some-variable
}
```

Note however that the LLVM code can use pointer types, and the the pointed-to values are not necessarily SSA. For example, this is a valid LLVM program:

```
define i32 @main() {
       %some-pointer = alloca i32 ; declares an int in memory (stack)
       store i32 42, i32* %some-pointer
       store i32 54, i32* %some-pointer
       %retval = load i32* %some-pointer
       ret i32 %retval
}
```

Fortunately, the command `opt -mem2reg` converts memory uses to "register" (i.e. non-pointer values), which allows working on a fully-SSA representation.

4.4.2 A comparison of LLVM and GCC's SSA form, and their behavior with SystemC

Practical Issues with GCC The use of GCC as a component in Pinapa has shown several issues. They are not deep theoretical problems, but prevent the tool from being as simple as it should, both for the user and the developer:

- GCC is licensed under the GNU General Public License (GPL), which is incompatible with the license of SystemC. This makes it illegal to distribute the whole Pinapa tool as a binary, and forces separate distribution, and therefore compilation by the user (which takes around 1 hour and several hundreds of megabyte of disk space).
- GCC is not made to be modular, by design and for non-technical reasons [14]. Using GCC in Pinapa means embedding the whole code generator. Even the front-end is highly code-generation oriented, does some on-the-fly optimizations. This has lead to some de-optimization surgery in the implementation of Pinapa, which we would like to get rid of.

LLVM solves both issues. It is licensed under the University of Illinois/NCSA Open Source License, a BSD-like, very permissive license. It is designed in a very modular manner.

Description of the experiment To understand the way LLVM code and GCC-SSA code can express various constructs, we used the following setup:

- LLVM code is generated with `llvm-gcc`, without optimization, and then passed to `opt -mem2reg` to use registers instead of memory, to take full advantage of the SSA form.
- GCC-SSA code is generated with `gcc -O -fdump-tree-ssa` (the -O is necessary to force GCC to use the SSA form). We give here the default dump format, which is a pretty-print in a C-like syntax.

For the details, the Makefile for generating these files is given in appendix A.

Simple examples on plain C++ code To get an idea of what the code looks like, here is a minimalistic C program and its translation to LLVM and GCC-SSA code:

C code for the simple example:

```
extern int f(int x);

int main (void) {
        int x;
        x = f(x);
        x = f(x);
        return x;
}
```

GCC-SSA code:

```
;; Function main (main)

main ()
{
  int x;
```

```
  int D.1531;
  int D.1530;
  int D.1529;

<bb 2>:
  D.1529_2 = f (x_1);
  x_3 = D.1529_2;
  D.1530_4 = f (x_3);
  x_5 = D.1530_4;
  D.1531_6 = x_5;
  return D.1531_6;

}
```

LLVM code:

```
; ModuleID = 'very-simple.opt.bc'
target datalayout = "e-p:32:32:32-i1:8:8-i8:8:8-i16:16:16-i32:32:32-i64:32:64-f32:32:32-f64
target triple = "i386-pc-linux-gnu"

define i32 @main() nounwind  {
entry:
        %"alloca point" = bitcast i32 0 to i32          ; <i32> [#uses=0]
        %tmp2 = call i32 @f( i32 undef ) nounwind               ; <i32> [#uses=1]
        %tmp4 = call i32 @f( i32 %tmp2 ) nounwind               ; <i32> [#uses=1]
        br label %return

return:         ; preds = %entry
        ret i32 %tmp4
}

declare i32 @f(i32)
```

The codes are very similar. Now, let's see a little bit more complex example with a non-linear control-flow.

C code:

```
extern int f(int arg);
int cond;

int main (void) {
        int x;
```

```
        x = f(42);
        if (cond) {
                x = f(x);
        } else {
                x = f(x + 1);
        }
        x = f(x + 2);
        return x;
}
```

GCC-SSA code:

```
;; Function main (main)

main ()
{
  int x;
  int D.1537;
  int D.1536;
  int D.1535;
  int D.1534;
  int D.1533;
  int D.1532;
  int cond.0;
  int D.1530;

<bb 2>:
  D.1530_2 = f (42);
  x_3 = D.1530_2;
  cond.0_5 = cond;
  if (cond.0_5 != 0) goto <L0>; else goto <L1>;

<L0>:;
  D.1532_6 = f (x_3);
  x_7 = D.1532_6;
  goto <bb 5> (<L2>);

<L1>:;
  D.1533_8 = x_3 + 1;
  D.1534_9 = f (D.1533_8);
  x_10 = D.1534_9;

  # x_1 = PHI <x_7(3), x_10(4)>;
<L2>:;
  D.1535_11 = x_1 + 2;
```

```
  D.1536_12 = f (D.1535_11);
  x_13 = D.1536_12;
  D.1537_14 = x_13;
  return D.1537_14;

}
```

LLVM code:

```
; ModuleID = 'if.opt.bc'
target datalayout = "e-p:32:32:32-i1:8:8-i8:8:8-i16:16:16-i32:32:32-i64:32:64-f32:32:32-f64
target triple = "i386-pc-linux-gnu"
@cond = weak global i32 0               ; <i32*> [#uses=1]

define i32 @main() nounwind  {
entry:
        %"alloca point" = bitcast i32 0 to i32          ; <i32> [#uses=0]
        %tmp1 = call i32 @f( i32 42 ) nounwind          ; <i32> [#uses=2]
        %tmp2 = load i32* @cond, align 4                ; <i32> [#uses=1]
        %tmp3 = icmp ne i32 %tmp2, 0            ; <i1> [#uses=1]
        %tmp34 = zext i1 %tmp3 to i8            ; <i8> [#uses=1]
        %toBool = icmp ne i8 %tmp34, 0          ; <i1> [#uses=1]
        br i1 %toBool, label %bb, label %bb7

bb:             ; preds = %entry
        %tmp6 = call i32 @f( i32 %tmp1 ) nounwind               ; <i32> [#uses=1]
        br label %bb11

bb7:            ; preds = %entry
        %tmp9 = add i32 %tmp1, 1                ; <i32> [#uses=1]
        %tmp10 = call i32 @f( i32 %tmp9 ) nounwind              ; <i32> [#uses=1]
        br label %bb11

bb11:           ; preds = %bb7, %bb
        %x.0 = phi i32 [ %tmp10, %bb7 ], [ %tmp6, %bb ]         ; <i32> [#uses=1]
        %tmp13 = add i32 %x.0, 2                ; <i32> [#uses=1]
        %tmp14 = call i32 @f( i32 %tmp13 ) nounwind             ; <i32> [#uses=1]
        br label %return

return:         ; preds = %bb11
        ret i32 %tmp14
}

declare i32 @f(i32)
```

Again, the codes are very similar. We can see the "PHI" statement in both cases right after the "if".

Simple SystemC constructs: read/write on a signal SystemC constructs are much more complex: everything is object-oriented, and method calls imply more complexity in the generated code than plain functions are.

Here is another example, using read and write on an sc_signal (only excerpts are given here, the whole code would be much longer):

C++ code:

```
SC_MODULE(some_systemc_module) {
        sc_in<bool> in_bool;
        sc_out<bool> out_bool;
        void compute(void) {
                volatile int read_value, written_value;
                read_value = in_bool.read();
                written_value = read_value;
                out_bool.write(written_value);
        }
        SC_CTOR(some_systemc_module) {
                SC_METHOD(compute);
        }
};
```

GCC-SSA code:

```
void some_systemc_module::compute() (this)
{
        // [...]
<L1>:;
  D.108566_9 = this_5->m_interface;
  D.108553_10 = D.108566_9;
  D.108554_11 = D.108553_10;
  D.108555_12 = D.108554_11->D.16938._vptr.sc_interface;
  D.108556_13 = D.108555_12 + 28B;
  D.108557_14 = *D.108556_13;
  D.108558_15 = OBJ_TYPE_REF(D.108557_14;D.108554_11->7) (D.108554_11);
  D.108559_16 = D.108558_15;
  D.93006_17 = D.108559_16;
  D.93007_18 = *D.93006_17;
  D.93008_19 = (int) D.93007_18;
  read_value = D.93008_19;
  read_value.481_22 = read_value;
  written_value = read_value.481_22;
  written_value.482_25 = written_value;
```

```
  D.93011_26 = written_value.482_25 != 0;
  D.93004 = D.93011_26;
  D.93012_29 = &this_1->out_bool.D.92966;
  this_30 = D.93012_29;
  value__31 = &D.93004;
  D.108569_32 = &this_30->D.61853.D.58996;
  this_33 = D.108569_32;
  D.108576_34 = this_33->m_interface;
  if (D.108576_34 == 0B) goto <L2>; else goto <L3>;

<L2>:;
  SC_ID_GET_IF_.531_35 = (const char *) &SC_ID_GET_IF_;
  D.108580_36 = &this_33->D.58883;
  report_error (D.108580_36, SC_ID_GET_IF_.531_35, &"port is not bound"[0]);

<L3>:;
  D.108581_37 = this_33->m_interface;
  D.108570_38 = D.108581_37;
  D.108571_39 = D.108570_38;
  D.108572_40 = D.108571_39->D.54207.D.16938._vptr.sc_interface;
  D.108573_41 = D.108572_40 + 52B;
  D.108574_42 = *D.108573_41;
  OBJ_TYPE_REF(D.108574_42;D.108571_39->13) (D.108571_39, value__31);
  return;
}
```

Unfortunately, the template code in `sc_in` and `sc_out` have been inlined (one can see the message "port is not bound") although we used the option `-fno-inline-functions`. The C++ method call has already been translated intro a virtual function table lookup, uses pointer arithmetic, ...

LLVM code:

```
define linkonce void @_ZN19some_systemc_module7computeEv(%struct.some_systemc_module* %this)
; [...]
        %tmp2 = getelementptr %struct.some_systemc_module* %this, i32 0, i32 1          ; <%
        %tmp3 = call i8* @_ZNK7sc_core5sc_inIbE4readEv( %"struct.sc_core::sc_in<bool>"* %tmp
        %tmp4 = load i8* %tmp3, align 1         ; <i8> [#uses=1]
        %tmp45 = zext i8 %tmp4 to i32           ; <i32> [#uses=1]
        volatile store i32 %tmp45, i32* %read_value, align 4
        %tmp6 = volatile load i32* %read_value, align 4         ; <i32> [#uses=1]
        volatile store i32 %tmp6, i32* %written_value, align 4
        %tmp7 = volatile load i32* %written_value, align 4              ; <i32> [#uses=1]
        %tmp8 = icmp ne i32 %tmp7, 0            ; <i1> [#uses=1]
        %tmp89 = zext i1 %tmp8 to i8            ; <i8> [#uses=1]
        store i8 %tmp89, i8* %tmp, align 1
        %tmp11 = getelementptr %struct.some_systemc_module* %this, i32 0, i32 2         ; <%
```

```
        %tmp12 = getelementptr %"struct.sc_core::sc_out<bool>"* %tmp11, i32 0, i32 0
        call void @_ZN7sc_core8sc_inoutIbE5writeERKb( %"struct.sc_core::sc_inout<bool>"* %tm
; [...]
}
```

The code is still very complex compared to the C++ source, but in this case, it might be feasible to match a SystemC construct: we still have most high-level constructs in the code, they haven't been broken down into pointer arithmetic yet.

4.5 Actually modify a compiler to prevent SSA expansion

LLVM uses SSA, perhaps modifying clang is an option ?

Use tracability already deployed for debug info to attach our own information to a piece of code.

Feasibility study at least for now.

5 Conclusion

A Makefile

This is the Makefile used for the generation of GCC-SSA and LLVM code.

```
ifndef SUF
SUF=cpp
endif

ifndef COMP
COMP=g++
endif

ifndef SRC
SRC=${wildcard *.$(SUF)}
endif

ifndef LL
LL=${patsubst %.$(SUF),%.ll,$(SRC)}
endif

ifndef EXE
EXE=${patsubst %.$(SUF),%.exe,$(SRC)}
endif

ifndef LLVMGCCFLAGS
LLVMGCCFLAGS=-fno-inline-functions
endif

ifndef CPPFLAGS
CPPFLAGS=-fno-inline-functions
endif
```

```
ifndef OPTFLAGS
OPTFLAGS=-mem2reg -disable-opt
endif

GCC_SSA=${patsubst %.$(SUF),%.$(SUF).ssa,$(SRC)}
LLOPT=${patsubst %.ll,%.opt.ll,$(LL)}


ll: $(LL)
llopt: $(LLOPT)
exe: $(EXE)
gcc-ssa: $(GCC_SSA)

%.ll: %.$(SUF) Makefile ../include.mk
        llvm-$(COMP) $(LLVMGCCFLAGS) -emit-llvm -S $< -o $@ $(INCLUDE)

%.opt.ll: %.ll Makefile ../include.mk
        llvm-as -f $*.ll -o $*.bc
        opt -f $(OPTFLAGS) $*.bc -o $*.opt.bc
        llvm-dis -f $*.opt.bc -o $@

# -fdump-tree-ssa works only if -O is provided.
%.$(SUF).ssa: %.$(SUF) Makefile ../include.mk
        $(COMP) -O $(CPPFLAGS) -c -fdump-tree-ssa $(INCLUDE) $< -o /dev/null
        mv $<.*.ssa $@

%.exe: %.$(SUF) Makefile ../include.mk ../minilib/minilib.o
        $(COMP) $< -o $@ ../minilib/minilib.o $(LIBS) $(INCLUDE)

realclean: clean
        $(RM) *~ ../minilib/minilib.o

clean:
        -$(RM) *.exe *.bc *.ssa *.o

clean-ll:
        -$(RM) *.ll

../minilib/minilib.o: ../minilib/minilib.c
        gcc -c $< -o $@
```

References

[1] David Berner. *Using formal methods for the coDesign of embedded systems.* PhD thesis, Université de Rennes 1, Rennes, France, March 2006.

[2] David Berner, Hiren Patel, Deepak Mathaikutty, Jean-Pierre Talpin, and Sandeep Shukla. Systemcxml: An extensible systemc front end using xml. In *Forum on specification and design languages (FDL)*, Lausanne, Switzerland, September 2005.

[3] IRISA Espresso Team. Polychrony tool. http://www.irisa.fr/espresso/Polychrony.

[4] The Apache Software Foundation. Xerces C++ validating XML Parser. Website: http://xml.apache.org/xerces-c/.

[5] Free Software Foundation. The GNU compiler collection. http://gcc.gnu.org.

[6] GCC Developers Summit. *Proceedings of the 2003 GCC Developers Summit*, Ottawa, Ontario Canada, May 2003.

[7] Hervé Marchand and Eric Rutten. Sigali user manual. http://www.irisa.fr/espresso/Polychrony.

[8] Jason Merrill. GENERIC and GIMPLE: A new tree representation for entire functions. In *GCC Developers Summit*, Ottawa, Canada, May 2003.

[9] Matthieu Moy. *Techniques and Tools for the Verification of Systems-on-a-Chip at the Transaction Level*. PhD thesis, INPG, Grenoble, France, December 2005.

[10] Matthieu Moy, Florence Maraninchi, and Laurent Maillet-Contoz. Pinapa: An extraction tool for systemc descriptions of systems-on-a-chip. In *EMSOFT*, pages 317 – 324, September 2005.

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

[12] The Tree SSA project. Tree-SSA. `http://gcc.gnu.org/projects/tree-ssa`.

[13] STMicroelectronics SPG Group. Tac: transaction accurate communication channel. `http://www.greensocs.com/projects/TACPackage`.

[14] Richard Stallman. Converting the gcc backend to a library? gcc@gcc.gnu.org mailing list for the GCC project, Jan 2000. `http://gcc.gnu.org/ml/gcc/2000-01/msg00572.html`.

[15] Jean-Pierre Talpin, Paul Le Guernic, Sandeep Kumar Shukla, Frédéric Doucet, and Rajesh Gupta. Formal refinement checking in a system-level design methodology. *Fundam. Inf.*, 62(2):243–273, 2004.

[16] Jean-Pierre Talpin, Paul Le Guernic, Sandeep Kumar Shukla, and Rajesh Gupta. A compositional behavioral modeling framework for embedded system design and conformance checking. *Int. J. Parallel Program.*, 33(6):613–643, 2005.

[17] Dimitri van Heesch and The Doxygen Team. Doxygen, an automated code documentation system. http://www.doxygen.org.

[18] IRISA Vertecs/Espresso Teams. Sigali tool. http://www.irisa.fr/vertecs/Softwares/sigali.html.