Automated Translation of C/C++ Models into a Synchronous Formalism

Hamoudi Kalla, Jean-Pierre Talpin, David Berner, Loic Besnard
IRISA-INRIA, ESPRESSO team,
Campus Universitaire de Beaulieu
35042 Rennes, Cedex France
{kalla,talpin,dberner,lbesnard}@irisa.fr

Abstract

For complex systems that are reusing intellectual property components, functional and compositional design correctness are an important part of the design process. Common system level capture in software programming languages such as C/C++ allow for a comfortable design entry and simulation, but mere simulation is not enough to ensure proper design integration. Validating that reused components are properly connected to each other and function correctly has become a major issue for such designs and requires the use of formal methods. In this paper, we propose an approach in which we automatically translate C/C++ programs into the synchronous formalism SIGNAL*, hence enabling the application of formal methods without having to deal with the complex and error prone task to build formal models by hand. The main benefit of considering the model of* SIGNAL *for C/C++ languages lies in the formal nature of the synchronous language* SIGNAL*, which supports verification and optimization techniques. The C/C++ into* SIGNAL *transformation process is performed in two steps. We first translate C/C++ programs into an intermediate Static Single Assignment form, and next we translate this into* SIGNAL *programs. Our implementation of the* SIGNAL *generation is inserted in the GNU Compiler Collection source code as an additional Front end optimization pass. It does benefit from both GCC code optimization techniques as well as the optimizations of the SIGNAL compiler.*
Keywords: *Static Single Assignment, GNU Compiler Collection,* SIGNAL*, Synchronous Formalism, Functional and Compositional Design Correctness, Formal Methods*

1. Introduction

Number of software programming languages such as C/C++ are used to describe hardware and software functionality of embedded systems. Designers often write system models using such languages to verify the functional correctness and the performance behavior of the entire design as well as for fast functional simulation. In the aim to reduce design cost and to accelerate the design process of complex embedded systems, designers are bound to reuse existing Intellectual Property components (IPs). An IP can be any piece of design artifact that is reusable in space (by other groups or companies) or time (in subsequent projects) [23]. The reuse of customizable IPs may lead to significant improvements in design productivity and reliability.

Functional and compositional correctness of IPs, are an important part of the system 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 system models. Many automated simulator and test tools [16] have been developed to deal with design verification problems. However, mere simulation with non-formal development tools does by no means cover all design errors. What we therefore need is to use formal methods to ensure the quality of system designs. 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 standard C/C++ programs [8].

The solution we propose is based on SIGNAL [4], a multi-clocked synchronous data-flow formalism predominantly used in embedded system design. The implicit use of the synchronous language SIGNAL promises to ensure a higher-quality of the overall system design by taking advantage of the formal nature of the synchronous language to validate system designs. 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. We use GCC to transform C/C++ programs into SSA form and then apply a translation scheme to transform SSA into SIGNAL processes (Figure 1). SIGNAL programs are themselves represented by the CDFG (control-data flow graph)

structure of the Polychrony workbench [10], the tool that implements SIGNAL. In order to reduce system validation time, code optimization techniques can be performed at GNU Compiler Collection level and at SIGNAL compiler level.

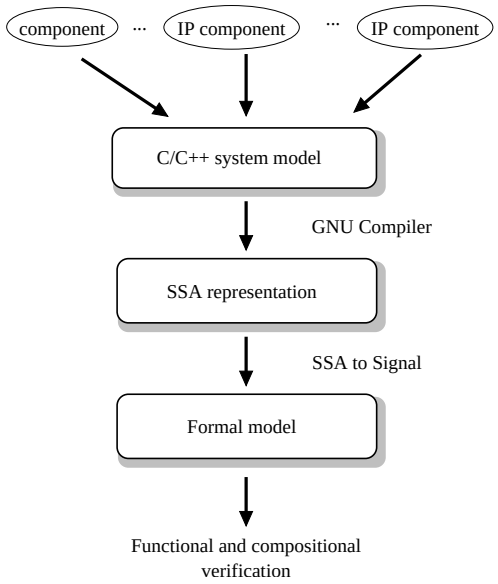


Figure 1. Our methodology

Section 2 shortly presents the intermediate representations under consideration. Section 3 gives a short overview on hardware/software design using the C/C++ programming language. Section 4 presents basic translation rules to transform SSA trees into SIGNAL's CDFGs, an example of which is detailed in Section 5. Finally, Section 6 presents the some implementation details of our technique.

2. Preliminaries

2.1. Synchronous data-flow

SIGNAL [3] offers a formal framework to give executable specifications of hardware/software components. In SIGNAL, an executable specification is represented by a process P, which itself consists of the simultaneous composition of elementary equations $x := y\ f\ z$ or $x := f(y, z)$. Equations and processes are combined using synchronous composition $P \,|\, Q$ to denote the simultaneity of P and Q with respect to a common timing model. Restricting a signal name x to the lexical scope of a process P is written P/x.

$$\text{(process)} \quad P, Q ::= x := y\ f\ z \mid (P \,|\, Q) \mid P/x$$

Informally, the structure of a process in concrete syntax is as follows:

```
process NAME =
  { [parameters] }
  ( ? [input signals];
    ! [output signals] )
  (|
     [equations]
   |)
     where
       [local declarations];
     end;
```

The inputs and outputs of an equation are signals. A signal is the sampling of a continuous stream of values at a given symbolic clock (see Figure 2). An event corresponds to the value carried by the signal at a given sample of time or instant.

A signal x is represented by an ordered sequence of events v_t composed of a sampled value v and of a time tag t. When the signal does not carry a value that is relevant to the time-sampling t, it is regarded as absent, noted $\perp$.

In SIGNAL, the presence of a value along a signal x is denoted by the proposition $\hat{}x$ that refers to the *clock* of x and can be defined by a boolean operation, e.g., $y := \hat{}x \stackrel{\text{def}}{=} y := (x = x)$. In the example of Figure 2, the signals x and y are synchronous, i.e they are always present (or absent) at the same time samples. By contrast, the signals x and z (resp. y and z) are not synchronous.

x :	1	2	$\perp$	2	1	9	$\perp$	8	$\perp$	...
y :	0	3	$\perp$	6	5	5	$\perp$	3	$\perp$	...
z :	2	$\perp$	1	$\perp$	2	$\perp$	4	$\perp$	6	...

Figure 2. Three signals x, y and z.

In SIGNAL, an equation "$x := yfz$" denotes a relation between the input signals y and z and an output signal x by an operator f. Particular some basic operators are:

- **Delay "$"**, which gives access to the value of a signal at its previous time sample. The delayed value $x\$$ of x is defined at the same clock as x.
- **Initialization "init"** defines the initial value of a delayed signal: the equation "$y := x\$\ init\ v$" defines the value of y by v and then by the previous value of the signal x.

  Example. For "$y := x\$\ init\ 1$", we have:

x :	0	5	$\perp$	0	4	$\perp$	1	3	$\perp$	2	...
y :	1	0	$\perp$	5	0	$\perp$	4	1	$\perp$	3	2

- **Sampling or conditioning, "when" operator:** the equation "$z := x\ when\ b$" defines z by x when b is present with the value true. Not that the input signals x and b may have unrelated clocks.

  Example. For $z := x\ when\ b$, we have:

$x:$	4	3	$\perp$	0	18	1
$b:$	$true$	$false$	$\perp$	$true$	$true$	$false$
$z:$	4	$\perp$	$\perp$	0	18	$\perp$

- **Merging, "default" operator:** the equation "$x := y\ default\ z$" merges the signals y and z with a priority to y when both y and z are present.

  Example. For $z := x\ default\ y$, we have:

$x:$	4	$\perp$	$\perp$	5	1	$\perp$	1	0	$\perp$
$y:$	0	1	$\perp$	$\perp$	2	$\perp$	0	2	4
$z:$	4	1	$\perp$	5	1	$\perp$	1	0	4

2.2. Control Data Flow Graph

In compilation technology, a Control Data Flow Graph (CDFG) 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. We define a control flow node as one of the following type of vertices:

- **Basic blocks:** are instructions without any jumps.
- **Test blocks:** describe conditional branching expressions.
- **Join blocks:** the end of the conditional Branch is represented by a Join node. Every Test node T_i has a corresponding Join successor node J_i.

Figure 3 gives the example of a CFDG with four Basic blocks $\{B1, B2, B3, B4\}$, one Test block $T1$ and one Join block $J1$.

2.3. Static Single Assignment

Many compiler optimization methods use information provided by CDFGs. SSA is a form of CDFG that allows optimizations to be done efficiently and easily. Since the release of version 4.0 in spring 2005, GCC uses SSA as an intermediate representation and pivotal format for all optimization passes. In SSA, every variable receives exactly one assignment during its lifetime. Two steps are required to translate the CDFG form into SSA. In the first step, each variable x is given a new name x_i for each x assignment statement (see Figure 3). In the second step, a

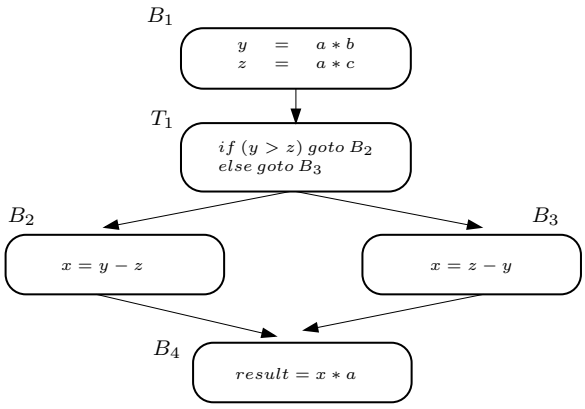


Figure 3. A Control Data Flow Graph

special assignment statement, called ϕ function, is added to Join blocks J_m. It merges all the different versions of the variables coming from the predecessors of J_m. It has the form $x_i = \phi(\ldots, x_j, \ldots, x_k, \ldots)$, where operands of ϕ are the new names of x associated with the predecessors of J_m. The Φ function returns as output the value of x_j that appears on the execution path. Figure 4 gives the SSA form of the CDFG graph of Figure 3.

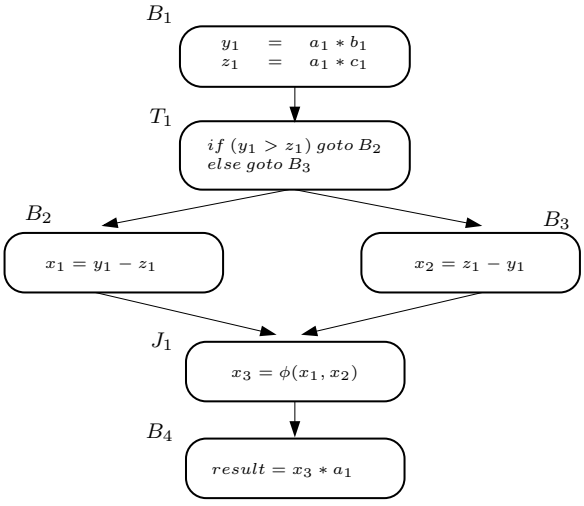


Figure 4. A Static Single Assignment Graph

3. Hardware/Software Design using C/C++

Over the past decade, important efforts have been devoted on proposing modeling and design environments for hardware/software architectures using general purpose programming languages such as C/C++. The goal pursued with SpecC, SystemC, HandelC, and other environments is to enable engineers and programmers to design, simulate and

evaluate embedded systems descriptions comprising both hardware and software components using mostly standard C/C++ development environments.

Moreover, many tools such as CONES [18] from AT&T Bell Laboratories have been developed that attempt automated behavioral synthesis of C/C++ descriptions. Behavioral synthesis consists in transforming a behavioral (algorithmic) description of a design into a Register Transfer Level (RTL) description of the design.

It cannot be denied that system design with C/C++ does not present any problems. High-level programming constructs and data-structures such as pointer and arbitrary control flow operations make the program optimization process and design validation difficult in getting behavioral information, which is untraceable to most program analysis algorithms.

Also, compositional verification of the designed system correctness is equally difficult in the presence of pre-defined libraries and IP ("intellectual property") components. Many simulators and test tools have been developed to validate system design, but simulation can only check for a limited number of input vectors whose effectiveness naturally decreases with the increased complexity of systems. When reaching certain levels of system complexity, simulation does not give sufficient confidence on the actual functional correctness of the system being modeled. Therefore, there is a need for new validation techniques to ensure a "reasonable" design quality that goes beyond what pure simulation and testing can do.

4. Translating C/C++ System Models into SIGNAL Processes

In this section, we present a framework to automatically transform C/C++ design descriptions into a formal description where we are able to use formal methods such as correct-by-construction transformation algorithms and model checking techniques to ensure design optimization and correctness.

An important point here is to hide the actual complexity of the underlying formal framework to a great extent, hence facilitating access to formal methods that otherwise would be impossible to use or only with important time and effort. The transformation scheme that we propose is a two-step process. In the first step, we use GCC to translate C/C++ programs that model the system behavior into SSA representations. And second, we automatically transform these SSA representations into SIGNAL process descriptions.

Here we are only presenting how SSA is transformed into SIGNAL. The C/C++ to SSA transformations are largely discussed in the literature [14, 15] and need no further analysis. We first present basic rules to translate SSA nodes and edges into SIGNAL equations. Then, we give the equivalent SIGNAL equations of SSA assignment and conditional branching statements. Finally, we show how C++ pointer variables can be encoded in SIGNAL.

4.1. Encoding SSA Graph

4.1.1 Encoding SSA blocks

There are three types of SSA blocks (or 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 of a block in one logical instant, meaning that these statements may be conditioned by one boolean condition or clock. Therefore, for each Basic block we create a boolean signal, and the statements of that block are then scheduled for execution only when this signal is present and its value is true.

Figure 5 shows concretely, how a Basic B_i block is represented by a boolean signal B_i in the SIGNAL code.

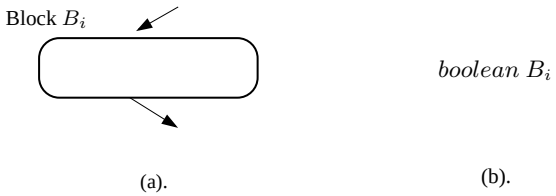


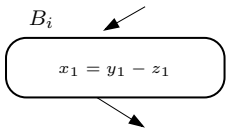
Figure 5. Encoding (a). SSA Basic blocks into (b). SIGNAL

In contrast to this, we do not need to create boolean signals for Test and Join Blocks. This is because their statements can be scheduled for execution only when the signal of its Basic predecessor block is present and its value is true.

4.1.2 Encoding Assignment Statements

An SSA assignment statement never contains more than three operands (except function calls) and has no implicit side effects. SIGNAL assignment equations have the same form than the SSA assignment statement, which makes it straightforward to provide for each SSA assignment an equivalent SIGNAL equation.

Since statements are executed only when their corresponding block is activated, we condition the execution of their SIGNAL counterparts by the boolean signal of the corresponding block as shown in Figure 6 (except for Join block statements).



(a).

$boolean\ B_i$
$x_1 := (y_1 - z_1)\ when\ B_i$

(b).

Figure 6. An SSA assignment statement (a) and its SIGNAL representation (b)

4.1.3 Encoding ϕ Functions

The ϕ function of a Join block J_k merges all the different versions of the variables coming from the predecessors of J_k. It produces as output the most recent version of a variable. In SIGNAL, we represent this function by a sampling equation. For example, the statement of the block J_k of Figure 7:

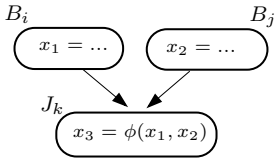


Figure 7. A ϕ function

is represented in SIGNAL by the equation:

$$x_3 := x_1\ when\ B_i \quad default \quad x_2\ when\ B_j \qquad (1)$$

where, x_1 is defined on block B_i and x_2 is defined on block B_j.

Equation (1) means that x_3 takes the value of x_1 (resp. x_2) when signal B_i (resp. B_j) is present and true. Note that, the value of x is depending on the present value of B_i and B_j and not on their previous value.

4.1.4 Encoding Conditional Branching Statements

Each SSA Test block defines a conditional branching statement. In Figure 8, the execution of the two successors blocks B_j and B_k of a Test block T_m depends on the value of its conditional expression $(y_1 > z_1)$. Therefore, statement of T_m defines B_j and B_k in SIGNAL by the equations:

$$\begin{array}{lcl} B_j & := & true\ when \quad (y_1 > z_1) \quad default \quad false \\ B_k & := & true\ when\ not\ (y_1 > z_1) \quad default \quad false \end{array}$$

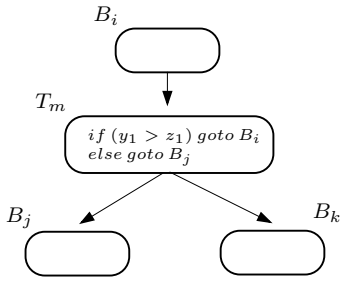


Figure 8. A Conditional Branching Statements

However, The execution of the conditional expression $(y_1 > z_1)$ of T_m depends on the signal of its predecessor block B_i. Therefore, the new equations of B_j and B_k are given by:

$$\begin{array}{lcl} B_j & := & true\ when\ (y_1 > z_1)\ when\ B_i \\ & & default\ false \\ B_k & := & true\ when\ not\ (y_1 > z_1)\ when\ B_i \\ & & default false \end{array}$$

4.1.5 Encoding Loop Statements

Figure 9 shows how a followed loop statement is represented on SSA:

```
{ ...
  for(i=1;i<10;i++)
  {
    ...
  }
 ... }
```

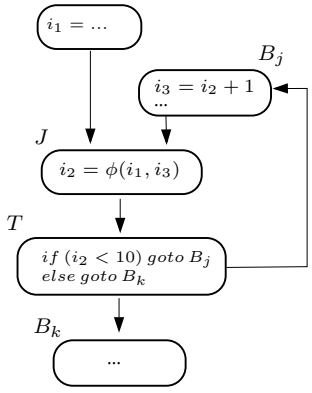


Figure 9. A Loop Statement

In this SSA graph the block B_j of the loop statement is defined before its conditional expression $(i_2 < 10)$, and its execution is depend on the value of this conditional expression. We use the same rules defined on Sections 4.1.1

and 4.1.3 to encode blocks B_i, B_k and J on SIGNAL. However, statements of block B_j are defined before their use in the SSA graph. Therefore, we use the delay operator \$ of SIGNAL to encode statements of this block. We encode each variable used in statements of block B_j and defined after the block B_j by its previous value.

In our example, we encode the statement of B_j in SIGNAL by the followed equation:

$$i_3 := i_2\$ \ + 1 \quad when\ B_j \tag{2}$$

where, $i_2\$$ is the previous value of i_2.

The equations of B_j and B_k are given by:

$$\begin{array}{lll} B_j & := & true\ when\ (i_2\$ < 10)\ when\ \ pre_B_j \\ & & default\ \ false \\ B_k & := & true\ when\ not\ (i_2 < 10)\ when\ \ B_j \\ & & default\ \ false \end{array}$$

where,

$$pre_B_j := B_j\$\ \ init\ \ true \tag{3}$$

4.2. Encoding Pointers

On of the major problem designers encounter during the validation of C/C++ models is pointer alias analysis. In this section, we propose a solution as how to encode pointers in SIGNAL that allows fast alias analysis. Our solution is based on the approach proposed in [17] to encode pointer variables for the behavioral synthesis from C. The encoding process is performed in two steps:

1. The first step is applicable at SSA level, where we replace each pointer p by two new variables: p_tag and $start_p$. Since each pointer p may point to more than one variable - as is the case for arrays for example -, we associate to each pointer variable p points another variable p_tag with an integer value that represents the index within an array. Thus, the value stored in the variable p_tag refers to the index within the variable pointed to by the pointer p. For example, $(p_tag = 0)$ means that p points-to the first variable of the set that p points-to. The second variable $start_p$ defines the value of this variable. As a result, we first replace each occurrence of the assignment statement $(p = \&x_1)$ in SSA graph by $(p_tag = i)$, assuming that x_1 is the i^{th} variable of the set that p points-to. Then, assuming that y_1 is the first variable of the set that p points-to, we replace each assignment statement of the form "$a_1 = f(\ldots, *p, \ldots)$" by the conditional statements:

$$\begin{array}{lll} & if(p_tag == 0) & start_p_m = y_1 \\ else & \ldots & \ldots \\ else & if(p_tag == i) & start_p_m = x_1 \\ else & \ldots & \ldots \end{array} \tag{4}$$

followed by the assignment statement:

$$a_1 = f(\ldots, start_p_m, \ldots) \tag{5}$$

In our transformation, we apply SSA renaming variables to $start_p$ only for different conditional statements, and we may use more than one definition for p_tag inside a procedure or function.

2. The second step to encode pointers consists of transforming each assignment $(p_tag = i)$ of block B_k into SIGNAL by the following partial equation (p_tag may be defined in one or more SSA blocks):

$$p_tag \ ::= \ i\ when\ B_k$$

A partial equation "$x \ ::= \ \ldots$" in SIGNAL allows us to give multiple definitions to a signal x, provided that these clocks are mutually exclusive (and hence the resulting signal definition deterministic).

In this step, we transform the conditional statements of equation (4) of block B_k into:

$$\begin{array}{lll} start_p_1 & := & \ \ y_1 \quad when\ (p_tag = 0)\ when\ B_k \\ & & default \\ & & \ \ \ldots \\ & & default \\ & & \ \ x_1 \quad when\ (p_tag = i)\ when\ B_k \\ & & default \\ & & \ldots \end{array} \tag{6}$$

and, we transform the assignment statement of equation (5) of block B_k into:

$$a_1 \ := \ f(\ldots, start_p_m, \ldots) \ \ when\ B_k \tag{7}$$

5. Example

Figure 10 gives a small example of our approach by showing the corresponding SIGNAL formal model of the SSA form of Figure 4.

$$
\begin{array}{lcl}
B_1 & := & true\ when\ start_proc\ default false \\
y_1 & := & (a_1 * b_1)\ when\ B_1 \\
z_1 & := & (a_1 * c_1)\ when\ B_1 \\
\\
B_2 & := & true\ when\ (y_1 > z_1)\ when\ B_1\ default\ false \\
x_1 & := & (y_1 - z_1)\ when\ B_2 \\
\\
B_3 & := & true\ when\ not\ (y_1 > z_1)\ when\ B_1\ default\ false \\
x_2 & := & (z_1 - y_1)\ when\ B_3 \\
\\
x_3 & := & x_1\ when\ B_2\ default\ x_2\ when\ B_3 \\
\\
B_4 & := & true\ when\ B_2\ default\ true\ when\ B_3\ default \\
 & & false \\
result & := & (x_3 * a_1)\ when\ B_4
\end{array}
$$

Figure 10. SIGNAL equations of SSA code of Figure 4

Here, the SSA form of Figure 10 represents a procedure with three inputs $\{a, b, c\}$. This procedure is transformed into the following SIGNAL process:

process NAME =
(? *integer* a_1, b_1, c_1; *boolean start_proc*;
! *integer results*)
(| equations of Figure 10
| $B_1\hat{} = B_2\hat{} = B_3\hat{} = B_4$
|)
where
boolean B_1, B_2, B_3, B_4;
integer x_1, x_2, x_3, y_1, z_1;
end

The procedure is activated only if the boolean signal *start_proc* is present and its value is true. Therefore, the signal B_1 of its first block is conditioned by this new input signal. Finally, we synchronize all boolean signals in the last equations of the signal process.

6. 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* [9]. This allows an easy use of our approach by designers using a common software programming language to describe their systems. In this work, as we are targeting C/C++ system models for which we have implemented our SSA to SIGNAL translation using GCC. Designers using C/C++ can easily integrate our translator in their installed C/C++ software programming framework. We are using the GCC version 4, which implements a new optimization framework (Tree-SSA [22]) based on SSA that operates on GCC's tree representation. Figure 11 describes our translation scheme as a two step process:

1. **Converting C/C++ into SSA:** The first step of our 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 [11]; 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. The transformations of this step are implemented in the files "gimplify.c", "c-simplify.c", "tree-cfg.c" and "tree-ssa.c" of GCC version 4. An overview of the SSA implementation in GCC is presented in [6].

2. **Converting SSA into SIGNAL:** The next step of our translation scheme consists in converting SSA into SIGNAL processes. The translation scheme is implemented in the GCC front end, where we have integrated new C file called "ssa2signal.c". This file implements several debugging functions which represent all the transformation rules presented in Section 4. The output of this step is a SIGNAL program. As we can see in Figure 11, our implementation of the SIGNAL generation is inserted in the GCC source tree as an *additional Front end optimization pass*. GCC 4 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 input code they may result in less performing code.

One advantage of our translation scheme is that systems modeled using some programming languages that are supported by GCC other than C/C++, such as Java and Fortran, can be easily translated into SIGNAL processes with no additional effort (see Figure 11). Finally, in our approach to translate C/C++ programs into SIGNAL process, designers can optimize their code by using first the GCC optimization passes based on SSA, and then the SIGNAL compiler optimization (see Figure 11).

7. Related Work

The idea of validating C/C++ system design using formal approach rather than simulator and test tools have been discussed on previews work. Authors of [2, 19, 21, 20] present approaches for validating modular system design. They describe how SystemC [7] imperative components can be transformed into a formal model. [1] presents a front

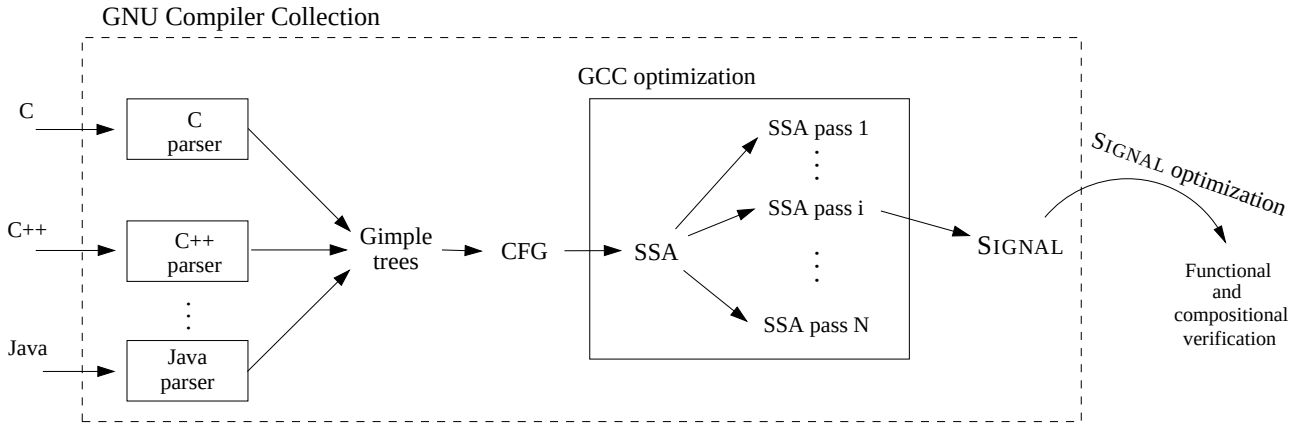


Figure 11. Translating C/C++ models into SIGNAL processes

end for SystemC called SystemCXML that uses an XML-based approach to extract structural information from SystemC models, which can be easily exploited by back end passes for analysis, visualization and other structural analysis purposes.

A different approach is illustrated in [13, 12] where a SystemC front end is created that uses GCC's front end to parse all C++ constructs and infer the structural information of the SystemC model by executing the elaboration phase. Pinapa examines the data structures of SystemC's scheduler and creates its own IR.

As in [1, 2, 19, 21, 20], we use the synchronous approach to validate system designs. The main contribution of this paper is to implement and to extend these works, and automate the actual translation process. In particular, we propose a solution to encode pointer variables in SIGNAL which allows easy pointers analysis to be performed. Our approach is implemented using GCC which is an advantage for designers using C/C++ to describe their system models as well as it can benefit of present and future optimization passes within the GCC.

8. Conclusion

In this work, we try to improve functional and compositional correctness of IP-based system design using the C/C++ programming language. We propose an approach to automatically create formal models from C/C++ system models. A formal model allows designers to verify the correctness of their design using methods of mathematical proof rather than simulation and testing to ensure the quality of the design. Our solution is based on the internal representation SSA of GCC and uses the synchronous language SIGNAL as a formal platform.

Using GCC we have built a compiler that translates, SSA representations into SIGNAL processes. SSA representations can be automatically generated by the GCC compiler. Our implementation of the SIGNAL generation is inserted in the GNU Compiler Collection source code as an *additional Front end optimization pass*. Right now, we are working on reducing the number of variables generated by our translator and on implementing array conversions into SIGNAL.

References

[1] D. Berner, H. Patel, D. Mathaikutty, J.-P. Talpin, and S. Shukla. SystemCXML: An extensible SystemC front end using XML. In *Proceedings of the Forum on specifi cation and design languages (FDL)*, Lausanne, Switzerland,, September 2005.

[2] D. Berner, J.-P. Talpin, S. K. Shukla, and P. Le Guernic. Modular design through component abstraction. In *Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*, pages 202–211, Washington DC, USA, September 2004.

[3] M. L. Borgne, H. Marchand, E. Rutten, and M. Samaan. Formal verifi cation of signal programs: Application to a power transformer station controller. In Springer-Verlag, editor, *Proceedings of the Fifth International Conference on Algebraic Methodology and Software Technology AMAST'96*, pages 271–285, Munich, Germany, Juillet 1996.

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

[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] T. Grotker. *System Design with SystemC*. Kluwer Academic Publishers, Norwell, MA, USA, 2002.

[8] A. Hall. Seven myths of formal methods. *IEEE Software*, 7(5):11–19, Sept. 1990.

[9] Jalapeno research project. Java virtual machine-jikes rvm. http://jikesrvm.sourceforge.net.

[10] P. Le Guernic, J.-P. Talpin, and J.-C. Le Lann. Polychrony for system design. *Circuits, Systems and Computers, Special Issue on Application Specifi c Hardware Design*, 2003.

[11] J. Merrill. Generic and gimple: A new tree representation for entire functions. In *Proceedings of the 2003 GCC Summi*, Ottawa, Canada, May 2003.

[12] 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*, pages 26–35. IEEE Computer Society, 2005.

[13] 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, New York, NY, USA, 2005. ACM Press.

[14] D. Novillo. Tree ssa — a new high-level optimization framework for the gnu compiler collection. *Proceedings of the Nord/USENIX Users Conference*, February 2003.

[15] D. Novillo. Design and implementation of tree-ssa. In *GCC Summit Proceedings*, Ottawa, Canada, 2004.

[16] OSCI, SC-HDL and NC-SystemC. Simulators for systemc. http://www.systemc.org/.

[17] L. Semeria and G. D. Micheli. Resolution, optimization, and encoding of pointer variables for the behavioral synthesis from c. In *IEEE Transactions on Computer Aided Design*, pages 213–233, Feb. 2001.

[18] C. Stoud, R. Munoz, and D. Pierce. Behavioral model synthesis with cones. *IEEE Design & Test of Computers*, 5(3):22–30, June 1988.

[19] J.-P. Talpin, D. Berner, S. Shukla, A. Gamatié, P. Le Guernic, and R. Gupta. A behavioral type inference system for compositional system-on-chip design. In *International Conference on Application of Concurrency to System Design (ACSD)*, Hamilton, Canada, June 2004.

[20] J.-P. Talpin, D. Berner, S. K. Shukla, A. Gamatié, P. Le Guernic, and R. Gupta. *Formal Methods and Models for System Design*, chapter Behavioral Type Inference for Compositional System Design. Kluwer Academic Publishers, June 2004.

[21] J.-P. Talpin, P. Le Guernic, S. Shukla, and R. Gupta. Compositional behavioral modeling of embedded systems and conformance checking. *International Journal on Parallel processing, special issue on testing of embedded systems*, 2005.

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

[23] J. Zhu and W. S. Mong. Specifi cation of non-functional intellectual property components. In *Proceedings of the conference on Design, Automation and Test in Europe (DATE'03)*, Washington, DC, USA, 2003. IEEE Computer Society.