Evaluating SDVG translation validation: from Signal to C

van Chan Ngo, Jean-Pierre Talpin, Thierry Gautier, Paul Le Guernic

Evaluating SDVG translation validation: from Signal to C

Van Chan Ngo, Jean-Pierre Talpin, Thierry Gautier, Paul Le Guernic

RESEARCH REPORT
N° 8508
March 2014
Project-Team ESPRESSO

ISSN 0249-6399 ISRN INRIA/RR--8508--FR+ENG

Evaluating SDVG translation validation: from Signal to C

Van Chan Ngo, Jean-Pierre Talpin, Thierry Gautier, Paul Le Guernic

Project-Team ESPRESSO

Research Report n° 8508 — March 2014 — 43 pages

Abstract: In this work, we describe how the preservation of value-equivalence of variables can be proved based on translation validation of synchronous data-flow value-graphs. It focuses on proving that every output variables in the original program and their counterparts in the transformed program, the generated C code, have the same values. The computation of each output variable and its counterpart is represented by a formal representation, a shared value-graph.

This graph deterministically represents the computation of the output in the original program and its counterpart in the transformed program, and the nodes for the common variables have been shared in the graph. Given a SDVG, support that we want to show that the two output variables have the same value. We simply need to check that they are represented by graphs which are rooted at the same graph node. We manage to make the check by *normalizing* SDVGs by some rewrite rules.

Key-words: Formal Verification, Translation Validation, Certified Compiler, Value-Graphs, Synchronous Programs

RESEARCH CENTRE
RENNES – BRETAGNE ATLANTIQUE

Campus universitaire de Beaulieu
35042 Rennes Cedex

Translation Validation d'évaluation de SDVG: de Signal vers C

Résumé : Dans ce travail, nous décrivons comment la préservation de la valeur d'équivalence de variables peut être prouvée sur la base de la translation validation de valeur-graphe synchrone. Il se concentre sur ce qui prouve que toutes les variables de sortie dans le programme d'origine et leurs correspondants dans le programme transformé, le code C généré, ont les mêmes valeurs. Le calcul de chaque variable de sortie et son correspondant est représentée par une représentation formelle, un valeur-graphe partagé.

Ce graphique représente la facon déterministe le calcul de la sortie dans le programme d'origine et son équivalent dans le programme transformé, et les noeuds pour les variables communes ont été partagées dans le graphique. Étant donné un SDVG, soutenons que nous voulons montrer que les deux variables de sortie ont la même valeur. Nous avons simplement besoin de vérifier qu'ils sont représentés par des graphiques qui sont enracinées dans le même noeud du graphe. Nous parvenons à faire le chèque en normalisant SDVGs par des règles de réécriture.

Mots-clés : Formal Verification, Translation Validation, Certified Compiler, Value-Graphs, Synchronous Programs

1 Introduction

At a high level, our tool works as follows. For a transformation, it takes the input program and its transformed counterpart, constructs the corresponding SDVG for each output variable. Then it checks that for every output variable in input program and its counterpart in transformed program, they have the same value. If the result says that there exists any non-equivalence then the compiler emits compilation error. Otherwise, the compiler continues its work. The integration of this verification process into the compilation process can be depicted as in Figure 1.

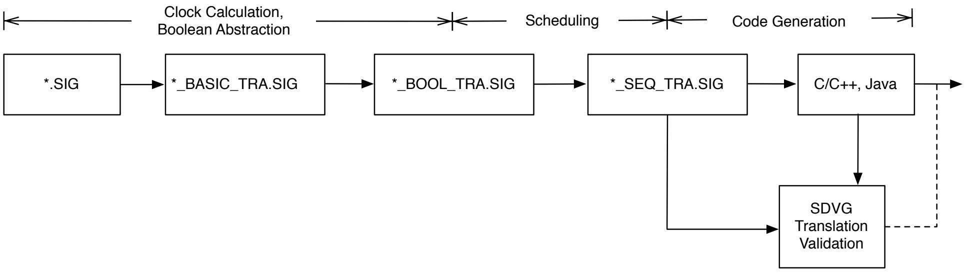


Figure 1: A bird's-eye view of the verification process

We believe that our validator must have the following features to be effective. First, we do not modify or instrument the compiler, and we treat the compiler as a "black box", hence our validator is not affected from the future update or modification of the compiler. Our approach is to apply the verification to the compiler transformations themselves in order to automatically generate formal evidence that the clock semantics of the source program is preserved during program transformations, as per applicable qualification standard. Second, it is important that the validator can be scaled to large programs, in which we represent the desired program semantics with our scalable abstraction and use efficient graph libraries to achieve the expected goals: traceability and formal evidence.

The remainder of this chapter is organized as follows. Section 3 describes how code can be generated, as the final step of the compilation process, following different schemes. Section 4 illustrates the concept of SDVG and the verification procedure. In Section 5, we consider the formal definition of SDVG and the representation of Signal program and generated C code as SDVGs. Section 6 addresses the mechanism of the verification process based on the rewrite rules and the normalization of a SDVG methods, the application of the verification process to the Signal compiler, and its integration in the Polychrony toolset [1]. Section 7 presents some related works, concludes our work and outlines future directions.

2 The Signal language

2.1 Language features

Signal [2, 3] is a polychronous data-flow language that allows the specification of multi-clocked systems. Signal handles unbounded sequences of typed values $(x(t))_{t\in\mathbb{N}}$, called *signals*, denoted as x. Each signal is implicitly indexed by a logical *clock* indicating the set of instants at which the signal is present, noted C_x. At a given instant, a signal may be present where it holds a value, or absent where it holds no value (denoted by #). Given two signals, they are *synchronous* if and only if they have the same clock. In Signal, a process (written P or Q) consists of the synchronous composition (noted $|$) of equations over signals x, y, z, written $x := y\, op\, z$ or $x := op(y, z)$, where op is an operator. A program is a process.

2.1.1 Data domains

Data types consist of usual scalar types (Boolean, integer, float, complex, and character), enumerated types, array types, tuple types, and the special type *event*, subtype of the Boolean type which has only one value, `true`.

2.1.2 Operators

The *core language* consists of two kinds of "statements" defined by the following primitive operators: first four operators on signals and last two operators on processes. The operators on signals define basic processes (with implicit clock relations) while the operators on processes are used to construct complex processes with the parallel composition operator:

- *Stepwise functions:* $y := f(x_1, ..., x_n)$, where f is a *n-ary* function on values, defines the extended stream function over synchronous signals as a basic process whose output y is synchronous with $x_1, ..., x_n$ and $\forall t \in C_y, y(t) = f(x_1(t), ..., x_n(t))$. The implicit clock relation is $C_y = C_{x_1} = ... = C_{x_n}$.
- *Delay:* $y := x\$1$ `init` a defines a basic process such that y and x are synchronous, $y(0) = a$, and $\forall t \in C_y \wedge t > 0, y(t) = x(t-1)$. The implicit clock is $C_y = C_x$.
- *Merge:* $y := x$ `default` z defines a basic process which specifies that y is present if and only if x or z is present, and that $y(t) = x(t)$ if $t \in C_x$ and $y(t) = z(t)$ if $t \in C_z \setminus C_x$. The implicit clock relation is $C_y = C_x \cup C_z$.
- *Sampling:* $y := x$ `when` b where b is a Boolean signal, defines a basic process such that $\forall t \in C_x \cap C_b \wedge b(t) = \texttt{true}, y(t) = x(t)$, and otherwise, y is absent. The implicit clock relation is $C_y = C_x \cap [b]$, where the sub-clock $[b]$ is defined as $\{t \in C_b | b(t) = \texttt{true}\}$.
- *Composition:* If P_1 and P_2 are processes, then $P_1 \mid P_2$, also denoted $(|P_1 \mid P_2|)$, is the process resulting of their parallel composition. This process consists of the composition of the systems of equations. The composition operator is commutative, associative, and idempotent.
- *Restriction:* P `where` x, where P is a process and x is a signal, specifies a process by considering x as local variable to P (i.e., x is not accessible from outside P).

2.1.3 Clock relations

In addition, the language allows clock constraints to be defined explicitly by some derived operators that can be replaced by primitive operators above. For instance, to define the clock of a signal (represented as an *event* type signal), $y := \hat{}x$ specifies that y is the clock of x; it is equivalent to $y := (x = x)$ in the core language. The synchronization $x \hat{=} y$ means that x and y have the same clock, it can be replaced by $\hat{}x = \hat{}y$. The clock extraction from a Boolean signal is denoted by a unary `when`: `when` b, that is a shortcut for b `when` b. The clock union $x \hat{+} y$ defines a clock as the union $C_x \cup C_y$, which can be rewritten as $\hat{}x$ `default` $\hat{}y$. In the same way, the clock intersection $x \hat{*}\, y$ and the clock difference $x \hat{-}\, y$ define clocks $C_x \cap C_y$ and $C_x \setminus C_y$, which can be rewritten as $\hat{}x$ `when` $\hat{}y$ and `when` (`not`($\hat{}y$) `default` $\hat{}x$), respectively.

2.1.4 Example

The following Signal program emits a sequence of values $FB, FB-1, ..., 2, 1$, from each value of a positive integer signal FB coming from its environment:

```
1 process DEC=
2 (? integer FB;
3  ! integer N)
4  (| FB ^= when (ZN<=1)
5   | N := FB default (ZN-1)
6   | ZN := N$1 init 1
7   |)
8 where integer ZN init 1
9 end;
```

Let us comment this program:

- Lines (2) and (3): FB, N are respectively input and output signals of type *integer*.
- Line (4): FB is accepted (or it is present) only when ZN becomes less than or equal to 1.
- Line (5): N is set to FB when its previous value is less than or equal to 1, otherwise it is decremented by 1.
- Line (6): defines ZN as always carrying the previous value of N (the initial value of ZN is 1).
- Line (8): indicates that ZN is a local signal whose initial value is 1.

Note that the clock of the output signal is more frequent than that of the input. This is illustrated in the following possible trace:

```
  t  .   .   .   .   .   .   .   .   .   .
FB   6   #   #   #   #   #   3   #   #   2
ZN   1   6   5   4   3   2   1   3   2   1
 N   6   5   4   3   2   1   3   2   1   2
CFB  t0                      t6          t9
CZN  t0  t1  t2  t3  t4  t5  t6  t7  t8  t9
CN   t0  t1  t2  t3  t4  t5  t6  t7  t8  t9
```

2.1.5 Program structure

The language is modular. In particular, a process can be used as a basic pattern, by means of an interface that describes its parameters and its input and output signals. Moreover, a process can use other subprocesses, or even external parameter processes that are only known by their interfaces. For example, to emit three sequences of values $(FB_i) - 1, ..., 2, 1$ for all three positive integer inputs FB_i, with $i = 1, 2, 3$, one can define the following process (in which, without additional synchronizations, the three subprocesses have unrelated clocks):

```
process 3DEC=
(? integer FB1, FB2, FB3;
 ! integer N1, N2, N3)
 (| N1 := DEC(FB1)
  | N1 := DEC(FB2)
  | N3 := DEC(FB3)
  |)
end;
```

3 Code generation in Signal compiler

Code generation is the final step in the compilation process of Signal compiler as depicted in Figure 2. When a program P consists of no deadlocks, free of clock constraints one could generate code for P following the Kahn semantics with the code generation functionalities of Polychrony toolset. The code is generated for different general purpose languages (C, C++, Java) on different architectures. The generated code in this case is called *reactive code*. However, one can generate the *defensive code* when the program consists of some clock constraints, in this mode, all alarms are emitted when a contraint is violated during the simulation.

3.1 The principle

The principle of code generation [4] is based on the use of the clock hierarchy resulting from the clock calculation and the graph of conditional dependencies not only to schedule the instructions in sequences, but also the schedule component activation in a hierarchical target code. The code generation follows the general scheme that is depicted in Figure 2. The generated code contains a main program which controls the *step block*. The step block consists of a step scheduler that drives the execution of its step component and updates the state variables corresponding to delay operators and local variables. The execution of the step block is scheduled by the step scheduler. The step component can be hierarchical which consists of a set of sub-components called *clusters* and has its own local step scheduler. The step block communicates with its environment through the IO container.

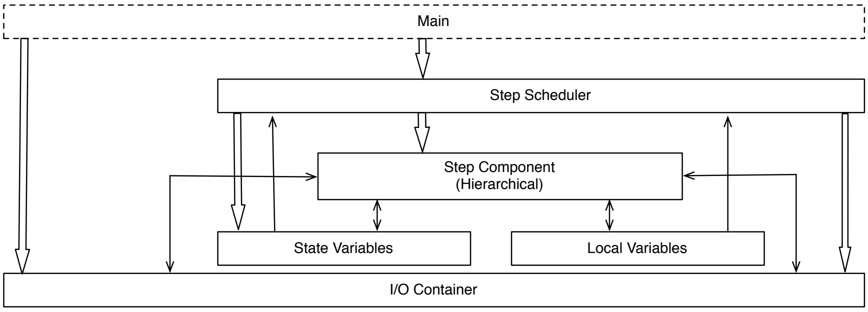


Figure 2: Code generation: General scheme

In general, the generated program will have many files. We consider the target language code is C. For a process P, a main program is defined in the file *P_main.c*, the program which contains the step block called body program is defined in the file *P_body.c*, and an input-output program which contains the IO container is defined in the file *P_io.c*. Each component of the generated code can be seen as a Signal process. Then they can be reused in an embedding Signal process.

3.1.1 The main program

The pseudo-code in Listing 1 shows the structure of the main program. The main program opens the IO communication channels with the program environment, and calls the initialization function. If everything goes fine then it calls the step function repeatedly in the infinite loop to interact with the environment. The infinite loop can be stopped if the step function returns a error code 0, meanings that the input stream is empty, and the main program will close the communication channels. All the called functions are defined in the body program.

Listing 1 : Structure of P_main.c

```
EXTERN int main()
{
   logical code;
   P_OpenIO();
   code = P_initialize();
   while(code)
   {
      P_stepIO_begin();
      code = P_step();
      P_stepIO_end();
   }
   P_CloseIO();
}
```

3.1.2 The step block

Once the IO communication channels and the initialization are completed, the step function $P\_step()$ is responsible for the calculation of the effect of one synchronous step of the system to interact with the environment, is the essential part of the concrete code. It reads data from the input streams, computes the outputs and writes the results to the output streams. In Polychrony toolset, the implementation of the step function can be done in many schemes based on the clock hierarchy and the graph of conditional dependencies which are produced in the previous compilation phases by the compiler front-end. These code generation schemes consist of:

- Global code generation: sequential code, clustered code with static scheduling, clustered code with dynamic scheduling.
- Modular code generation.
- Distributed code generation.

The next section will describe the sequential, inlining code generation of the step block. For other code generation schemes, interested readers can refer to [5, 6, 7].

3.1.3 The IO container

The IO container implements the communication of the generated program with the environment in case the being compiled process contains input and output signals. In the simulation mode, each input or output signal communicates with the environment via a file as the input stream or output stream. The IO container in Listing 2 consists of global functions for opening, closing all files, and for reading and writing data for each input and output signal.

Listing 2 : Structure of P_io.c

```
EXTERN void P_OpenIO()     EXTERN void P_CloseIO()
{                          {
   fra = fopen(...);          ...
   if (!fra) {                fclose(fwx);
      ...                  }
      exit(1);
   }                       EXTERN int r_P_a(integer *a)
   fwx = fopen(...);       {
   if (!fwx) {                return (fscanf(fra,"%d",a) != EOF);
      ...                  }
      exit(1);
   }                       EXTERN void w_P_x(integer x)
   ...                     {
}                             fprintf(fwx,"%d",x);
                              fprintf(fwx,"\n");
                              fflush(fwx);
                           }
```

3.2 Sequential code generation

In the context of this work, we will consider the sequential, inlining code generation scheme for the step function that directly interprets the Signal process obtained after the clock calculation, boolean abstraction, and scheduling phases of the compiler front-end. We describe the code structure of the step function for a simple process in Listing 3. The step function obtained by compiling it is given in Listing 4. The C code introduces an explicit variable for each signal to represent the clock. Variable C_N is the clock of N and C_FB1, C_FB2 are the clocks of $FB1$ and $FB2$, respectively. As soon as the clock is evaluated and is `true`, the signal is read if it is an input signal or updated, otherwise. The state variables (corresponding to the delays) are updated at the end of the step block in the step finalization. The precedence of the statements must be consistent with the graph of conditional dependencies, and one can observe that the tree structure of conditional if-then-else statements which express directly the clock hierarchy.

The step function works as follows. It reads the clock values of $FB1$ and $FB2$. If C_FB2, the clock of $FB2$, has the value `true`, a new value for $FB2$ is read and used to compute the clock of N. In the similar way, if C_FB1 has the value `true`, a new value for $FB1$ is read. If C_N, the clock of N, has the value `true`, N gets the value $4 * FB1$. The updated value of N is also output.

A computation of this program is given below. At the initialization, the variables can have arbitrary values which are denoted by $*$.

```
FB1    *   1   2   2    3   5   4   6   9    ...
FB2    *   3   0   1    5   4   2   6   2    ...
  N    *   4   4   4   12  20  20  24  24    ...
```

Taking into account that N in C code is the value of the corresponding signal in Signal program, we have an observation that the value of N is remained when C_N, the clock of N, has the value `false`. Intuitively, based on this observation, we can say that if a variable in the generated C program whose value is never updated then it will be assigned no value, denoted as $\#$. In the next sections, we will show how this assumption can be formalized to represent the computation of step function as a shared value-graph.

4 Illustrative example

We begin by showing how our verification process works for an illustrative example. Consider the following synchronous program WHENOP written in Signal language which is given in Listing 3.

Listing 3 : Program WHENOP in Signal

```
process WHENOP=
(? integer FB1; integer FB2;
 ! integer N)
(| N := 4*FB1 when (FB2>=3)
 |)
end;
```

And WHENOP_step is *step function* of the generated C code. This function which is called repeatedly in an infinite loop, simulates one synchronous step of the Signal program which is shown in Listing 4.

Listing 4 : Synchronous Step of WHENOP

```
EXTERN logical WHENOP_step()
{
   if (!r_WHENOP_C_FB1(&C_FB1))
      return FALSE;
   if (!r_WHENOP_C_FB2(&C_FB2))
      return FALSE;
   if (C_FB2)
   {
      if (!r_WHENOP_FB2(&FB2))
         return FALSE;
   }
   C_CLK_36 = (C_FB2 ? (FB2 >= 3)
              : FALSE);
   C_N = C_FB1 && C_CLK_36;
   if (C_FB1)
   {
      if (!r_WHENOP_FB1(&FB1))
         return FALSE;
   }
   if (C_N)
   {
      N = 4 * FB1;
      w_WHENOP_N(N);
   }
   WHENOP_step_finalize();
   return TRUE;
}
```

In this example, we use the concept of *gated* ϕ-function such as $x = \phi(c, x_1, x_2)$ which is mentioned more details in the next sections. It is used to represent the branching in program, which means x takes the value of x_1 if the condition c is satisfied, and the value of x_2, otherwise. Since the generated C programs use *persistent* variables (i.e. variables that always have some values), while Signal programs which use *volatile* variables, we will assume that if a variable (including the input and output variables) in the generated C program whose value is never updated then it will be assigned the absent value, denoted as $\#$. And if a statement involves a variable x before its value update then the value of this variable is the previous value, denoted as $m.x$.

Considering the equation $N := 4 * FB1$ when $(FB2 >= 3)$, at a considered instant t, signal N is present if signal $FB1, FB2$ are present and $FB2$ is greater than or equal to 3. When N is present, its value is defined by the value of $FB1$ multiplied by 4. The value of N is # when it is absent. The computation of this equation can be replaced by the following gated ϕ-function:

$$N = \phi(\widehat{N}, \overline{N}, \#)$$

where $\widehat{N} \Leftrightarrow (\widehat{FB1} \wedge \widehat{FB2} \wedge (\overline{FB2} >= 3))$, $\overline{N} = 4 * \overline{FB1}$. Here, $\widehat{N}, \widehat{FB1}, \widehat{FB2}$ are boolean variables that represent the states (`false`: absent, `true`: present) of signals $N, FB1$ and $FB2$ at instant t, respectively. And $\overline{N}, \overline{FB1}, \overline{FB2}$ are values of signals $N, FB1$ and $FB2$ with the same types. Then, this gated ϕ-function indicates that at any instant t such that signals $FB1$ and $FB2$ are present and the value of $FB2$ is greater than or equal to 3, then the value of N is equal to the value of $FB1$ multiplied by 4, otherwise the value of N is #.

In the same way, we use a gated ϕ-function to represent the branching in C code. For instance, if C_N is `true` then the value of variable N is defined by the value of $FB1$ multiplied by 4. Otherwise, the value of N is never updated. This computation can be replaced by the following gated ϕ-function:

$$N = \phi(C\_N, 4 * FB1, \#)$$

We replace the variables C_CLK_36 and C_N by their definition, we obtain the synchronous data-flow value-graph for the output N that is presented in Figure 3. Notice that the compiler prefers to write $C\_CLK\_36 = (C\_FB2?(FB2 >= 3) : FALSE)$ instead of $C\_CLK\_36 = C\_FB2\&\&(FB2 >= 3)$, which can be represented by the following gated ϕ-function:

$$C\_CLK\_36 = \phi(C\_FB2, FB2 >= 3, false)$$

The dashed arrows are not parts of the graph, they only mean that for each node, there is a set of labels that indicates which nodes of the graph correspond to which signals, clocks or variables in the programs. The Signal program and its generated C program have been represented in the same graph, in which the nodes are labelled by the same structures (clocks, signals, variables and function symbols) have been reused. And the unique occurrence of a reused node is said to be *shared*. For example, the nodes labelled >= and # are shared in the graph.

Here, the values of input signals and their corresponding variables in the generated C code are represented by the same nodes in the shared graph. In general, it is safe to assume that the values of input signals and the corresponding variables in the generated C code are equal. Thus, in the shared graph, the input signal values $\overline{FB1}, \overline{FB2}$ and the variables $FB1, FB2$ in the C code are represented by the same nodes.

Suppose that we want to verify that the signal N in the `WHENOP` and `WHENOP_step` (denoted by N^c) will have the same computation. This can be done by showing that they are represented by the subgraphs which are rooted at the same node. In Figure 3, we cannot conclude that they are equivalent, however we can transform the value-graph by applying *normalization* rules. First, by exploiting the generated program, C_FB1, C_FB2 are clocks of $FB1$ and $FB2$, respectively. Because the values of inputs $FB1, FB2$ are updated only when C_FB1, C_FB2 are valid. The first rule, we will apply is that:

"If x is an input and the clock of x is read as input parameter (it is not defined in the program) then its clocks in Signal program and C code are represented by the same node".

Thus, they are represented by the same nodes C_FB1 and C_FB2 in Figure 4. Until now, the subgraphs represent the variable N in the two programs which are not rooted at the same node. We will apply the following second rewrite rule to the resulting graph, we will replace $\phi(C\_FB2, \overline{FB2} >= 3, \texttt{false})$ with $C\_FB2 \wedge \overline{FB2}$

$\phi(c, x, false)$ is replaced by $(c \wedge x)$ for any boolean expression x.

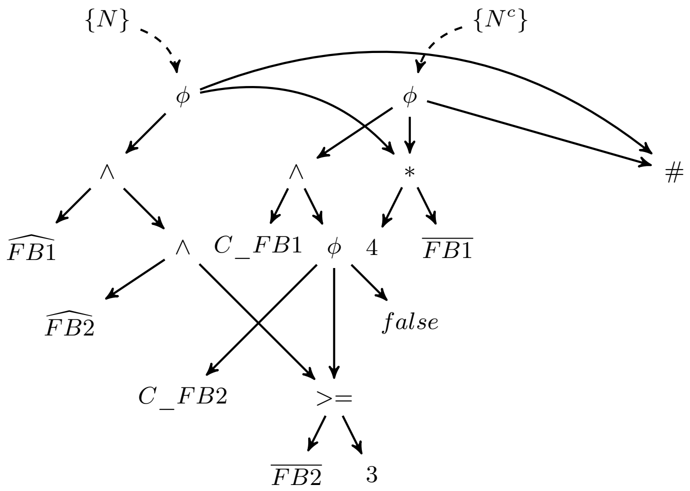


Figure 3: The shared value-graph of WHENOP and its generated C code

We will go into details about the rules in the next section. After this replacement, and maximizing the variable sharing, the variable N in two programs points to the same node in the resulting graph in Figure 5. Therefore, we can conclude that the outputs are equivalent.

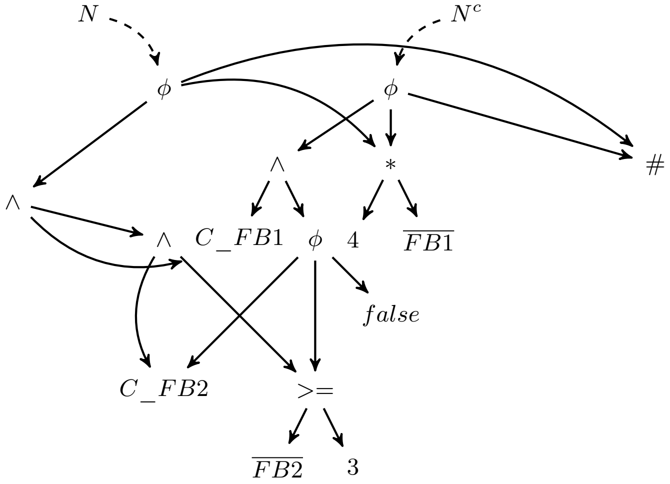


Figure 4: The resulting transformed value-graph

5 Synchronous data-flow value-graph

In this section, we describe the computation of signal in a Signal program and the corresponding variable in the generated C code in terms of SDVGs. Let us recall the computation of the output signal y in the equation $y := x$ when b in the previous section. At any instant, the signal y holds the value of x if the following conditions are satisfied:

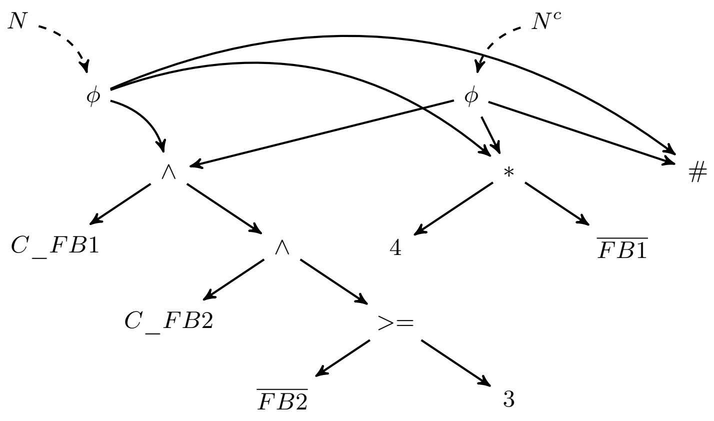


Figure 5: The resulting transformed value-graph

- x and b are defined.
- b holds the value `true`.

Otherwise, it holds no value. Thus, at a given instant, to represent the underlying control conditions and the computation of this equation, we can use the following gated ϕ-function:

$$y = \phi(\hat{x} \wedge \hat{b} \wedge \bar{b}, \bar{x}, \#)$$

The condition $(\hat{x} \wedge \hat{b} \wedge \bar{b})$ represents the state in which x holds a value ($\hat{x} =$ `true`), and b holds the value `true` ($\hat{b} = true \wedge \bar{b} = true$). This section explores a method to construct that shared value-graph for both Signal and generated C code programs, which is the computational model of our translation validation approach.

5.1 Definition of SDVG

Graphs can be used to describe many structures in computer science: program control flows, communication processes, computer networks, pointer structure on the heap and many others. In fact, for most activities in the software development, many types of visual notations have been introduced, including UML, state diagrams, control flows graphs, block diagrams. These notations construct a models that can be seen as graphs. This section intends to focus on graphs which represent expressions for computing the variable values in programs. We presents basic definitions, including the notion of gated ϕ-function, and introduces a linear syntax presentation for terms represented as graphs. Finally, we provide the definition of our considered type of graph, synchronous data-flow value-graph. The interested readers can refer to [8] for more detailed discussion on term graphs and linear syntax presentation for graphs.

5.1.1 Gated ϕ-function

In Static Single Assignment (SSA), a ϕ-node is placed at the confluence of a program control flow to represent the different choices of a variable. However, it does not contain the condition to determine which a incoming branch reaches a confluence node is chosen. By contrast, *gating functions* are defined with some extra parameters to represent the conditions for choosing. To construct a SDVG, we will employ the notation of gating function to capture the branching statements in computation of signals in synchronous data-flow programs and variables in the generated C code.

Gating functions were first introduced by Ballance et al. in [9] to represent the conditions that guard the paths to a ϕ-node. There are several types of gating functions as follows:

- The gated ϕ-function, which is an *if - then - else* representation. It captures the condition for choosing which branch of the confluence node. For instance, $x_3 = \phi(c, x_1, x_2)$ returns the value x_1 or x_2 depending on the value of c. If c is `true`, $x_3 = x_1$ and $x_3 = x_2$ if c is `false`.
- The μ function is used to capture the initial and loop-carried values at the header of a loop. For instance, $x_2 = \mu(i = 1, n, x_0, x_1)$ represent that x_2's initial value is x_0 when i is the first iteration and its subsequent value is x_2 from the previous iteration.
- The η function is placed at the loop exit. It selects the last value at the end of the loop. For instance, $x_2 = \eta(i > n, x_1)$ means that x_2 takes the last value of x computed by the loop.

5.1.2 Terms as trees and graphs

Let X and F be an infinite set of *variables* and a (finite or infinite) set of *function symbols* such that $X \cap F = \emptyset$. Each $f \in F$ has a number of arguments (or *arity*) greater or equal to 0, denoted by $f(x_1, x_2, ..., x_n)$, where n is the arity of f. Function symbols of arity 0 are called *constants*. Then the set of *terms* T is inductively defined by the following rules:

- Any variable is a term.
- Any expression $f(t_1, t_2, ..., t_n)$, where $f \in F$ and $t_1, ..., t_n \in T$, is a term.

Given a term t, the *substerms* of t is t and, if $t = f(t_1, ..., t_n)$, all subterms of $t_1, ..., t_n$. For example, let $X = \{x, y\}$ and $F = \{f, g\}$, then $T = \{x, y, f(x), g(y), f(x, y)...\}$. Note that we do not assume that function symbols have fixed arities.

Definition A directed graph over X and F is a tuple $\langle N, succ \rangle$ involving a (finite or infinite) set N of nodes which can be labelled by an element in $X \cup F$ and a function $succ : N \longrightarrow N^*$. In which the set of nodes $n_1, ..., n_k = succ(n)$ is the successors of n.

We write $succ(n)_i$ to denote the i^{th} element of $succ(n)$. The pair of nodes $e = (n, succ(n)_i)$ is called an edge, the set of all edges is denoted by E. When we draw the visual representation of graphs, a directed edge e will go from n to $succ(n)_i$, the ordering of the edges from n to $succ(n)$ is left-to-right corresponding to the ordering of the elements of $succ(n)$.

For example, let $X = \{a, b, c\}, F = \{+, *\}$ be the set of variables and the set of function symbols, respectively. The set of nodes and the function $succ$ is defined as follows:

$$N = \{+, *, a, b, c\}$$
$$succ(+) = (a, *), succ(*) = (b, c), succ(a) = (), succ(b) = (), succ(c) = ()$$

This defines a directed graph which is depicted on the left of Figure 6. Other directed graphs over F and X are depicted on the right side of the figure. Note that the in the third graph, the nodes with repeated label b are represented by a same node, and the label is shared in the graph.

Given a directed graph $G = \langle N, succ \rangle$, a path in G is a list of nodes $(n_0, n_1, ..., n_k)$ where $k \geq 0$ and n_{i+1} is a successor of n_i. This path is said to be from n_0 to n_k and k is the length of the path. A path of length greater than 0 from a node to itself is called a *cycle*, and the repeated node is called *cyclic* node. A graph which contains a cycle is *cyclic* graph, otherwise it is *acyclic*.

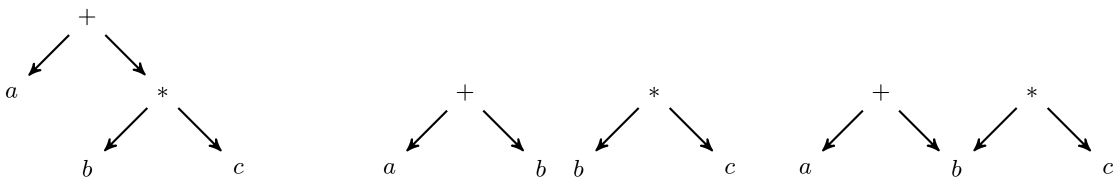


Figure 6: The directed graphs of $a + b * c$ and $a + b, b * c$

Definition A tuple $\langle N, succ, r\rangle$ is a term graph at the node $r \in N$ where $\langle N, succ\rangle$ is a directed graph, if every nodes of the term graph is reachable by a path from r. The node r is called the root of the graph.

In a term graph, a path from the root is said to be *rooted*. The term graph is *root-cyclic* if there is a cycle containing the root. Given a directed graph $G = \langle N, succ\rangle$, let n is a node in G. The subgraph of G rooted at n is the term graph $\langle N', succ', n\rangle$ where $N' = \{n' \in N |$ there is a path from n to $n'\}$ and $succ'$ is the restriction of $succ$ to N'. We write $G \setminus n$ to denote the fact that $G \setminus n$ is the subgraph rooted at n of G.

Consider, for example, the following directed graph which is depicted on the left of Figure 7. The subgraph rooted at the node labelled + is depicted as the seconde graphs in the figure. The third graph is a root-cyclic graph.

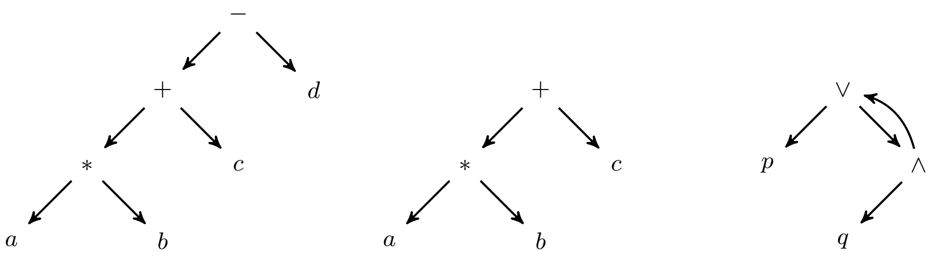


Figure 7: The subgraph rooted at node labelled + and a root-cyclic graph

We will use the similar notations as in the definition of term to introduce a linear notation for graphs. The notation is defined by the following context-free grammar:

$$\begin{array}{rcl} graph & := & node \mid node\ +\ graph \\ node & := & x \mid f(node, ..., node) \mid nid \mid nid : x \mid nid : f(node, ..., node) \end{array}$$

f and x are a symbol function in F and a variable in X. nid ranges over a set, disjoint from X and F, of *node identifiers*. Any node identifier nid in a graph which identifies a node must occur exactly once in the context $nid : x$ or $nid : f(node, ..., node$. The constants are simply represented by symbol functions with arities equal 0. This syntax is similar to the syntax for graphs in [10]. For instance, the three graphs of the examples in Figure 6 can be expressed in this syntax as follows:

$$+(a, *(b, c)), +(a, b) + *(b, c) \text{ and } +(a, nid_1 : b) + *(nid_1, c)$$

Note that multiple uses of the same node identifier mean that the multiple references to the same node.

Definition A tree is a term graph at the node r such that there is exactly one path from r to each node in the graph.

Based on the above definition, in Figure 7 the first and second graphs are trees and the third one is not. Thus it is obvious that a tree is always acyclic.

5.1.3 Homomorphisms of graphs and trees

Let $G_1 = \langle N_1, succ_1, r_1\rangle$ and $G_2 = \langle N_2, succ_2, r_2\rangle$ be two graphs, the definition of *homomorphism* is given as follows:

Definition A homomorphism from G_1 to G_2 is a map $f : N_1 \longrightarrow N_2$ such that for all $n \in N_1$,

- $f(n)$ and n have the same label
- $succ_2(f(n)) = f(succ_1(n))$

where $f(n_1, ..., n_k) = (f(n_1), ..., f(n_k))$. This definition states that homomorphisms preserve node labels, successors and their order. We write $G_1 \longrightarrow G_2$ to denote the fact that there is a homomorphism from G_1 to G_2. Figure 8 shows an example of a homomorphism.

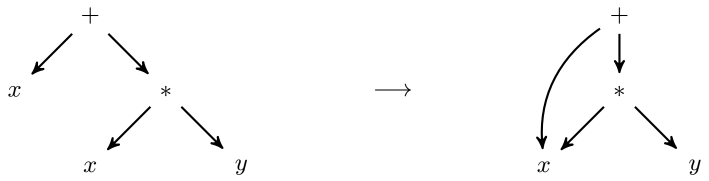


Figure 8: An example of homomorphism

Definition Let $G_1 = \langle N_1, succ_1, r_1\rangle$ and $G_2 = \langle N_2, succ_2, r_2\rangle$ be two graphs.

- A homomorphism f from G_1 to G_2 is *rooted* if $f(r_1)$ and r_2 have same label.
- An *isomorphism* is a homomorphism which is inverse. We denote an isomorphism from G_1 to G_2 by $G_1 \sim G_2$.
- When there is a rooted isomorphism from G_1 to G_2, we say that they are *equivalent*, denoted by $G_1 \approx G_2$.

Proposition 5.1 *For any graphs G_1 and G_2, we have $G_1 \approx G_2$ implies $G_1 \sim G_2$. Every rooted homomorphism from one tree to another is an isomorphism.*

5.1.4 Synchronous data-flow value-graph

Let X be the set of all variables which are used to denote the signals, clocks and variables in a Signal program and its generated C program. In our consideration, the functions which apply on signal values in the primitive *stepwise functions* are usual logic operators (`not`, `and`, `or`), numerical comparison functions (`<`, `>`, `=`, `<=`, `>=`, `/=`) and numerical operators (`+`, `-`, `*`, `/`). A constant is defined as a function symbol of arity 0. Thus in this chapter, we consider the set of function symbols which consists of the above functions and the gated ϕ-function, denoted by F.

As it is illustrated in Section 4, the computation of signals in a Signal program and variables in the corresponding generated C code can be represented as a directed graph, in which a node can have multiple parents, and identical subgraphs are reused. That makes the maximal sharing among graph nodes. We will consider the definition and examine some basic properties of SDVG. Formally, a SDVG is defined as follows:

Definition A SDVG associated with a Signal program and its generated C code is a directed graph $G = \langle N, E, I, O, m_N \rangle$, where:

- N is a finite set of nodes. Each node is labelled by an element in $X \cup F$. A node represents a clock, a signal, a variable, an operator or a gated ϕ-node function. And the subgraph rooted at a node is used to describe the computation of a value of the corresponding element labelled at the node.
- $E \subseteq N \times N$ is the set of edges. It describes the computation relation such that a operand and operator relation between the nodes.
- $I \subseteq N$ is the set of input nodes. They are the input signals and their corresponding variables in the generated C code.
- $O \subseteq N$ is the set of output nodes. They are the output signals and their corresponding variables in the generated C code.
- $m_N : N \longrightarrow \mathcal{P}(V)$ is a mapping labeling each node with a finite set of clocks, signals, and variables. It defines the set of clocks, signals or variables in the Signal program and the generated C code such that they are equivalent to the computation value which is pointed by the node.

In the rest of this chapter, we denote the fact that there exists edge between two nodes x and y by $x \longrightarrow y$. A *path* from x to y is any set of nodes $s = \{x_0, x_1, ..., x_k\}$ such that $\forall i = 0, .., k-1, x_i \longrightarrow x_{i+1}$. An edge is a special case when $k = 1$.

5.2 SDVG of Signal programs

Let P be a Signal program, we write $X = \{x_1, ..., x_n\}$ to denote the set of all signals in program P which consists of input, output, state (corresponding to delay operator) and local signals, denoted by I, O, S and L, respectively. For each $x_i \in X$, we use $\mathbb{D}_{x_i}$ to denote its domain of values, and $\mathbb{D}^{\#}_{x_i} = \mathbb{D}_{x_i} \cup \{\#\}$ to denote its domain of values with the absent value, where $\# \notin \mathbb{D}_{x_i}$. Then, the domain of values of X with absent value is defined as follows:

$$\mathbb{D}^{\#}_{X} = \bigcup_{i=1}^{n} \mathbb{D}_{x_i} \cup \{\#\}$$

With each signal x_i (boolean or non-boolean type), we attach a boolean variable $\widehat{x_i}$ to encode its clock at a given instant (`true`: x_i is present, `false`: x_i is absent), and $\overline{x_i}$ with the same type as x_i to encode its value. Formally, the abstract values to represent the abstract clock and value of a signal can be represented by a gated ϕ-function, $x_i = \phi(\widehat{x_i}, \overline{x_i}, \#)$, where $\widehat{x_i}$ and $\overline{x_i}$ are computed using the following functions:

$$\begin{array}{ll} \hat{} : X \longrightarrow \mathbb{B} & \text{associates a signal with a boolean value,} \\ \bar{} : X \longrightarrow \mathbb{D}_X & \text{associates a signal with a value of same type as the signal.} \end{array}$$

Assume that we have the computation of signals in processes P_1, P_2 is represented as shared value-graphs G_1 and G_2, respectively. Then the value-graph G of the synchronous combination

$P_1|P_2$ can be defined as $G = \langle V, E, I, O, m_N \rangle$ in which for any node x, we replace it by the subgraph that represents the definition of x, in G_1, G_2. And in G_1 and G_2, every identical subgraph is reused, in other word, we maximize sharing among graph nodes in G_1 and G_2. Thus, the shared value-graph of a Signal program P can be constructed as a combination of the sub-value-graphs of its equations as above.

To demonstrate the above combination rule, we consider a simple Signal program P as follows:

Listing 5 : Simple Program in Signal

```
process P=
(? integer x;
 ! integer y)
(| y := x * x1
 | x1 := x + 1
|)
where integer x1
end;
```

Suppose that we have the subgraphs that represent the equations $y := x * x1$ and $x1 := x + 1$ as depicted in Figure 9 (here, we omit to represent the abstract clocks of x and $x1$ at the node $\hat{y}$, $\overline{x1}$ at the node $+$). We replace the node $x1$ by the subgraph which defines it while reusing the identical node x. As a result, the shared synchronous data-flow value-graph of P is depicted in Figure 10.

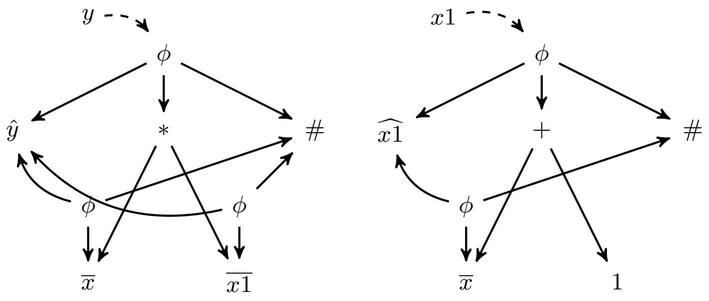


Figure 9: The subgraphs of $y := x * x1$ and $x1 := x + 1$

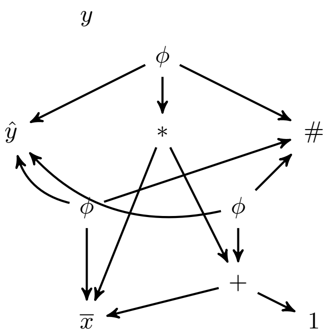


Figure 10: The SDVG Graph of P

The Signal program is built through the set of primitive operators. Therefore, it is obvious that to construct SDVGs of Signal programs, we will construct a subgraph for each primitive operator. In the following, we present the value-graph corresponding to each Signal primitive operator.

5.2.1 Stepwise functions

In our consideration, the functions which apply on signal values in the primitive *stepwise functions* are usual logic operators (`not`, `and`, `or`), numerical comparison functions (`<`, `>`, `=`, `<=`, `>=`, `/=`) and numerical operators (`+`, `-`, `*`, `/`).

Consider the operator $y := f(x_1, ..., x_n)$, it indicates that if all signal from x_1 to x_n are defined, then the output signal y is defined by the result of the function f on the values of $x_1, ..., x_n$. Otherwise, it is assigned no value. Thus, the computation of y can be represented by the following gated ϕ-function:

$$y = \phi(\hat{y}, f(\overline{x_1}, \overline{x_2}, ..., \overline{x_n}), \#)$$

where

$$\hat{y} \Leftrightarrow \widehat{x_1} \Leftrightarrow \widehat{x_2} \Leftrightarrow ... \Leftrightarrow \widehat{x_n}$$

The synchronous data-flow value-graph of the *stepwise functions* is depicted in Figure 11. Note that in the graph, $\{\widehat{x_1}, ..., \widehat{x_n}\}\ \ \hat{y}$ means that the clocks $\widehat{x_1}, ..., \widehat{x_n}$ and $\hat{y}$ are equivalent meaning that they point to the same node in the graph.

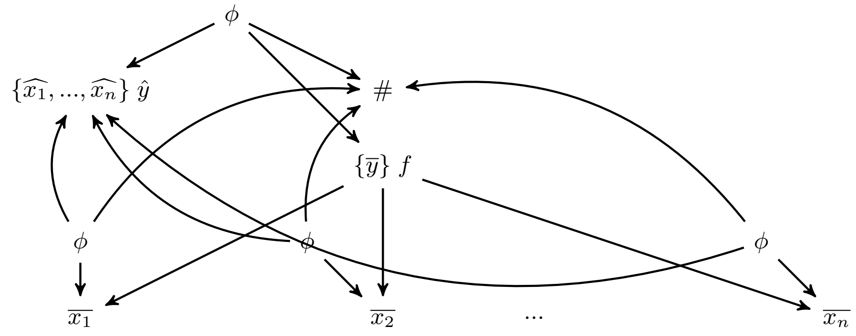


Figure 11: The graph of $y := f(x_1, ..., x_n)$

For instance, consider the following Signal equation:

$$y := (x >= 1) \text{ and } c$$

It can be represented by the gated ϕ-function, $y = \phi(\hat{y}, \overline{x_1} \wedge \overline{c}, \#)$, where we replace $(x >= 1)$ by the fresh signal x_1. Thus the graphic representation is depicted in Figure 12.

5.2.2 Delay

Consider the basic process which corresponds to the *delay* operator $y := x\$1 \text{ init } a$. The output signal y is defined by the last value of the signal x when the signal x is present. Otherwise, it is assigned no value. Thus, the computation of y can be represented by the following nodes:

$$y = \phi(\hat{y}, \overline{m.x}, \#) \text{ and } \overline{m.x_0} = a$$

where

$$\hat{y} \Leftrightarrow \hat{x}$$

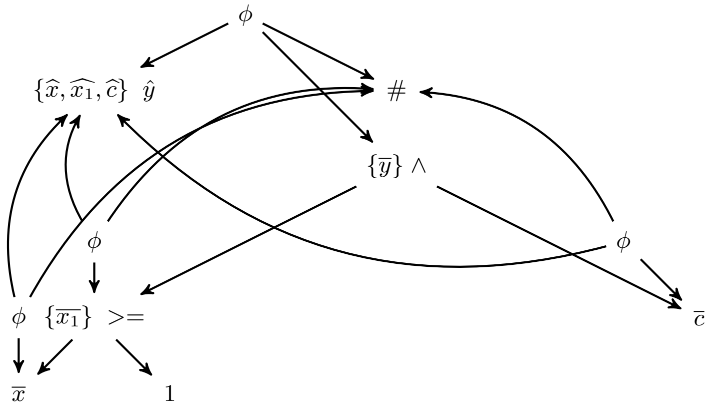

ϕ
$\{\widehat{x}, \widehat{x_1}, \widehat{c}\}$ $\hat{y}$
$\#$
$\{\overline{y}\}$ $\wedge$
ϕ
ϕ
ϕ $\{\overline{x_1}\}$ $>=$
$\overline{c}$
$\overline{x}$
1

Figure 12: The graph of $y := (x >= 1)$ and c

$\overline{m.x}$ and $\overline{m.x_0}$ are the last value of x and the initialized value of y. The synchronous data-flow value-graph of the *delay* operator is depicted in Figure 13. Note that in the graph, $\{\hat{x}\}$ $\hat{y}$ means that the clocks $\hat{x}$ and $\hat{y}$ are equivalent meaning that they point to the same node in the graph.

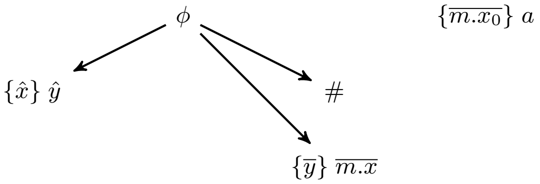

ϕ
$\{\overline{m.x_0}\}$ a
$\{\hat{x}\}$ $\hat{y}$
$\#$
$\{\overline{y}\}$ $\overline{m.x}$

Figure 13: The graph of $y := x\$1$ init a

For instance, consider the following Signal equation:

$$y := (x\$1 \text{ init } 1) + z$$

It can be represented by the gated ϕ-function, $y = \phi(\hat{y}, \overline{m.x} + \overline{z}, \#)$ and the node $\overline{m.x_0} = 1$. Thus the graphic representation is depicted in Figure 14.

5.2.3 Merge

Consider the basic process which corresponds to the *merge* operator $y := x$ default z. If the signal x is defined then the signal y is defined and holds the value of x. The signal y is assigned the value of z when the signal x is not defined and the signal z is defined. When both x and z are not defined, y holds no value. The computation of y can be represented by the following node:

$$y = \phi(\hat{y}, \phi(\hat{x}, \overline{x}, \overline{z}), \#)$$

where

$$\hat{y} \Leftrightarrow (\hat{x} \vee \hat{z})$$

The representation uses a nested ϕ-function which indicates that when $\hat{y}$ is `true`, y is defined by the gated ϕ-function $\phi(\hat{x}, \overline{x}, \overline{z})$. Otherwise, it holds no value. The synchronous data-flow

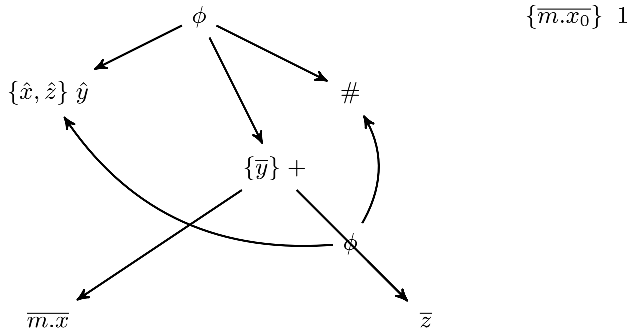


Figure 14: The graph of $y := (x\$1 \text{ init } 1) + z$

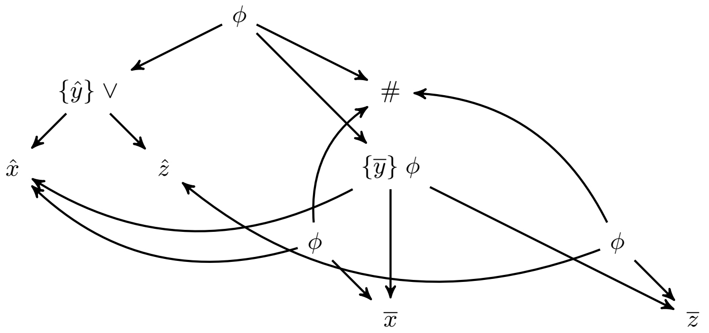


Figure 15: The graph of $y := x$ default z

value-graph of the *sampling* operator is depicted in Figure 15. Note that in the graph, the clock $\hat{y}$ is represented by the subgraph of $\hat{x} \vee \hat{z}$.

For instance, consider the following Signal equation:

$$y := x \text{ default } (z + 1)$$

It can be represented by the nested ϕ-function, $y = \phi(\hat{y}, \phi(\hat{x}, \overline{x}, \overline{z_1}), \#)$, where we replace $(z + 1)$ by the fresh signal z_1. Thus the graphic representation is depicted in Figure 16.

5.2.4 Sampling

Consider the basic process which corresponds to the *sampling* operator $y := x$ when b. If the signal x, b are defined and b holds the value `true`, then the signal y is defined and holds the value of x. Otherwise, y holds no value. The computation of y can be represented by the following node:

$$y = \phi(\hat{y}, \overline{x}, \#)$$

where

$$\hat{y} \Leftrightarrow (\hat{x} \wedge \hat{b} \wedge \overline{b})$$

The synchronous data-flow value-graph of the *sampling* operator is depicted in Figure 17. Note that in the graph, the clock $\hat{y}$ points to the root of the subgraph of $(\hat{x} \wedge \hat{b} \wedge \overline{b})$.

For instance, consider the following Signal equation:

$$y := x \text{ when } (z >= 1)$$

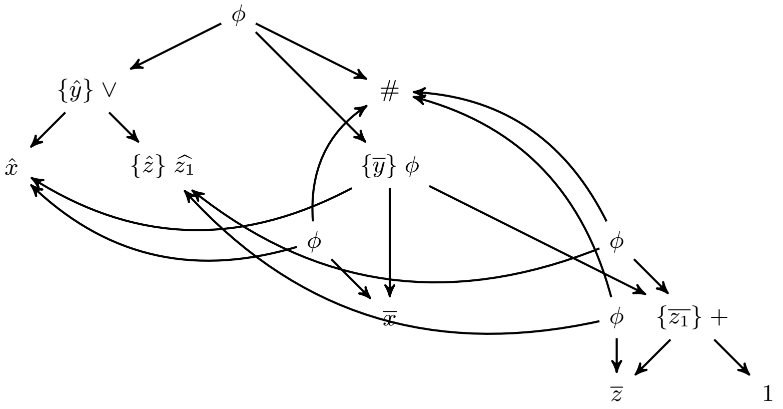


Figure 16: The graph of $y := x$ default $(z + 1)$

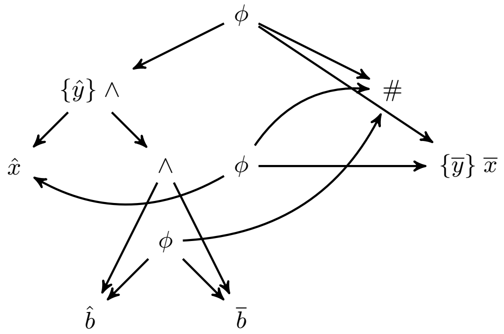


Figure 17: The graph of $y := x$ when b

It can be represented by the gated ϕ-function, $y = \phi(\hat{y}, \overline{x}, \#)$, where $\hat{y} \Leftrightarrow (\hat{x} \wedge \widehat{z_1} \wedge \overline{z_1})$ and we replace $(z >= 1)$ by the fresh signal z_1. Thus the graphic representation is depicted in Figure 18.

5.2.5 Restriction

The shared synchronous data-flow value-graph of *restriction* process P_1 where x is the same as the graph of P_1.

5.2.6 Clock relations

Given the above graph representations of the primitive operators, we can obtain the shared value-graphs for derived operators on clocks as depicted in Figure 19 and Figure 20. Here, z is a signal of type *event*. Its computation can be represented by the following gated ϕ-function:

$$z = \phi(\hat{z}, \texttt{true}, \#)$$

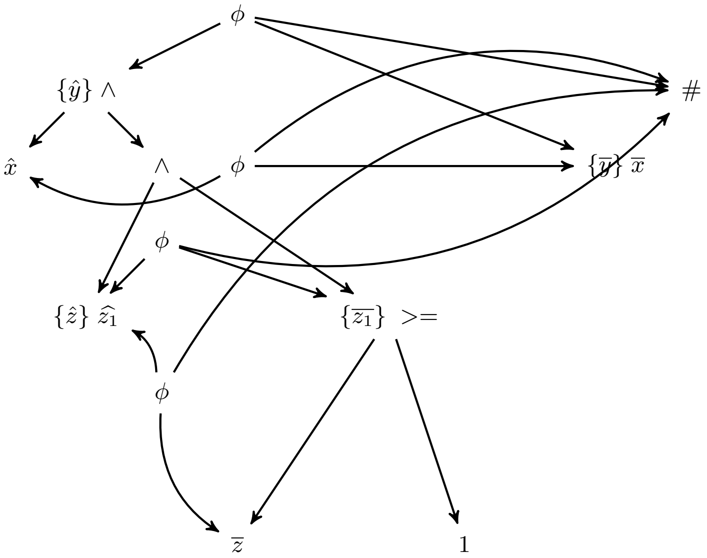
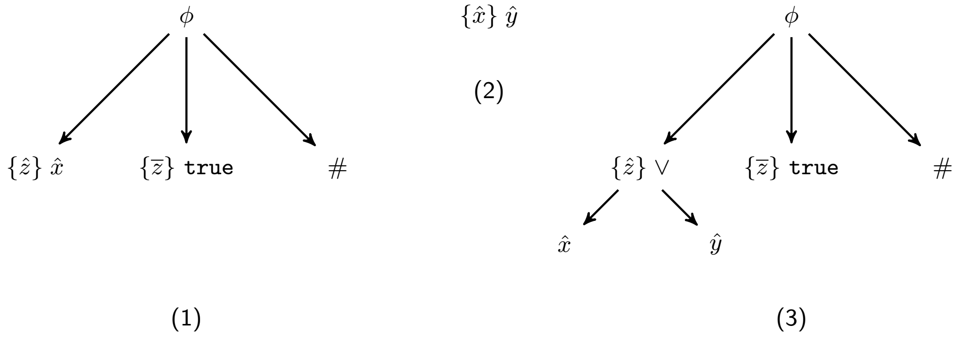


Figure 18: The graph of $y := x$ when $(z >= 1)$

The clock relations between signals are given as follows:

$$\begin{array}{rl}
z :=\hat{}x : & \hat{z} \Leftrightarrow \hat{x} \\
x\hat{}=\ y : & \hat{x} \Leftrightarrow \hat{y} \\
z := x\hat{}+\ y : & \hat{z} \Leftrightarrow (\hat{x} \vee \hat{y}) \\
z := x\hat{}*\ y : & \hat{z} \Leftrightarrow (\hat{x} \wedge \hat{y}) \\
z := x\hat{}-\ y : & \hat{z} \Leftrightarrow (\hat{x} \wedge \neg\hat{y}) \\
z :=\ \text{when}\ b : & \hat{z} \Leftrightarrow (\hat{b} \wedge \bar{b})
\end{array}$$



(1)

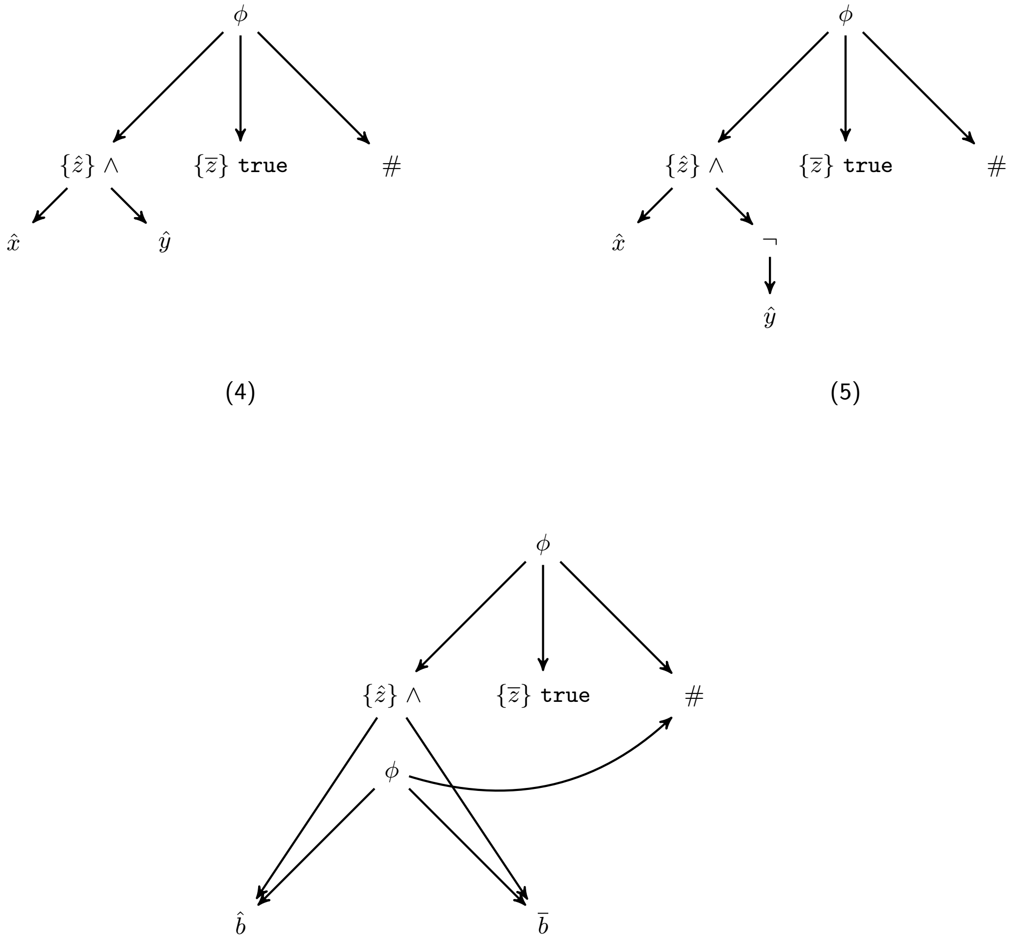
(2)

(3)

Figure 19: The graphs of (1) $z :=\hat{}x$, (2) $x\hat{}=\ y$ and (3) $z := x\hat{}+\ y$

5.3 SDVG of generated C code

For constructing the shared value-graph, the generated C program is also translated into the sub value-graph along with the sub value-graph of the Signal program. Let A be a Signal program



(4)

(5)

(6)

Figure 20: The graphs of (4) $z := x\,\hat{}*\ y$, (5) $z := x\,\hat{}-\ y$ and (6) $z :=\ when\ b$

and C be its generated C code, we write $X_A = \{x_1, ..., x_n\}$ to denote the set of all signals in program A, and $X_C = \{x_1^c, ..., x_m^c\}$ to denote the set of all variables in program C. We added "c" as superscript for the program variables, to distinguish them from the signals in the Signal program.

As description in the principle of code generation in Signal compiler , the generated C program of the Signal program A consists the following files:

- *A_ main.c* consists of the implementation of the *main function.* This function opens the IO communication channels, and calls the *initialization function.* Then it calls the *step function* repeatedly in an infinite loop to interact with the environment.

- *A_ body.c* consists of the implementation of the *initialization function* and the *step function.* The initialization function, which is called once to provide initial values to the program variables. The step function containing step initialization and finalization functions, is responsible for the calculation the outputs to interact with the environment. This function is called repeatedly in an infinite loop, is the essential part of the concrete code.

- *A_io.c* consists of the implementation of the *IO communication functions*. The IO functions, which are called to setup communication channels with the environment.

The scheduling and the computations is done inside the step function of the generated C program. Therefore, it is natural to construct the sub value-graph of this function in order to prove that the values of its variables and their corresponding signals are the same. To construct the value-graph of the *step function*, we need to consider the following considerations.

An original signal named x has a boolean variable named C_x in the *step function*. Then the computation of x is implemented by conditional *if - then -else* statements as follows:

```
if (C_x)
{
   computation(x);
}
```

If x is an input signal then its computation is the reading operation which gets the value of x from the environment. In case x is a output signal, after computing its value, it will be written to the IO communication channel with the environment. Note that the C programs use persistent variables (e.g. variables which are always have some values) to implement the input Signal programs which use volatile variables. As a result, there is a difference in the types of a signal in Signal program and the corresponding variable in C program. When a signal has absent value, #, at a given instant, the corresponding C variable is remained and can never have absent value. For instance, in the *step function* `WHENOP_step()`, we have an observation that the value of variable N is remained when the boolean variable C_N has the value `false` as the following code segment:

```
if (C_N)
{
   N = 4 * FB1;
   w_WHENOP_N(N);
}
```

This consideration implies that we have to detect whenever a variable in the C program whose value is never updated. It will be assigned the absent value, #. Thus, the computation of such variables x^c, recall that we use the superscript for the C program variables, to distinguish them from the signals in the Signal program, in the *step function* can fully be represented by a gated ϕ-function as follows:

$$x^c = \phi(C\_x^c, \overline{x^c}, \#)$$

where $\overline{x^c}$ denotes the newly updated value of the variable. For example, we consider the generated code of basic process corresponding to the primitive operator *merge* in the following listing.

Listing 6 : SDVGMerge in Signal

```
process SDVGMerge=
(? integer x, y; boolean cx, cy;
 ! integer z;
)
(| z := x default y
 | x ^= when cx
 | y ^= when cy
 |)
;
```

The generated C code of the *step function* are given in Listing 7. The computations of variable z in this function can be represented by the following gated ϕ-function:

$$z^c = \phi(C\_z^c, \phi(cx\_50^c, \overline{x^c}, \overline{y^c}), \#)$$

The shared value-graph of `SDVGMerge_step()` is depicted in Figure 21. In this graph, we replace the node $\phi(C\_cx^c, \phi(\overline{cx^c}, \overline{x^c}, \#), \#)$ by $\phi(C\_cx^c \wedge \overline{cx^c}, \overline{x^c}, \#)$. We do the same for the node $\phi(C\_cy^c, \phi(\overline{cy^c}, \overline{y^c}, \#), \#)$.

Listing 7 : Generated C code of SDVGMerge

```
EXTERN logical SDVGMerge_step()
{
   if (!r_SDVGMerge_C_cx(&C_cx)) return FALSE;
   if (!r_SDVGMerge_C_cy(&C_cy)) return FALSE;
   if (C_cx)
   {
      if (!r_SDVGMerge_cx(&cx)) return FALSE;
      if (cx)
      {
         if (!r_SDVGMerge_x(&x)) return FALSE;
      }
   }
   cx_50 = (C_cx ? cx : FALSE);
   if (C_cy)
   {
      if (!r_SDVGMerge_cy(&cy)) return FALSE;
      if (cy)
      {
         if (!r_SDVGMerge_y(&y)) return FALSE;
      }
   }
   cy_56 = (C_cy ? cy : FALSE);
   C_z = cx_50 || cy_56;
   if (C_z)
   {
      if (cx_50) z = x; else z = y;
      w_SDVGMerge_z(z);
   }
   SDVGMerge_step_finalize();
   return TRUE;
}
```

In the generated C program, the computation of the variable whose clock is the master clock, which ticks every time the *step function* is called, is implemented without the conditional *if* - *then* -*else* statement and it is always updated when the *step function* is invoked. The computation of such variables can be represented by a node in the shared value-graph as follows:

$$\{\overline{x^c}\}\ x^c$$

This representation is used for any variable in the generated C code such that this variable is always updated when the *step function* is called. For instance, consider the following code segment which computes the values of the output signal N in program `DEC` and the variable C_z in the `SDVGMerge_step()` function in the following listing:

```
if (C_FB) N = FB;
else N = N - 1;
w_DEC_N(N);

C_z = cx_50 || cy_56;
```

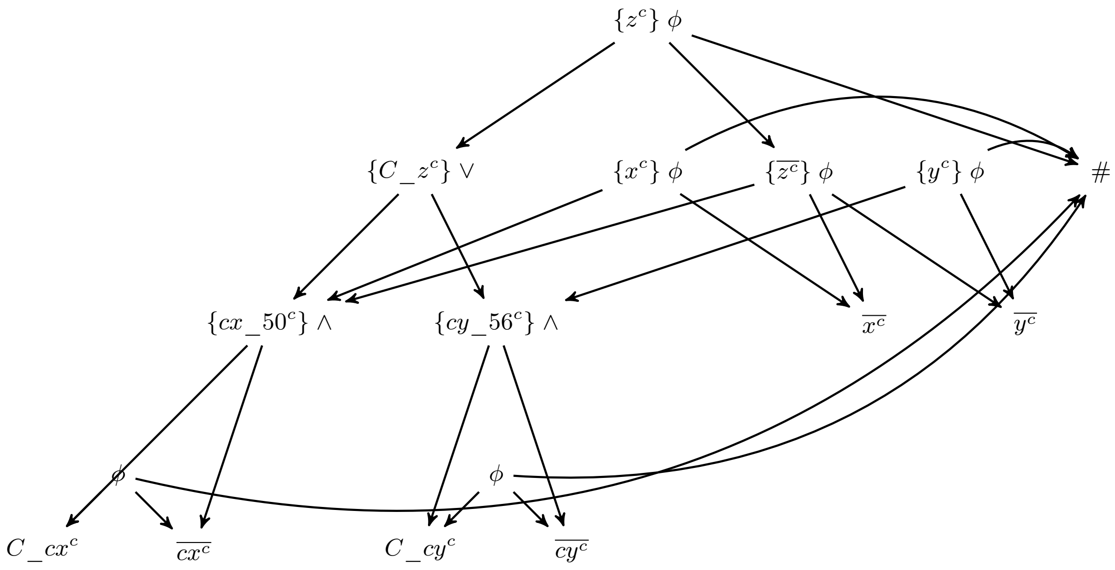


Figure 21: The graph of SDVGMerge step function

The computation of N and C_z can be represented by the sub value-graphs in Figure 22, where $m.N^c$ denotes the previous value of the variable N.

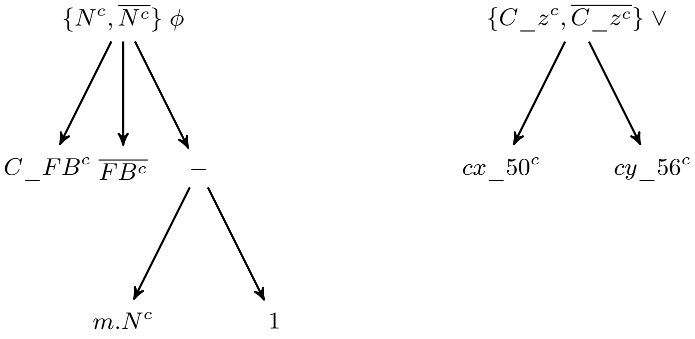

$\{N^c, \overline{N^c}\}\ \phi$
C_FB^c
$\overline{FB^c}$
$-$
$m.N^c$
1
$\{C\_z^c, \overline{C\_z^c}\}\ \vee$
cx_50^c
cy_56^c

Figure 22: The graphs of N and C_z's computations

Consider the following code segment. The observation is that the variable x is involved in the computation of the variable y before the updating of x.

```
if (C_y)
{
   y = x + 1;
}
...
if (C_x)
{
   x = ...
}
...
```

In this situation, we refer to the value of x as the previous value, denoted by $m.x^c$. It can be happened when the *delay* operator is applied on the signal x in the Signal program. The computation of y is represented by the following gated ϕ-function:

$$y^c = \phi(C\_y^c, m.x^c + 1, \#)$$

6 SDVG translation validation

In order to apply the translation validation to the code generation phase of the Signal compiler, we represent the computations of signals in the intermediate representation form written in Signal language and the computations of corresponding variables in the sequential generated code, written in C language by means of synchronous data-flow value-graphs.

Then we introduce the set of rewrite rules to transform the shared value-graph resulted in the previous step. This procedure is called *normalizing*. At the end of the normalizing procedure, we check that for any signal x and its corresponding variable x^c in the generated C code, if it is an output signal then x and x^c point to the same node in the shared value-graph. That means they are represented by the same subgraph. In other words, they have the same value. We also provide a method to implement the representation of synchronous data-flow value-graph and adapt the normalizing procedure with any feature optimization of the compiler.

We introduce the graph rewriting techniques in Section 6.1. Section 6.2 provides the set of rewrite rules which is used to perform the normalizing procedure on the shared value-graph. In Section 6.3, we consider a method to implement the data structure of synchronous data-flow value-graph and the normalizing procedure.

6.1 An introduction to graph rewriting

Graphs provide a simple and powerful approach to a variety of problems of software engineering, and system modeling in particular. These graphs are mostly static descriptions of system states. Adding dynamics requires some ways to express state changes and thus graph transformations are involved, either explicitly or behind the scenes. In fact, the theory of graph transformation has found many applications in the implementing functional languages based on term rewriting [11], in modeling of concurrent systems [12] and in other areas such as software engineering, hardware designs and visual languages.

First we recall the basic concepts of term rewriting, and we explain informally how the term rewriting work in a simple example. We then define our notion of graph rewriting. For a comprehensive introduction of term rewriting and graph rewriting, the interested reader may consult the textbook [8]. Consider, for example, the following rewrite rules for defining for natural number multiplication:

$$\begin{array}{rcl} x * 0 & \longrightarrow & 0 \\ x * (y+1) & \longrightarrow & (x * y) + x \end{array}$$

We will apply the second rewrite rule to the expression $t*(u+1)$, where t and u are subexpressions, which is represented by the graph on the left of Figure 23. When applying the above rewrite rule, the subexpression t is evaluated two times if it has not been yet evaluated as depicted on the right of the figure. To solve this problem, an easy solution is, instead of copying the subexpression t, to create two references to the existing subexpression t, meaning that the repeated subterms are shared. Then the result of applying the above rewrite rule looks as the graphs in Figure 24.

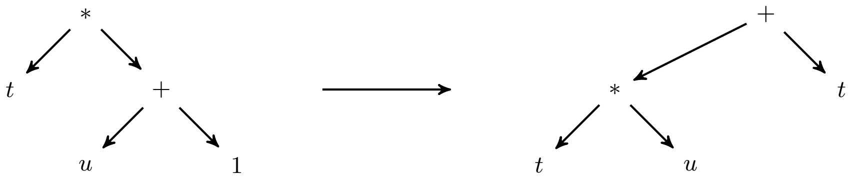


Figure 23: The transformation of the graph of $t * (u + 1)$

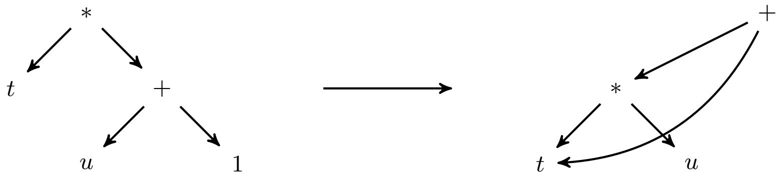


Figure 24: The transformation of graph of $t * (u + 1)$ with sharing of repeated subterms

Let X and F be the set of variables and function symbols such that $X \cap F = \emptyset$. We denote the set of all terms over X and F by $T_{X,F}$. A mapping $\sigma : T_{X,F} \longrightarrow T_{X,F}$ is called a *substitution* if $\sigma(c) = c$ and $\sigma(f(t_1, ..., t_n) = t(\sigma(t_1), ..., \sigma(t_n))$ for every constant c and term $f(t_1, ..., t_n)$.

Definition A term rewrite rule over $T_{X,F}$ is a pair of terms (t_l, t_r), written as $t_l \to t_r$, such that:

- t_l is not a variable, and
- $\forall x \in X.(x \in t_r \Rightarrow x \in t_l)$.

We call t_l and t_r are, respectively, the left and right-hand sides of the rewrite rule. A rewrite rule is *left-linear* (resp. *right-linear*) if no variable occurs more than once in its left-hand side (resp. right-hand side).

A *term rewriting system* is a tuple $\langle F, R\rangle$, in which F is a set of function symbols and R is the set of term rewrite rules over $T_{X,F}$. A term rewriting system is said to be left-linear (resp. right-linear) if all its rules are.

Given two term t_1 and t_2, we say that there exists a rewrite relation on $T_{X,F}$ for (t_1, t_2) induced by $\langle F, R\rangle$, denoted by $t_1 \rightsquigarrow t_2$, if it satisfies:

- There exists a rule $(l, r) \in R$ and a substitution σ such that $\sigma(l)$ is a subterm of t_1.
- The term t_2 is obtained from t_1 by replacing the occurrence of $\sigma(l)$ by $\sigma(r)$.

We now define the application of term rewriting rules to graph, in other words, we make the graph representation of term rewriting rules. Let G be a graph and n and n' be nodes of G, the triple (G, n, n') is called a *graph rewrite rule* and n, n' are the *left root* and the *right root* of the rule.

A pair $\Delta = (r, f)$ is a *redex* in a graph G_0 if r is a graph rewrite rule (G, n, n') and f is a homonorphism from $G|_n$ to G_0. The homonorphism is called an *occurrence* of the rule r. We first begin with some example to illustrate the definition of graph rewriting technique by showing how the translation of term rewrite rules to graph rewrite rule work.

Let $t_l \to t_r$ be a term rewrite rule, we will construct the corresponding graph rewrite rule (G, n, n'). The construction works as follows. First we take the graph representing of both left and right hand sides of the term rewrite rule to form the union of these graphs, sharing those nodes which represent the same structures in t_l and t_r. This resulting shared graph is G. Then we take the roots of t_l and t_r to be n and n', respectively. For example, we illustrate the construction of the graph rewrite rule for the following term rewrite rule:

$$\phi(c, x, false) \rightarrow c \wedge x$$

We first make the graph representing of $\phi(c, x, false)$ and $c \wedge x$ on the left of the Figure 25. Then the union graph sharing the same nodes (c and x) is depicted on the right. Finally, the left root n and right root n' are the nodes labelled ϕ and $\wedge$. Thus the graph rule is represented as follows:

$$(n : \phi(n_c : c, n_x : x, false) + n' : \wedge(n_c, n_x), n, n')$$

Let $((G, n, n'), f : G|_n \rightarrow G_0)$ be a redex in the graph G_0. We now present a formal definition of the general construction of the graph rewriting from a graph rewrite rule (G, n, n'). The construction consists of three phases: *build*, *redirection* and *garbage collection*, which are defined as follows.

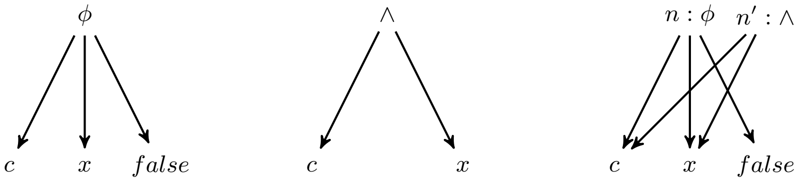


Figure 25: The graph rule of the term rule $\phi(c, x, false) \rightarrow c \wedge x$

The build phase The resulting graph $G_1 = \langle N_1, succ_1, r_1\rangle$ in this phase, denoted by $G_1 = G_0 +_f (G, n, n')$, is constructed as follows.

- $N_1 = N_{G_0} \uplus (N_{G|_{n'}} \setminus N_{G|_n})$, where $\uplus$ denotes the disjoint union of two sets. For any node $m \in N_{G_1}$, it has the same label as in G_0 if $m \in N_{G_0}$. Otherwise, it has the same label as in G.
- $r_1 = r_{G_0}$.
- for every node $m_i = succ_{G_1}(m)_i$,

$$m_i = \begin{cases} succ_{G_0}(m)_i, & \text{if } m \in N_{G_0} \\ succ_G(m)_i, & \text{if } m, succ_G(m)_i \in N_{G|_{n'}} \setminus N_{G|_n} \\ f(succ_G(m)_i), & \text{if } m \in N_{G|_{n'}} \setminus N_{G|_n}, succ_G(m)_i \in N_{G|_n} \end{cases}$$

The redirection phase In this step all references to the node $f(n)$ are replaced by the references to the node n', the resulting graph is denoted by $G_2 = \langle N_2, succ_2, r_2\rangle = G_1[f(n) := n']$. This replacement is defined by a substitution operation in the term graph G_1 with two nodes $f(n)$ and n' as follows.

- for every node $c \in N_1$ and $c \in N_2$, they have the same label.
- for every node $c \in N_1$, $succ_2(c)_i = n'$ if $succ_1(c)_i = f(n)$, otherwise $succ_2(c)_i = succ_1(c)_i$.
- $r_2 = n'$ if $r_1 = f(n)$, otherwise $r_2 = r_1$.

The garbage collection phase In this last step, we define $G_3 = G_2|_{r_2}$ which is a part of the graph G_2 accessible from its root. We denote the graph G_3 by $GC(G_2)$.
Given a redex $\Delta = (r, f)$ and a graph G_0, the construction of the graph resulting from the reducing the redex Δ in the graph G_0, denoted by $RED(\Delta, G_0)$, is defined as:

$$RED(((G, n, n')f), G_0) = GC((G_0 +_f (G, n, n'))[f(n) := n'])$$

To illustrate the above construction of graph rewriting, we consider the following example: $(G, n, n') = (n : \phi(n_c : c, n_x : x, false) + n' : \wedge(n_c, n_x), n, n')$ is the graph rule above, $G_0 = \vee(\phi(c, x, false), y)$ and a homomorphism f from $G|_n$ to G_0 as depicted in Figure 26.

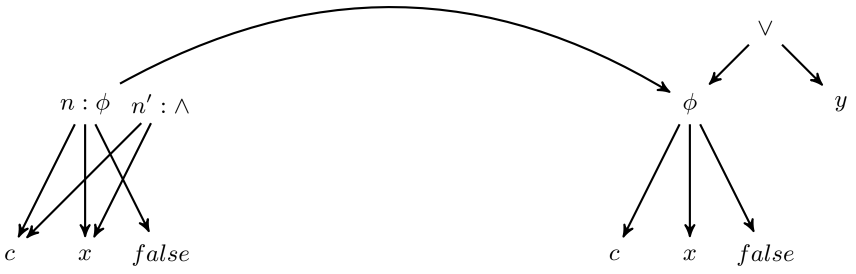


Figure 26: An example of graph rewriting

We first copy the part of $G|_n'$ which is not contained in $G|_n$ to G_0, with node labels, successors, and root defined in the build phase. We obtain the resulting graph G_1. Then all edges of G_1 pointing to $f(n)$ are replaced by edges pointing to the copy of n', giving the graph G_2 as depicted in Figure 27. The root of G_2 is the root of G_1 if that node not equal to $f(n)$. Otherwise, the root of G_2 is the copy of n' as described in the redirection phase. From the graph G_2, we remove all nodes which are not accessible from the the root, giving the result of the rewrite, the graph G_3 as depicted in Figure 28.

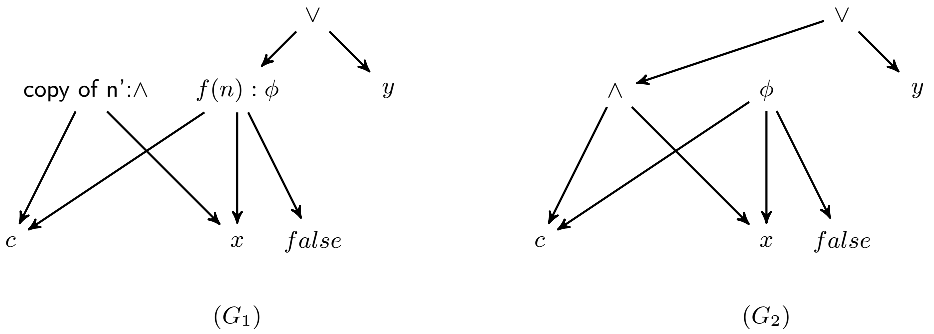
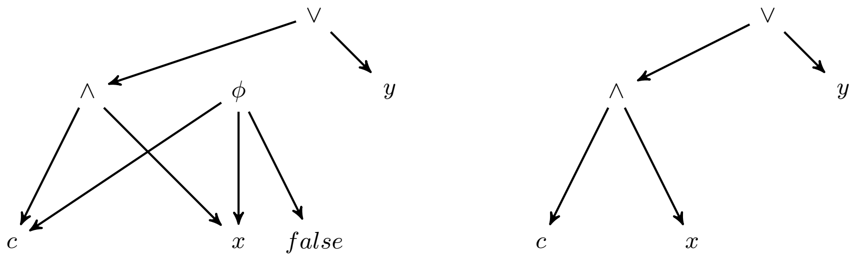


Figure 27: Graph rewriting: Build and redirection phases

6.2 Normalizing

Once a shared value-graph is constructed for the Signal program and its generated C program, if the values of an output signal and its corresponding variable in the C program are not already



(G_2) (G_3)

Figure 28: Graph rewriting: Garbage collection phases

equivalent (they do not point the same node in the shared value-graph), we start to normalize the graph. Given a set of term rewrite rules, the normalizing process works as described in Listing 8.

Listing 8 : Normalizing value-graph

```
Input: $G$: A shared value-graph.
       $R$: The set of rewrite rules.
       $S$: The sharing among graph nodes.
Output: The normalized graph

while ($\exists s \in S$ or $\exists r \in R$ can be applied on $G$) do
{
   while ($\exists r \in R$ can be applied on $G$)
   {
   for ($n \in G$)
      if ($r$ can be applied on $n$)
         apply the rewrite rule to $n$
   }
   maximize sharing
}
return $G$
```

The normalizing algorithm indicates that we apply the rewrite rules to each graph node individually. When there is no more rules can be applied to the resulting graph, we maximize the shared nodes. When there exists no more sharing or rules can be applied, the process terminates.

We classify our set of rewrite rules into three basic types: *general simplification rules*, *optimization-specific rules* and *synchronous rules*. In the following, we will present the rewrite rules of these types, and we assume that all nodes in our shared value-graph are typed. Note that we only write the rewrite rules in forms of term rewrite rules, $t_l \rightarrow t_r$.

6.2.1 General simplification rules

The general simplification rules contain the rules which are related to the general rules of inferences of operators, denoted by the corresponding function symbols in F. In our consideration, the operators used in the primitive *stepwise functions* and in the generated C code are are usual logic operators (`not`, `and`, `or`), numerical comparison functions (`<`, `>`, `=`, `<=`, `>=`, `/=`), and numerical operators (`+`, `-`, `*`, `/`). When applying these rules, we will replace a subgraph rooted at a node by a smaller subgraph. In consequence of this replacement, they will reduce the number of nodes by eliminating some unnecessary structures.

$$= (t, t) \rightarrow true \tag{1}$$
$$\neq (t, t) \rightarrow false \tag{2}$$
$$= (t, true) \rightarrow t \tag{3}$$
$$\neq (t, true) \rightarrow \neg t \tag{4}$$
$$= (t, false) \rightarrow \neg t \tag{5}$$
$$\neq (t, false) \rightarrow t \tag{6}$$

The first set of general simplification rules simplifies applied numerical and boolean comparison expressions. In these rules, the term t represents a structure of value computing (e.g., the computation of expression $b = x \neq true$). The rules 3, 4, 5, and 6 only apply on the boolean type. These rules are self explanatory, for instance, with any structure represented by a term t, the expression $t = t$ can always be replaced with the value `true`.

The second set of general simplification rules eliminates unnecessary nodes in the graph that represent the ϕ-functions, where c, c_1 and c_2 are boolean expressions. For better representation, we divide this set of rules into several subset as follows.

$$\phi(true, x_1, x_2) \rightarrow x_1 \tag{7}$$
$$\phi(false, x_1, x_2) \rightarrow x_2 \tag{8}$$

The rules in this set replace a ϕ-function with its left branch if the condition always holds the value `true`. Otherwise, if the condition holds the value `false`, it is replaced with its right branch.

$$\phi(c, false, true) \rightarrow \neg c \tag{9}$$
$$\phi(c, true, false) \rightarrow c \tag{10}$$

The rules operate on boolean expressions represented by the branches. When the branches are boolean constants and holds different values, the ϕ-function can be replaced with the value of the condition c.

$$\phi(c, false, x) \rightarrow \neg c \wedge x \tag{11}$$
$$\phi(c, true, x) \rightarrow c \vee x \tag{12}$$
$$\phi(c, x, false) \rightarrow c \wedge x \tag{13}$$
$$\phi(c, x, true) \rightarrow \neg c \vee x \tag{14}$$

The rules operate on boolean expressions represented by the branches. When one of the branches is boolean constant, the ϕ-function can be replaced with a boolean expression of the condition c and the non-constant branch. For instance, when the left branch is a constant and holds the value `true`, then the ϕ-function is replaced with the boolean expression $c \vee x$.

$$\phi(c, x, x) \rightarrow x \tag{15}$$

The rule 15 removes the ϕ-function if all of its branches contain the same value. A ϕ-function with only one branch is a special case of this rule. It indicates that there is only one path to the

ϕ-function as happens with branch elimination.

$$\phi(c, \phi(c, x_1, x_2), x_3) \rightarrow \phi(c, x_1, x_3) \tag{16}$$
$$\phi(c, x_1, \phi(c, x_2, x_3)) \rightarrow \phi(c, x_1, x_3) \tag{17}$$
$$\phi(c, \phi(\neg c, x_1, x_2), x_3) \rightarrow \phi(c, x_2, x_3) \tag{18}$$
$$\phi(c, x_1, \phi(\neg c, x_2, x_3)) \rightarrow \phi(c, x_1, x_2) \tag{19}$$
$$\phi(c_1, \phi(c_2, x_1, x_2), x_3) \rightarrow \phi(c_1, x_1, x_3) \text{ if } c_1 \Rightarrow c_2 \tag{20}$$
$$\phi(c_1, \phi(c_2, x_1, x_2), x_3) \rightarrow \phi(c_1, x_2, x_3) \text{ if } c_1 \Rightarrow \neg c_2 \tag{21}$$
$$\phi(c_1 \wedge c_2, \phi(c_1, x_1, x_2), x_3) \rightarrow \phi(c_1 \wedge c_2, x_1, x_3) \tag{22}$$
$$\phi(c_1 \wedge c_2, \phi(c_2, x_1, x_2), x_3) \rightarrow \phi(c_1 \wedge c_2, x_1, x_3) \tag{23}$$
$$\phi(c_1, x_1, \phi(c_2, x_2, x_3)) \rightarrow \phi(c_1, x_1, x_2) \text{ if } \neg c_1 \Rightarrow c_2 \tag{24}$$
$$\phi(c_1, x_1, \phi(c_2, x_2, x_3)) \rightarrow \phi(c_1, x_1, x_3) \text{ if } \neg c_1 \Rightarrow \neg c_2 \tag{25}$$
$$\phi(c_1 \vee c_2, x_1, \phi(c_1, x_2, x_3)) \rightarrow \phi(c_1 \vee c_2, x_1, x_3) \tag{26}$$
$$\phi(c_1 \vee c_2, x_1, \phi(c_2, x_2, x_3)) \rightarrow \phi(c_1 \vee c_2, x_1, x_3) \tag{27}$$

Consider a ϕ-function such that one of its branches is another ϕ-function. The rules 16 to 27 removes the ϕ-function in the branch if one of the following conditions are satisfied:

- The conditions of the ϕ-functions are the same (as in the rules 16 and 17).
- The condition of the first ϕ-function is equivalent to the negation of the condition of the second ϕ-function (as in the rules 18 and 19).
- The condition of the first ϕ-function either implies the condition of the second ϕ-function or the negation (as in the rules 20 to 23).
- The negation of the condition of the first ϕ-function either implies the condition of the second ϕ-function or the negation (as in the rules 24 to 27).

The following code segment in C shows the use of the rewrite rules above:

```c
if (c)
{
   a = 0; b = 0; d = a;
}
else
{
   a = 1; b = 1; d = 0;
}
if (a == b)
   x = d;
else
   x = 1;
return x;
```

If we analyze this code segment the return value is 0. In fact, a and b have the same value in both branches of the first "if" statement. Thus in the second "if" statement the condition is always `true`, then x always holds the value of d which is 0. We will apply the general simplification rules to show that the value-graph of this code segment can be transformed to the value-graph of the value 0. We represent the value-graph in form of linear notation. The value-graph of the computation of x is $\phi(= (a, b), d, 0)$. Replace the definition of a, b and d, and normalize this graph, we get:

$$\begin{aligned}
x \mapsto & \phi(= (\phi(c,0,1), \phi(c,0,1)), \phi(c, \phi(c,0,1), 0), 0) & \\
& \phi(true, \phi(c, \phi(c,0,1),0),0) & \text{by } (1) \\
& \phi(c, \phi(c,0,1),0) & \text{by } (7) \\
& \phi(c,0,0) & \text{by } (16) \\
& 0 & \text{by } (15)
\end{aligned}$$

6.2.2 Optimization-specific rules

Based on the semantics of Signal compiler, we have a number of optimization-specific rules rewrite graphs in a way that reflexes the effects of specific optimizations of the compiler. The affection of these rules does not always reduce the graph or make it simpler. One has to know specific optimizations of the compiler when she want to add them to the set of rewrite rules. In our case, the set of rules for simplifying constants expressions of the Signal compiler are given as follows. Specific rules for constant expressions with numerical operators:

$$+(cst_1, cst_2) \to cst, \text{ where } cst = cst_1 + cst_2 \tag{28}$$
$$*(cst_1, cst_2) \to cst, \text{ where } cst = cst_1 * cst_2 \tag{29}$$
$$-(cst_1, cst_2) \to cst, \text{ where } cst = cst_1 - cst_2 \tag{30}$$
$$/(cst_1, cst_2) \to cst, \text{ where } cst = cst_1 / cst_2 \tag{31}$$

Specific rules for constant expressions with usual logic operators:

$$\neg false \to true \tag{32}$$
$$\neg true \to false \tag{33}$$
$$\wedge(cst_1, cst_2) \to cst, \text{ where } cst = cst_1 \wedge cst_2 \tag{34}$$
$$\vee(cst_1, cst_2) \to cst, \text{ where } cst = cst_1 \vee cst_2 \tag{35}$$

Specific rules for constant expressions with numerical comparison functions:

$$\Box(cst_1, cst_2) \to cst \tag{36}$$

where $\Box =$ `<`, `>`, `=`, `<=`, `>=`, `/=`, and the boolean value cst is the evaluation of the constant expression $\Box(cst_1, cst_2)$ which can hold either the value `false` or `true`.

We also may add a number of rewrite rules that are derived from the list of *rules of inferences* for propositional logic. For example, we have a group of laws for rewriting formulas with `and` operator, such as:

$$\begin{aligned}
\wedge(x, false) &\to false \\
\wedge(x, true) &\to x \\
\wedge(x, \Rightarrow(x,y)) &\to x \wedge y
\end{aligned}$$

Consider the following Signal program and its generated C code, in which the input signal x is present when the other boolean input signal cx is `true`.

```
/* Signal equation */
| x ^= when cx
/* Generated C code */
if (C_cx)
```

```
{
   if (cx)
   {
      if (!r_P_x(&x)) return FALSE;
   }
}
```

When we construct the value-graph of the signal x, it is represented by $x = \phi(\hat{x}, \overline{x}, \#)$ where $\hat{x} \Leftrightarrow \widehat{cx} \wedge \overline{cx}$. In the generated C code, the value of x is read only when the condition $C\_cx \wedge cx$ is `true` that is represented by $x = \phi(C\_cx, \phi(cx, \overline{x}, \#), \#)$. This observation makes us add the following rewrite rule into the systems to mirror the above rewriting of the Signal compiler.

$$\phi(c_1, \phi(c_2, x_1, x_2), x_2) \rightarrow \phi(c_1 \wedge c_2, x_1, x_2) \tag{37}$$

6.2.3 Synchronous rules

In addition to the general and optimization-specific rules, we also have a number of rewrite rules that are derived from the semantics of the code generation mechanism of the Signal compiler. To illustrate why the synchronous rules need to be added in our validator, we consider the Signal program in Listing 9 and the corresponding generated C code in Listing 10. The shared value-graph of the Signal program and its generated C code is given in Figure 29.

Listing 9 : MasterClk in Signal

```
process MasterClk=
(? integer x;
 ! integer z)
 (| z := x default 0
  | pz := z$1 init 0
  | x ^= when (pz <= 1)
  |)
where integer pz
end;
```

Listing 10 : Generated C code of MasterClk

```
EXTERN logical MasterClk_step()
{
   C_x = z <= 1;
   if (C_x)
   {
      if (!r_MasterClk_x(&x)) return FALSE;
   }
   if (C_x) z = x; else z = 0;
   w_MasterClk_z(z);
   MasterClk_step_finalize();
   return TRUE;
}
```

In this example, the fastest clock $\hat{z}$, called the *master clock*. All other clocks are expressed as calculus expression of the master clock (i.e., the clock $\hat{x}$ is an under-sampling of $\hat{z}$ according to the values of boolean expression $pz <= 1$). Such programs, also referred to as endochronous, can be executed in a deterministic way.

Consider the generated C code, we observe that the value of the variable z is always updated. It holds the value of x if C_x is `true`, otherwise it is 0. Therefore, we have the following rule that

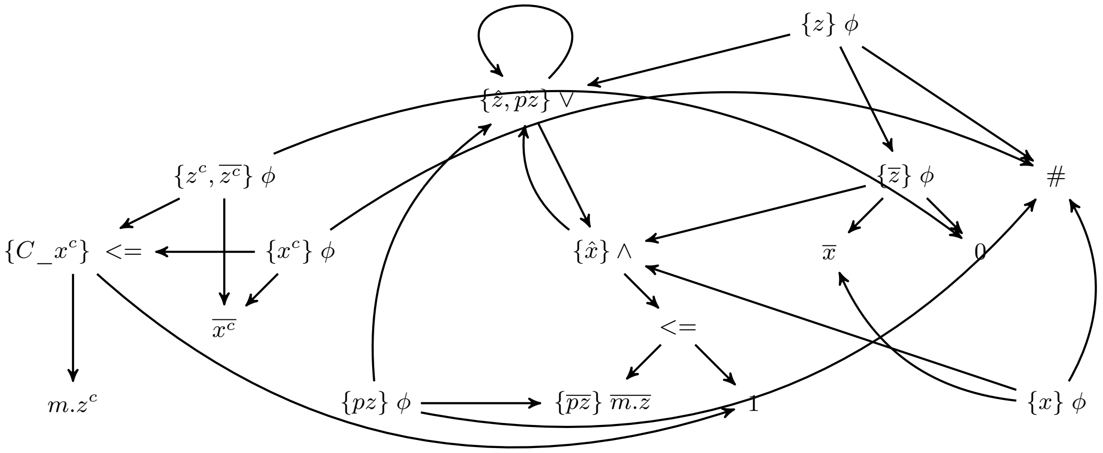


Figure 29: The shared value-graph of MasterClk and its generated C code

mirrors the rewriting of such endochronous programs that be applied by the off-the-shelf Signal compiler.

$$x^c \mapsto \phi(true, \overline{x^c}, \#) \to x \mapsto \phi(true, \overline{x}, \#) \tag{38}$$

We write $x \mapsto \phi(true, \overline{x}, \#)$ to denote that x points to the subgraph rooted at the node labelled by ϕ-function. The rule 38 indicates that if a variable in the generated C code is always updated, then we required that the corresponding signal in the source program is always present meaning that the signal never holds the absent value. In consequence of this rewrite rule, the signal x and its value when it is present $\overline{x}$ (resp. the variable x^c and its updated value $\overline{x^c}$ in the generated C code) point to the same node in the shared value-graph. Every references to x and $\overline{x}$ (resp. x^c and $\overline{x^c}$) point to the same node.

For example, consider the value-graph in Figure 29, we rewrite the subgraph representing the clock $\hat{z}$ and $\widehat{pz}$ into a single node labelled by the value `true`. Then, we apply the rule 7 on the resulting graph, we obtain the reduced graph in Figure 30.

We also have the second synchronous rule that mirrors the semantics of the delay operator of the Signal language. For instance, we consider the equation $pz := z\$1$ init 0 in Listing 9. In the representation of the Signal program, we use the variable $\overline{m.z}$ to capture the last value of the signal z. And in the generated C code the last value of the variable z^c is denoted by $m.z^c$. We require that the last values of a signal and the corresponding variable in the generated C code are the same. That means $\overline{m.z} = m.z^c$.

$$m.x^c \mapsto G_1 + \overline{m.x} \mapsto G_2 \to m.x^c, \overline{m.x} \mapsto G_1 \tag{39}$$

The rule 39 indicates that for any signal x and its corresponding variable in the generated C code which involves in a delay operator, then we required that the last values of x and x^c are the same. That means they are represented by the same subgraph or they point to the same node in the value-graph. In consequence of this rewrite rule, every references to $m.x^c$ and $\overline{m.x}$ point to the same node.

Consider the value-graph in Figure 30, $m.z^c$ and $\overline{m.z}$ point to the same node by 39. Then, C_x and $\hat{x}$ are represented by the same subgraph and any references to them from the ϕ-functions

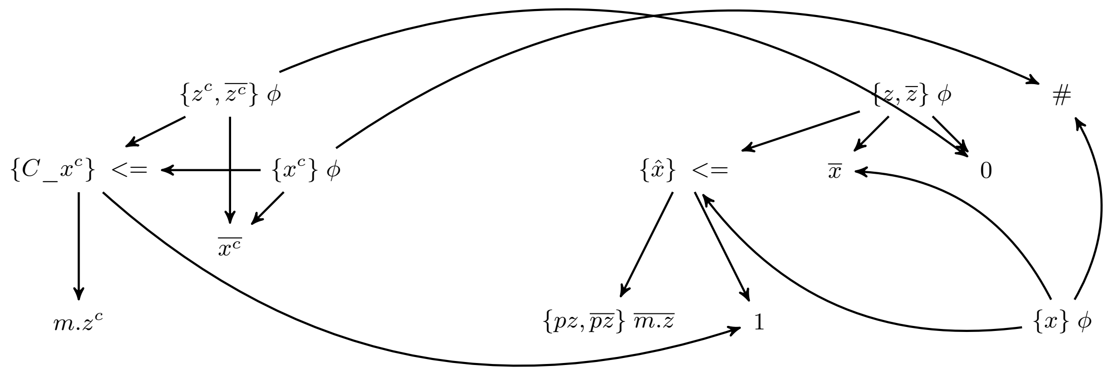


Figure 30: The resulting graph of MasterClk and its generated C code 38

which are labelled by $z^c, \overline{z^c}, z$ and $\overline{z}$ point to the same node in the graph as depicted in Figure 31.

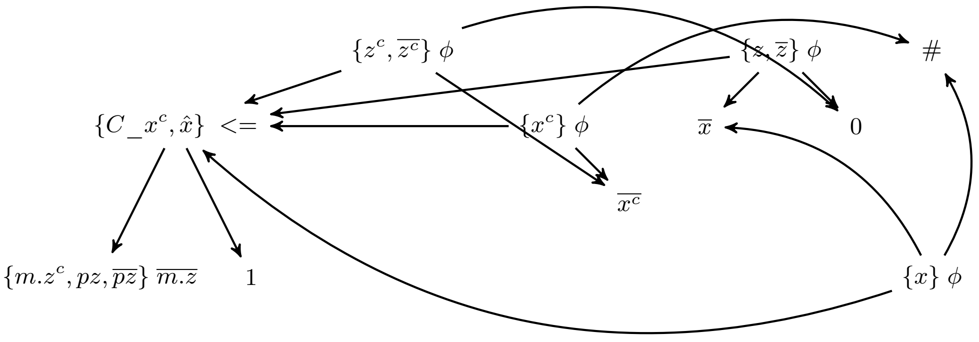

$\{z^c, \overline{z^c}\}\ \phi$
$\{z, \overline{z}\}\ \phi$
$\#$
$\{C\_x^c, \hat{x}\}\ <=$
$\{x^c\}\ \phi$
$\overline{x}$
0
$\overline{x^c}$
$\{m.z^c, pz, \overline{pz}\}\ \overline{m.z}$
1
$\{x\}\ \phi$

Figure 31: The resulting graph of MasterClk and its generated C code by 39

Finally, we add rules that mirrors the assumption on input signals and their corresponding variables in the generated C code. For any input signal x and the corresponding variable x^c in the generated C code, if x is present, then the value of x which is read from the environment then the value of the variable x^c after the reading statement in the C code must be equivalent. That means $\overline{x^c}$ and $\overline{x}$ are represented by the same subgraph in the shared value-graph. We require that the clock of the input signal x is equivalent to the condition in which the variable x^c is updated if the clock of x is also read from the environment as a parameter. Meaning that we represent $\hat{x}$ and C_x^c by the same subgraph in the shared value-graph.

$$\overline{x^c} \mapsto G_1 + \overline{x} \mapsto G_2 \rightarrow \overline{x^c}, \overline{x} \mapsto G_1 \tag{40}$$

$$C\_x^c \mapsto G_1 + \hat{x} \mapsto G_2 \rightarrow C\_x^c, \hat{x} \mapsto G_1 \tag{41}$$

In consequence of these rewrite rule, every references to $\hat{x}$ and C_x^c (resp. $\overline{x}$ and $\overline{x^c}$) point to the same node. By rule 40 we obtain the final normalized value-graph of the program in Listing

10 and its generated C code as in Figure 32. We can observe that the value of the signal z and the corresponding variable z^c are presented by the same subgraph. Therefore, we can safely conclude that the value of output signal z and the corresponding variable z^c in the generated C code are equivalent.

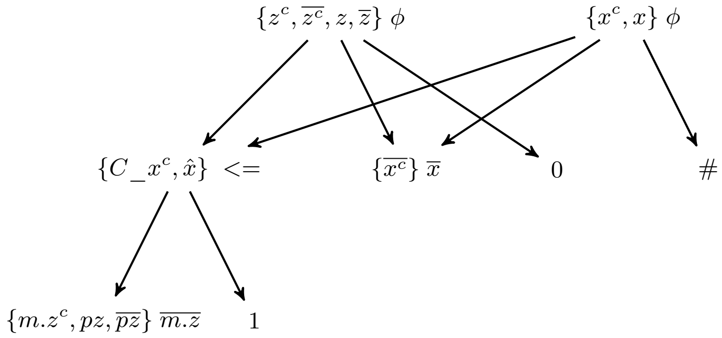

$\{z^c, \overline{z^c}, z, \overline{z}\}\ \phi$
$\{x^c, x\}\ \phi$
$\{C\_x^c, \hat{x}\}\ <=$
$\{\overline{x^c}\}\ \overline{x}$
0
#
$\{m.z^c, pz, \overline{pz}\}\ \overline{m.z}$
1

Figure 32: The final normalized graph of MasterClk and its generated C code

6.3 Implementation

We describe the main steps of our approach, and the main techniques which are used to implement them. This implementation can be integrated into the existing Polychrony toolset to prove the the preservation of value-equivalence of variables.

Given a Signal program A, with an unverified compiler, we consider the following process:

1. The compiler takes program A and compiles it.

2. If there is any error (e.g., syntax error), it outputs an `Error`.

3. Otherwise, it outputs the generated C code $C = Gr(A)$.

These steps can be represented in the following pseudo-code, where $Gr(A)$ is the code generation phase from the source program A to either C program C or compilation errors:

```
if (Gr(A) is Error)
   output Error;
else output C;
```

Now, it is followed by our validator which checks that for any output signal x in the source program A and the corresponding variable x^c in the C program, they have the same value, $\overline{x} = \overline{x^c}$. In other words, $\overline{x}$ and $\overline{x^c}$ point to the same node in the shared value-graph representing the computations of signals and variables in A and C. We denote this fact by $C \sqsubseteq_{value} A$.

```
if (Gr(A) is Error)
   output Error;
else
{
   if ($C \sqsubseteq_{value} A$)
      output C;
   else output Error;
}
```

This verification process will provide a formal guarantee as strong as that provided by a formal compiler verification approach with the assumption that our validator is formally correct.

The main components of the verification framework is depicted in Figure 33. The validator works as follows. First, it computes the Signal program and the corresponding C program into value-graphs that represent the output computations of the programs. The value-graph can be considered as a generalization of the result of symbolic evaluation. Then, the resulting value-graphs is used to construct a single graph by allowing sharing between the two distinct graphs. The shared value-graph is transformed by applying some graph rewrite rules, this process is called normalization. The set of rewrite rules reflexes the general rules of inferences of operators, or the optimizations of the compiler. For instance, consider the 3-node subgraph representing the expression $1 > 0$, the normalization will transform that graph into a single node subgraph representing the value `true`, as it reflexes the constant folding.

Finally, it compares the values of the output signals and the corresponding variables in the C program. For every pair of signal and variable, if their values are equivalent meaning that they point to the same node in the shared graph, the validator return *correct*. Therefore, in the best case, when semantics has been preserved, the value comparison of the output signals and their corresponding variables has constant time complexity $\mathcal{O}(1)$. In fact, it is always expected that most transformations and optimizations are semantics-preserving, thus the best-case complexity is important.

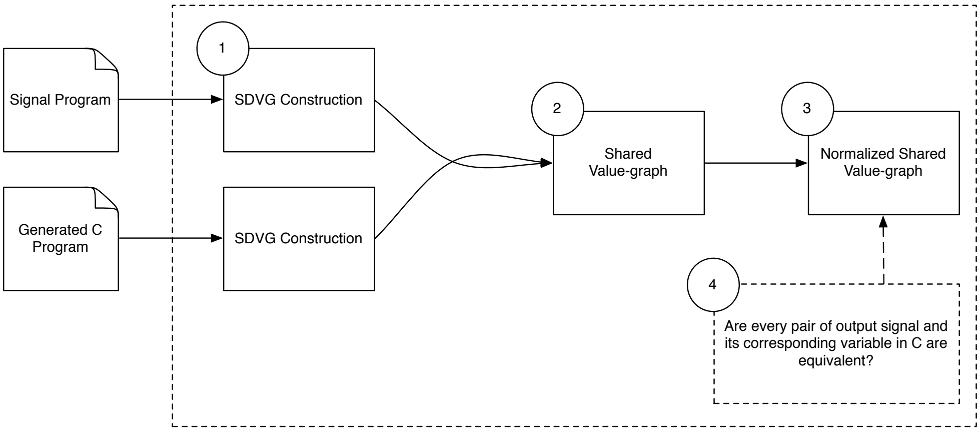

Signal Program
Generated C Program
1
SDVG Construction
SDVG Construction
2
Shared Value-graph
3
Normalized Shared Value-graph
4
Are every pair of output signal and its corresponding variable in C are equivalent?

Figure 33: A bird's-eye view of the SDVG translation validation

Let us illustrate the above steps on the program `DEC` and its generated C code `DEC_step()` in Listing 11 at the code generation phase of the compilation process. In the first step, we will compute the shared value-graph for both programs to represent the computations of signals and corresponding variables which is depicted in Figure 34.

Listing 11 : Generated C code of DEC

```
EXTERN logical DEC_step()
{
   C_FB = N <= 1;
   if (C_FB)
   {
      if (!r_DEC_FB(&FB)) return FALSE;
   }
8      if (C_FB) N = FB; else N = N - 1;
9      w_DEC_N(N);
10     DEC_step_finalize();
11     return TRUE;
12 }
```

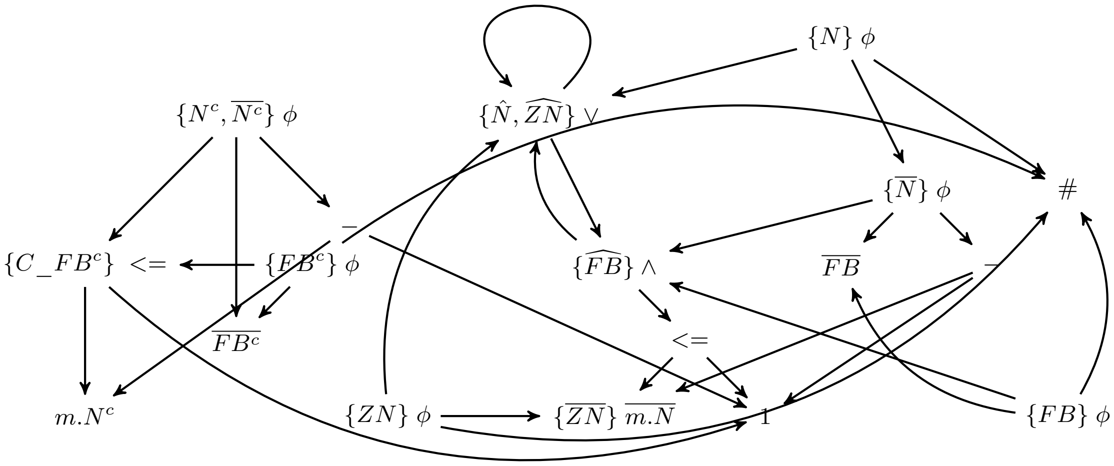

$\{N\}\ \phi$
$\{\hat{N}, \widehat{ZN}\}\ \vee$
$\{N^c, \overline{N^c}\}\ \phi$
$\{\overline{N}\}\ \phi$
$\#$
$-$
$\{C\_FB^c\}\ <=$
$\{\overline{FB^c}\}\ \phi$
$\{\widehat{FB}\}\ \wedge$
$\overline{FB}$
$-$
$\overline{FB^c}$
$<=$
$m.N^c$
$\{ZN\}\ \phi$
$\{\overline{ZN}\}\ \overline{m.N}$
1
$\{FB\}\ \phi$

Figure 34: The shared value-graph of DEC and its generated C code

First, note that in the C program the variable N^c ("c" is added as superscript for the C program variables, to distinguish them from the signals in the Signal program) is always updated as in line (8). In lines (3) and (8), the references to the variable N^c are the references to the last value of N^c which is denoted by $m.N^c$. The variable FB^c which corresponds to the input signal FB is updated only when the variable C_FB^c is `true`.

In the second step, we will normalize the above initial shared value-graph. Below is a potential normalization scenario. Figure 35 depicts the intermediate resulting value-graph of this normalization scenario. And Figure 36 is the final normalized value-graph from the initial graph when we cannot perform any more normalization.

- The clock of the output signal N is a master clock which is indicated in the generated C that the variable N^c is always updated. By rule 38, the nodes $\{\hat{N}, \widehat{ZN}\}\ \vee$ is rewritten into `true`.
- By rule $\wedge(true, x) \rightarrow x$, the nodes $\{\widehat{FB}\}\ \wedge$ is rewritten into $\{\widehat{FB}\}\ <=$.
- The node ϕ-function represents the computation of N is removed and N points to the node $\{\overline{N}\}\ \phi$ by rule 7.
- The node ϕ-function represents the computation of NN is removed and NN points to the node $\{\overline{ZN}\}\ \overline{m.N}$ by rule 7.
- The nodes $\overline{FB^c}$ and $\overline{FB}$ are rewritten into single node $\{\overline{FB}\}\ \overline{FB^c}$ by rule 40. And all references to them are replaced by the references to $\{\overline{FB}\}\ \overline{FB^c}$.
- The nodes $m.N^c$ and $\overline{m.N}$ are rewritten into single node $\{\overline{m.N}\}\ m.N^c$ by rule 39. And all references to them are replaced by the references to $\{\overline{m.N}\}\ m.N^c$.

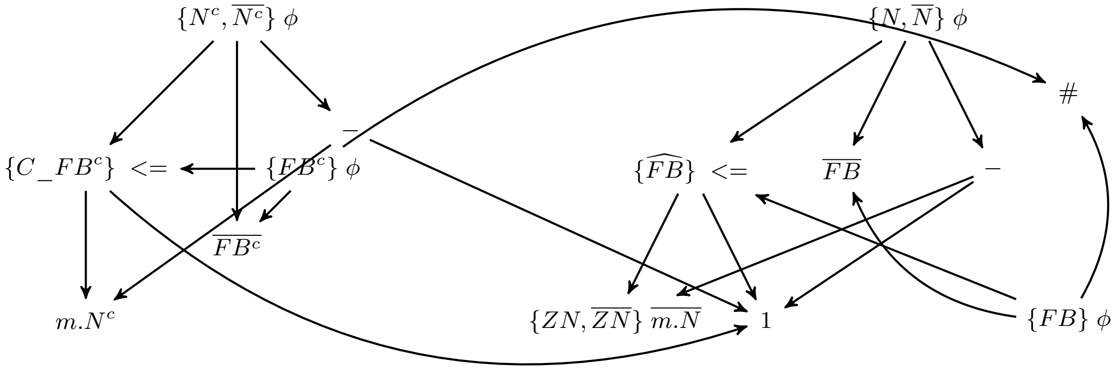

$\{N^c, \overline{N^c}\}$ ϕ
$\{N, \overline{N}\}$ ϕ
$\#$
$-$
$\{C\_FB^c\}$ $<=$
$\{\widehat{FB^c}\}$ ϕ
$\{\widehat{FB}\}$ $<=$
$\overline{FB}$
$-$
$\overline{FB^c}$
$m.N^c$
$\{ZN, \overline{ZN}\}$ $\overline{m.N}$
1
$\{FB\}$ ϕ

Figure 35: The resulting value-graph of DEC and its generated C code

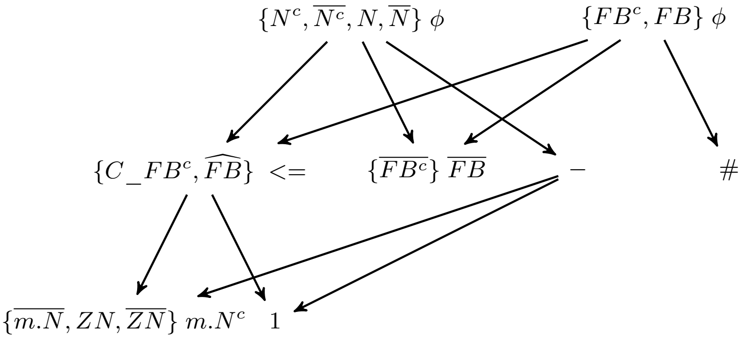

$\{N^c, \overline{N^c}, N, \overline{N}\}$ ϕ
$\{FB^c, FB\}$ ϕ
$\{C\_FB^c, \widehat{FB}\}$ $<=$
$\{\overline{FB^c}\}$ $\overline{FB}$
$-$
$\#$
$\{\overline{m.N}, ZN, \overline{ZN}\}$ $m.N^c$
1

Figure 36: The final normalized graph of DEC and its generated C code

In the final step, we check that the value of the output signal and its corresponding variable in the generated code merge into a single node. In this example, we can safely conclude that the output signal N and its corresponding variable N^c is equivalent since they point to the same node in the final normalized value-graph.

7 Discussion

There is a wide range of works for value-graph representations of expression evaluations in a program. For example, in [13], Weise et al. present a nice summary of the various type of value-graph.

Another related work which adopts the translation validation approach in verification of optimizations. Tristan et al. [14] recently proposed a framework for translation validation of LLVM optimizer. For a function and its optimized counterpart, they compute a shared value-graph. The graph is *normalized* (roundly speaking, the graph is reduced). After the normalizing, if the outputs of two functions are represented by the same sub-graph, they can safely conclude that two functions are equivalent.

On the other hand, Tate et al. [15] proposed a framework for translation validation. Given a function in the input program and the corresponding optimized version of the function in the output program, they compute two value-graphs to represent the computations of the variables. They then transform the graph by adding equivalent terms through a process called *equality saturation*. After the saturation, if two value-graph are the same, they can conclude that the return value of two given functions are the same. However, for the translation validation purpose, our normalization process is more efficient and scalable since we can add some rewrite rules into the validator that reflexes what a typical compiler intends to do (e.g., a compiler will do the constant folding optimization, then we can add the rewrite rule for constant expressions such as $1 + 2$ is replaced by a single node 3).

The present chapter provides a proof of correctness of a multi-clocked synchronous programming language compiler for the preservation of value-equivalence of variables and applies this approach to the synchronous data-flow compiler Signal. We believe that the translation validation of synchronous data-flow value-graph of the industrial compiler Signal without instrumentation is feasible and efficient.

References

[1] Inria/Espresso, “Polychrony toolset.” http://www.irisa.fr/espresso/Polychrony, 2013.

[2] A. Benveniste and P. L. Guernic, “Hybrid dynamical systems theory and the signal language,” *IEEE Transactions on Automatic Control*, vol. 35(5), pp. 535–546, 1990.

[3] T. Gautier, P. L. Guernic, and L. Besnard, “Signal, a declarative language for synchronous programming of real-time systems,” *In Proc. 3rd. Conf. on Functional Programming Languages and Computer Architecture, LNCS 274, Springer Verlag*, 1990.

[4] L. Besnard, T. Gautier, P. L. Guernic, and J.-P. Talpin, “Compilation of polychronous data-flow equations,” *In Synthesis of Embedded Software Springer*, pp. 1–40, 2010.

[5] O. Maffeï and P. L. Guernic, “Distributed implementation of signal: Scheduling and graph clustering,” *In 3rd International School and Symposium on Formal Techniques in Real-time and Fault-tolerant Systems, LNCS*, vol. 863, pp. 547–566, 1994.

[6] T. Gautier and P. L. Guernic, “Code generation in the sacres project,” *In Towards System Safety, Proceedings of the Safety-critical Systems Symposium*, pp. 127–149, 1999.

[7] P. Aubry, P. L. Guernic, and S. Machard, “Synchronous distribution of signal programs,” *In Proceedings of the 29th Hawaii International Conference on System Sciences, IEEE Computer Society Press*, vol. 1, pp. 656–665, 1996.

[8] H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg, *Handbook of Graph Grammars and Computing by Graph Transformation, Vol.2: Applications, Languages and Tools*. World Scientific, 1999.

[9] R. Ballance, A. Maccabe, and K. Ottenstein, “The program dependence web: A representation supporting control, data, and demand driven interpretation of imperative languages,” *In Proceedings of the SIGPLAN’90 Conference on Programming Language Design and Implementation*, pp. 257–271, 1990.

[10] H. Barendregt, M. van Eekelen, J. Glauert, J. Kennaway, M. Plasmeijer, and M. Sleep, “Towards an intermediate language based on graph rewriting,” *These Proceedings van den Broek, P.M and G.F van der Hoeven*, 1987.

[11] D. Turner, “A new implementation technique for applicative languages,” *In Software: Practice and Experience*, vol. 9, pp. 31–49, 1979.

[12] H. Ehrig, H.-J. Kreowski, U. Montanari, and G. Rozenberg, *Handbook of Graph Grammars and Computing by Graph Transformation, Vol.3: Concurrency, Parallellism, and Distribution*. World Scientific, 1999.

[13] D. Weise, R. Crew, M. Ernst, and B. Steensgaard, “Value dependence graphs: Representation without taxation,” *In 21th Principles of Programming Languages*, pp. 297–310, 1994.

[14] J.-B. Tristan, P. Govereau, and G. Morrisett, “Evaluating value-graph translation validation for llvm,” *In ACM SIGPLAN Conference on Programming and Language Design Implementation*, 2011.

[15] R. Tate, M. Stepp, Z. Tatlock, and S. Lerner, “Equility saturation: A new approach to optimization,” *In 36th Principles of Programming Languages*, pp. 264–276, 2009.

RESEARCH CENTRE
RENNES – BRETAGNE ATLANTIQUE

Campus universitaire de Beaulieu
35042 Rennes Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399