Encoding SystemC Models in Formal Synchronous Formalism

Riadh HOCINE
Department of Computer Science, University of Batna
Algeria

Hamoudi KALLA
Department of Computer Science, University of Batna
Algeria

ABSTRACT
The size and thus the complexity of many systems, that use an intellectual property component (IP), have reached a level where design validation with mere testing and simulation does not deliver the required quality any more. Obtaining a formal model from a non-formal one is a complex and error prone task. A logical step is therefore to try to generate automatically a formal description from an existing non-formal system model, thus making this step faster and more reliable.

In this paper, we describe a methodology to automatically generate formal synchronous models from existing non-formal system level design descriptions that integrates smoothly into existing co-design flows. We exemplify the approach with the popular system design language SystemC and the flexible and expressive synchronous dataflow formalism SIGNAL. SystemC is a HDL which allows for modeling systems in behavioral level, it is a set of library routines and macros implemented in C++, it is a good language for input of design flow for the systems which requires verification, but it is not a formal language.

General Terms
Embedded Systems, Intellectual Property Components, Hardware Description Language, Synchronous Formalism, Formal Methods.

Keywords
SystemC, SIGNAL, Pointers Analysis, Static Single Assignment, Functional and Compositional Correctness.

1. INTRODUCTION
The miniaturization of silicon transistors makes it possible to integrate hundreds of millions of them on a single chip - 267 millions for example for the Power5 processor of IBM in 2004. The Semiconductor Industry Association (SIA) is predicting that by 2020 the number of transistors on a single high performance chip will be exceeding 22 billion (Figure 1).
The size and therefore the complexity of systems are growing exponentially but competition and consumer demand is asking for shorter time to market. While observing these developments, one may ask how companies will be able to deal with these challenges. One possibility is the degradation of product quality. Indeed, many electronic products we can buy today, especially in the leading edge lines, show more or less annoying problems. In order to work against this trend, designer productivity has to be increased drastically. This can be done by (i) modeling at a higher level of abstraction, (ii) the reuse of components, (iii) using formal methods in the design process, and (iv) using more advanced tools that properly integrate one or more of the above in existing design flows.

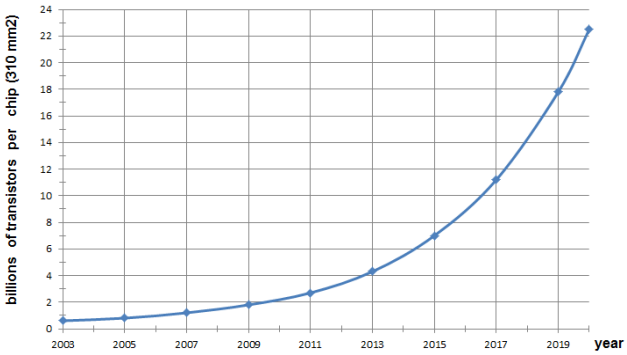


Fig 1: Number of transistors on a single chip predicted by the SIA (January 2006)

Various hardware description languages (HDLs) have been used as input to design process of digital systems in the aim to describe hardware and sometimes software functionality of systems. Designers often write system models using such languages to verify the functional correctness. In order to reduce design cost and accelerate the design process of complex systems, the designers are bound to reuse existing blocks called Intellectual Property (IPs). From this IP blocks, designers adapt quickly the system to the target application.

Functional and compositional correctness of IPs, are an important part of design process, however they are typically a weak spot of general purpose imperative programming languages. The problem is even more apparent when designers use pointers in the model description. Many automated simulator and test tools have been developed to deal with design verification problems such as [1]. However, mere simulation with non-formal development tools does by no means cover all design anomalies. This requires the use of formal methods to ensure the quality of system design. Formal methods have a big potential for detecting and preventing errors, however, their application requires a formal model. There are many reasons why most system designs languages and methodologies are non-formal, one of them is simplicity.

SystemC presents a new approach to the concepts of HDL, as it combines hardware and software descriptions at different levels of design, by extending C++ with a new library [2]. This library contains all of the necessary components required to transform

C++ into a hardware description language. Such additions include constructs for concurrently, time notion, communication, reactivity and hardware data types. However, the verification of SystemC designs is a serious bottleneck in the system design flow. Much work has been developed to solving this problem [3, 4].

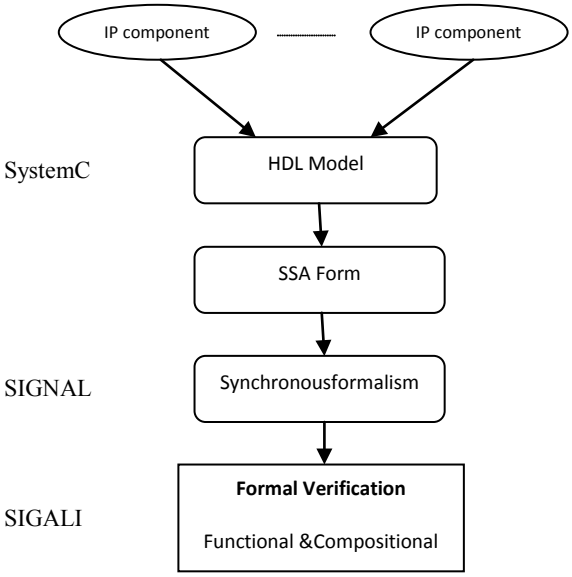


Fig 2: Our approach

As for any non-formal language, simulation and testing are its most important debugging and validation tools. However, the combination of formal verification methods and simulation may improve verification productivity and reduce the co stand time-to-market. Formal methods are mathematically based techniques for the description of system properties in the development of software and hardware systems. One major problem with formal methods however is the building of the formal system models. This is still considered too complex for a standard design engineer and an error prone and time consuming task. This paper is hence trying to leverage the situation by automatically generating formal models from SystemC programs. Fig 2 illustrates the approach of how to translate existing standard SystemC models into SIGNAL descriptions.

As the subject of the correctness of SystemC is receiving increasing attention, there are quite some attempts to link formal or semi-formal methods to a SystemC design flow. Some of them are SystemPerl [5], EDG [6], or C++ as in the BALBOA framework [7]. However, each of these approaches have their own drawbacks. SystemPerl and gSysC [8] for instance, require the user to add certain hints into the source file and although SystemPerl handles all SystemC structural information, it does not recognize all C++ constructs. EDG is a commercial front-end parser for C/C++ that parses C/C++ intoa data structure, which can then be used to interpret SystemC constructs. However, interpretation of SystemC constructs is a complex and time consuming task, plus EDG is not to be freely used in public domain. BALBOA implements its own reflection mechanism in C++ which again only handles a small subset of the SystemC language. Other approaches such as [9] require modifications of the SystemC libraries.

The paper is structured as follows. Section 2 covers preliminaries, and Section 3 presents our approach to automatically translate SystemC models into SIGNAL Models. Section 4 presents the implementation of our approach. Section 5 discusses how the resulting model can then be used to apply formal methods, and finally the article concludes in Section 6.

2. BASIC CONCEPTS

2.1 Intellectual Property (IP)

Designers try to encourage the reuse of code, moving towards the assembly of predesigned blocks and pre verification designated by the term “Intellectual Properties” (IP). This concept, born in the mid-90s, has led to several concepts: reuse components [10], virtual components or simply macros.
The Intellectual Property (IP) design reuse is one of the most promising techniques that solve the productivity gap problem. It is now accepted that *SoC* will be verified and synthesized from high-level description using HDL language [11].

2.2 Hardware/Software Co-design Using SystemC Language

SystemC is an object-oriented system level language for embedded systems design, co-design and verification. It presents a new approach to the concepts of HDLs, as combines hardware and software descriptions at different levels of design, by extending C++ with a new library. This library contains all of the necessary components required to transform C++ into a hardware description language which provide an event-driven simulation kernel [2]. Such additions include constructs for concurrency, time notion, communication, reactivity and hardware data types.

2.3 Synchronous Formalism in SIGNAL Language

A synchronous formalism rely on the synchronous hypothesis[12], which lets computations and behaviors be divided into a discrete sequence of computation steps which are equivalently called reactions or execution instants. In itself, this assumption is rather common in practical embedded system design. The synchronous hypothesis is based on the principle of determinism that in each instant of any signal’s clock, the propagation of value is well behaved so that the status of every signal is established and defined before to being tested or used.
SIGNAL is a data flow synchronous language for reactive systems that offers a framework to give executable specification of hardware/software components [13]. A SIGNAL program is a set of relationship between signals, which specified the constraints on the values of the clocks signals.
In SIGNAL, an executable specification is represented by a process ***P***, which itself consists of the simultaneous composition of elementary equations ***x:=f(y,z)***. Equations and processes are combined using synchronous composition ***P|Q*** to denote the simultaneity of ***P*** and ***Q*** with respect to the lexical scope of a process ***P*** is written ***P|x***.

2.4 Static Single Assignment (SSA)

The control structure of C++ is very complex, and because of the inherent differences compared to synchronous languages, these complex control structures would be difficult to represent in SIGNAL. The C++ compilers internally reduce the complexity of the control in order to simplify the application of optimizations. In GCC this step is called gimplification, and the result is the GIMPLE representation. GIMPLE retains much of the structure of the parse trees: lexical scopes and control constructs such as loops are represented as containers, rather than markers. However, expressions are broken down into a 3-address form, using temporary variables to hold intermediate values. Similarly, in GIMPLE no container node is ever used for its value; if a condition for example has a value, it is stored into a temporary variable within the controlled blocks, and that temporary variable is used in place of the container. For example if (x<10) will result in a GIMPLE code T1 = x<10; if (T1). All conditional jump expressions are transformed to gotos. Since GCC version 4.0 the internal data structure uses another modification called Static Single Assignment (SSA) [14]. SSA is a form of GIMPLE where each program variable is assigned one time only in the entire program. The SSA represents a procedure or a program as a directed graph G = (V,E), where the set V represents the control flow nodes or vertices and the relation E represents the jumps in the control flow.

3. ENCODING SYSTEMC STRUCTURES IN SIGNAL LANGUAGE

3.1 Encoding SSA structures

3.1.1 Encoding SSA Graph

There are three types of SSA blocks (nodes): Basic, Test and Join. Each of these blocks contains atomic statements, and every variable in Basic and Join blocks receives exactly one assignment. It is therefore possible to execute all atomic statements may be conditioned by one Boolean condition signal and the statement of that block are then scheduled for execution only when this signal is present and its value is true.

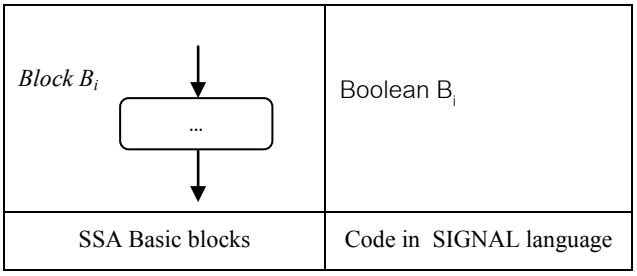


Fig 3: Encoding SSA Basic blocks into SIGNAL

3.1.2 Encoding SSA ϕFunction

The ϕ function of Join block J_k merges all the different versions of the variables coming of the predecessors of J_k. It produces as output the most recent version of variable. In SIGNAL, this function is represented by a sampling equation. For example, the statement of block J_k of Fig 4.

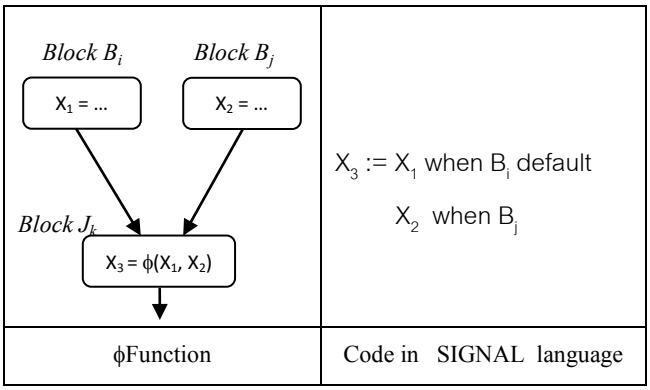


Fig 4: Example of Encoding of ϕin SIGNAL

3.2 Encoding Conditional Statements

Each SSA Test block defines a conditional branching statement. For example, the execution of two successors blocks Bj and Bk of a Test block Tm depends on the value of its conditional expression.

```
IF(Y>X)  <Statement1> ELSE <Statement2>
```

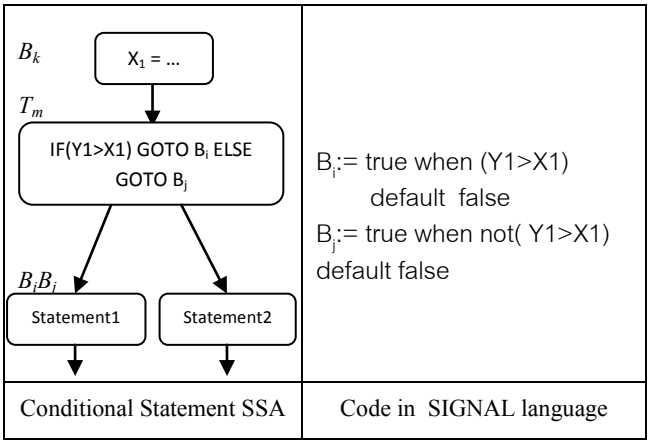
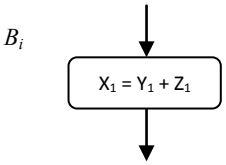


Fig 5: Example of Encoding Conditional Statement in SIGNAL

3.3 Encoding Assignment Statements

An SSA assignment statement never contains more than three operands (except function call) and has implicit side effects [14].SIGNAL assignment equations have the same form than the SSA assignment statement [15], which makes it straightforward to provide for each SSA assignment an equivalent SIGNAL equation.



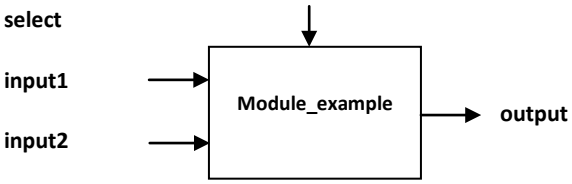
SSA form	Code in SIGNAL language
	boolean B_i $X_1 := (Y_1 + Z_1)$when B_i

Fig 6: Encoding SSA assignment statement inSIGNAL

Since statements are executed only when heir corresponding block is activated, we condition the execution of their SGNAL counterparts by the Boolean signal of the corresponding block except Join block statements.

3.4 Encoding Module Structure

A Module is the basic object in SystemC that includes ports, constructors, data members, function members and maybe internal memory storage and internal functions. A Module can be thought of as a process or a box in the hardware block diagram. The following figure(Fig 7) shown example of a simple Module and its translation in SIGNAL. The idea is to replace the module constructor process whose sensitivity list contains only (in this example) the Boolean signal "*select*" by a new SIGNAL process that is activated according to the Boolean value of the B_k block.




```
#include"systemc.h"
SC_MODULE(module_example)
{  sc_in<int>    input1;
   sc_in<int>    input2;
   sc_in<bool>  select;
   sc_out<int> output;
   void module_process()
   {  output = input1 + input2;
   }
   SC_CTOR(module_example)   //constructor
    {  SC_METHOD(module_process);
       Sensitive << select ;
    };
};
```

Fig 7: Example of a module and its SystemCdescription

- Translation into SIGNAL language

```
process module_example =
(  ? integer input1, input2 ;
   ? Boolean select ;
   ! integer output ;
)
(  | output := input1 + input2 when B1
   | B1 := true when select default false
)
where Boolean B1;
end
```

3.5 Encoding Pointers

In SystemC (C/C++), the semantics of pointers is the address of an element in memory and their uses in load (...=**P,* ...=*P*) and store *(*P=... or P=...)* instructions. It is also used into pass parameters by reference; access array elements and address dynamically allocated memory.

We propose a solution for encoding pointers in SIGNAL that allows fast alias analysis as in the case of [16]. Our solution is based on approach proposed in [17] and [18] to encoding of pointers for hardware synthesis from C language.

3.5.1 Expressions of reading pointers (Load statements)

For this, we have two types of expressions that:

- The assignment statements of the form: $A_i = f(P_k)$where P_k is a pointer to a data set.
- The conditions of: ***loop***, ***if*** and ***switch*** a statement which contains P_k. The idea is to assign to each expression $A_i = f(...,*P_k,...)$ where P_k is a pointer to a finite set of variables or array elements. The pointer P_k points to y_k where $y_k \in \{y0, y_1, \ldots, y_n\}$.
  1. The first step is to replace each pointer P_k of input program by the following new variables :
     - ***Start_P_k***: which contains the value of y_k, pointed by P_k at this time.
     - ***P_k_tag***:which contains the k where $k \in \{0,1, 2,..., n\}$a value associated with each element of y_k pointed by P_k.
     - ***P_k_index*** :the offset within the array (defined only in the case of a pointer to an array element), it contains the index of the array's cell pointed by P_k.
  2. The second step consists to apply the SSA renaming variables to $Start\_P_k$ only for different conditional statements and we may uses more than one definition P_k_tag inside a function.
     - Remove the symbols "*" and "***&***" associated with pointers in SSA form.
     - Replace each statement of the form $a_j = *P_k$ by the following conditional statements:

```
IF(P_k_tag == 0) Start_P_k = y_0;
ELSE IF(P_k_tag == 1) Start_P_k = y_1;
ELSE …
…
ELSE IF(P_k_tag ==n) Start_P_k = y_n;
```

In order to simplify and improve the automatic generation of SSA form, we replace the nesting if-else.

IF(P_k_tag == 0) Start_P_k_0 = y_0;
IF(Pk_tag == 1) Start_P_k_1 = y1;
…
IF(Pk_tag == n) Start_P_k_n = y_n;
Start_P_k_m = ϕ(Start_P_k_0, Start_P_k_1,…, Start_P_k_n)

Finaly, we most add the a_j = *Start_P_k_m* statement to completely replace a_j = ***P_k**.

3. The last step is to translate all the code in SIGNAL language as follows:

| Start_P_1 := y_0 when B_k
| Start_P_2 := y_1 when B_{k+1}
…
| Start_P_{n+1} := y_n when B_{k+n}
| Start_P_m := Start_P_1 when B_k default
Start_P_2 when B_{k+1}default
...
Start_P_n when B_{k+1}default
Start_P_{n+1} when B_{k+n}default false
// associating code with the assignment of reading the value pointed by Pk (Start_Pm)
| B_k := true when(P_tag_1 = 0)
| B_{k+1} := true when(P_tag_2 = 0)
…
| B_{k+n} := true when(P_tag_n = 0)

For reasons of simplification of automatic generation from SSA form, we propose the following SIGNALcode:

| Start_P_k := y_0 when (P_tag = 0) when B_k
default y_1 when (P_tag = 1) when B_{k+1}
…
default y_n when (P_tag = n) when B_{k+n}

3.5.2 Examples of translation reading pointers statement into synchronous formalism

The code gives a short example of our translation approach for reading pointers statements into SSA and Signal language.

First example: Translation of ***X=*P+1***

We suppose that *P*is a pointer which pointing at set of variables space *{a,b,c,tab[]}* and *P_tag ={0,1,2,3}*

- Translation into SSA form

B_k: IF($\_tag_1$ == 0) Start_P_1 = a_1;
B_{k+1}: IF($\_tag_1$ == 1) Start_P_2 = b_1;
B_{k+2}: IF($\_tag_1$ == 2) Start_P_3 = c_1;
B_{k+3}: IF($\_tag_1$ == 3) Start_P_4 = tab[P_index];
J: Start_P5 = ϕ(Start_P_1, Start_P_2, Start_P_3, Start_P_4)
B_j: X_1 = Start_P_5 + 1;

- Translation into SIGNAL language

| Start_P_1 := a when B_k
| Start_P_2 := b when B_{k+1}
| Start_P_3 := c when B_{k+2}
| Start_P_4 := tab[P_index] when B_{k+3}
| Start_P_5 := Start_P_1 when B_k default
Start_P_2 when B_{k+1} default
Start_P_3 when B_{k+2} default
Start_P_4 when B_{k+3}
| X_1 := (Start_P_5+1) when B_j
| Bj := true when $B_k$default
true when B_{k+1} default
true when B_{k+2} default
true when B_{k+3} default false
| B_k:= true when(P_tag_1=0) default false
| B_{k+1} := true when(P_tag_1=1) default false
| B_{k+2} := true when(P_tag_1=2) default false
| B_{k+3} := true when(P_tag_1=3) default false

Second example: Translation of ***if(X>*P) X=X-1 else X=0***

We suppose yet that *P*is a pointer which pointing at set of variables space *{a,b,c,tab []}* and *P_tag ={0,1,2,3}*.

- Translation into SSA form

B_k: IF($\_tag_1$ == 0) Start_P_1 = a_1;
B_{k+1}: IF($\_tag_1$ == 1) Start_P_2 = b_1;
B_{k+2}: IF($\_tag_1$ == 2) Start_P_3 = c_1;
B_{k+3}: IF($\_tag_1$ == 3) Start_P_4 = tab[P_index];
J_1: Start_P5 = ϕ(Start_P_1, Start_P_2, Start_P_3, Start_P_4)
B_j/B_i: IF(X_1> Start_P_5) X_2= X_1– 1; ELSE X_3= 0;
J_2: X_5 = ϕ(X_1, X_2, X_3, X_4)

- Translation into SIGNAL language

| Start_P_1 := a when B_k
| Start_P_2 := b when B_{k+1}
| Start_P_3 := c when B_{k+2}
| Start_P_4 := tab[P_index] when B_{k+3}
| Start_P_5 := Start_P_1 when B_k default
Start_P_2 when B_{k+1} default
Start_P_3 when B_{k+2} default
Start_P_4 when B_{k+3}
| X_2 := X1 -1 when B_i
| X_3 := 0 when B_j
| X_4 := $X_2$when B_i default X_3 when B_j
| B_i := true when (X_1> Start_P_5)when B default false
| B_j := true when not(X_1> Start_P_5) when B default false
| B :=true when B_{k+1} default
true when B_{k+2} default
true when B_{k+3} default false
| B_k:= true when(P_tag_1=0) default false
| B_{k+1} := true when(P_tag_1=1) default false
| B_{k+2} := true when(P_tag_1=2) default false
| B_{k+3} := true when(P_tag_1=3) default false

3.5.3 Expressions of writing pointers (Store statements)

There are two cases for writing pointers:

- The case of a change of address: $P_k = \&x_i$. When x_i is any variable program
- The case of a change memory location value pointed by P_k: $P_k :=$ <*statement*>

3.5.4 The case of Pk = &xiassignment

In this case, just replace this assignation in SSA form by:

B_j : $P_k\_tag_1 = i$; // when *i* is the only value associated to x_i

Start_p_k_1 = x_i

3.5.4.1 The case of P_k = <statement>assignment

In SSA form:

B_j : Start_p_k_1 = <statement>

In SIGNAL language:

| Start_p_k_1 : = <statement> when B_j

4. IMPLEMENTATION

One of the main reasons why we chose to use the SSA form in our work is that SSA has been adopted as an optimization framework by compilers, such as GCC and the Java virtual machine Jikes RVM. This allows an easy use of our approach by designers using a common software programming language to describe their systems. In this work, we are targeting SystemC models for which we have implemented our SSA to SIGNAL transformation using GCC. Designers using C/C++ can easily integrate our SIGNAL generation pass into their installed C/C++ software programming framework. We are using the GCC version 4.0, which implements a new optimization framework (Tree-SSA) based on SSA that operates on GCC's tree representation. One advantage of our transformation scheme is that systems modeled using some programming languages that are supported by GCC other than SystemC, such as Java and C/C++, can be easily transformed into Signal processes with no additional effort.

5. VERIFICATION

The design verification of the generated SIGNAL models can be accomplished at the selected level of abstraction using SIGNAL compiler and SIGALI tool. The Signal compiler allows static checking for types, dependencies, and clock constraints, while dynamic properties can be checked by SIGALI, which can verify for example the liveness of the system using the vivace command, safety properties using theinvariant command, and reachability properties using the accessible command. Other custom dynamic properties can also be constructed and verified. If an error is detected in the SIGNAL formal model, it has to be corrected directly in the SystemC model. This is because the transformation process is automated and there is no way to go back in the opposite direction. An error found in the formal model therefore still has to be located in the SystemC model; however the preservation of structure of the transformation is helping to localize it.

6. CONCLUSION

We present a methodology and tools to exemplify how a formal support platform for SystemC could work without having to deal with the complexity of formal design entry and inconsistencies of two separate models. We show how to create a formal model based on existing SystemC descriptions, which can then be used for verification and validation purposes. One characteristic of the approach is to separately obtain the structural and the behavioral information by extensively using existing tools and therefore concentrating on the actual problems. We detail the methodology, present the tools involved and show the process with the help of an example. The major difficulties in this approach were threefold: (i) choosing the proper intermediate format, that could lower the complexity of SystemC without making it unmanageable. (ii) the definition and verification of the correspondences between elementary SSA statements and SIGNAL constructs was a big part of the work. In hindsight we are surprised how smoothly each element found its counterpart.
In order to support a larger subset of SystemC an equivalent library in SIGNAL has to be established that includes all the SystemC statements.

7. ACKNOWLEDGMENTS

We wish to thank David Berner member of the ESSRESSO Team, IRISA, University of Rennes, France, for help.

8. REFERENCES

[1] Séméria L.and Ghosh A. 2000. Methodology for Hardware/Software Co-verification in C/C++.In Proceedings of the 2000 Asia and South Pacific Design Automation Conference(ASP-DAC)pp 405-408 (24-28 Jun. 2000),Yokohama, Japan.

[2] IEEE Std 1666-2005. IEEE Standard SystemCLangauge Reference Manual, 2005.

[3] Marquet K., Moy M. and JeannetB. "Efficient Encoding of SystemC/TML in Promela". DATICS-IMECS03, Hong-Kong 2011.

[4] Habibi A. and Tahar S. "On the Transformation of SystemC to AsmL Using Abstract Interpretation". Electronic Notes in Theoretical Computer Science (ENTSC) vol. 131, pp. 39-49, 2005 in press.

[5] Snyder W.2006.SystemPerl - APerl Library for SystemC. http://www.veripool.com/systemperl.html.

[6] E.D.G.C. Front-End. Edison Design Group C++ Front-End. Website:http://edg.com/cpp.html, 2006.

[7] Doucet F., Shukla S., and Gupta R.2003. Introspection in System-Level Language Frameworks: Meta-level vs.Integrated.In Proceedings of 2003 Design, Automation and Test in Europe Conference (DATE)vol. 1, pp. 382–387,(2003).

[8] Eibl C. J., Albrecht C., and R. Hagenau.2005. gSysC: A Graphical Front End for SystemC. In Proceedings of 19th

European Conference on Modelingand Simulation (ECMS), Riga, Latvia (Jun. 2005).

[9] Große D., Drechsler R, Linhard L. and Angst G.2003. Efficient Automatic Visualization of SystemC Designs.In Proceedings of the Forum on Specification and Design Languages (FDL), Frankfurt, Germany (Sep. 2003).

[10] Gajski, D.D., Wu A.C.-H., Chaiyakul V., Mori S., Nukiyama T. and Bricaud P. 2000. Essential Issues for IP Reuse. In Proceedings of the 2000 Asia and South Pacific Design Automation Conference (ASP-DAC)pp. 37-42 (25-28 Jan. 2000). Yokohama, Japan.

[11] Martin G. 1998. Design Methodologies for System Level IP. In Proceedings of the 1998 Design, Automation and Test in Europe Conference(DATE)pp. 286-189 (23-26 Feb. 1998), Paris France.

[12] Potop-Butucaru D., De Simone R. andTalpin J.-P."The Synchronous Hypothesis and Polychronous Languages". Embedded Systems Design and Verification, pp 6-16-6-27, CRC press 2009.

[13] Marchand H. and Rutten E. 2002. SIGNAL and SIGALI User's Manual. Research Report IRISA/INRIA-Rennes, France.

[14] Novello D. 2003. Tree SSA - A New High-Level Optimization Framework for the GNU Compiler Collection. In Proceeding of the Nord/USENIX Users Conference (Feb. 2003).

[15] Kalla H., TalpinJ.-P., Berner D. and BesnardL. 2006. Automated Translation of C/C++ Models into a Synchronous Formalism. In Proceedings of the 13th Annual IEEE International Symposium and Workshop on Engineering of Computer Based Systems (ECBS)vol. 9 pp. 436 (27-30 Mar. 2006). Postdam, Germany.

[16] Séméria L. 2001. Applying Pointer Analysis to the Synthesis of Hardware from C. Doctoral thesis, Department of Electrical Engineering of Stanford University, USA.

[17] SémériaL. and De Micheli G. "Encoding of Pointers for Hardware Synthesis". IEEE Transactions on Very Large Scale Integration (VLSI) Systems - System Level Design. Vol 9 Issues 6 (Jan. 2001).

[18] Séméria L. and De MicheliG.1998. SpC: Synthesis of Pointers in C Application of Pointer Analysis to the Behavioral Synthesis from C. In Proceedings of the 1998 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)pp. 340-346 (8-12 Nov. 1998).