Translation Validation for Synchronous Data-flow Specification in the SIGNAL Compiler

van Chan Ngo, Jean-Pierre Talpin, Thierry Gautier

Translation Validation for Synchronous Data-flow Specification in the SIGNAL Compiler

Van Chan Ngo, Jean-Pierre Talpin, and Thierry Gautier

INRIA, 35042 Rennes cedex, France
{chan.ngo,jean-pierre.talpin,thierry.gautier}@inria.fr

Abstract. We present a method to construct a validator based on translation validation approach to prove the value-equivalence of variables in the SIGNAL compiler. The computation of output *signals* in a SIGNAL program and their counterparts in the generated C code is represented by a *Synchronous Data-flow Value-Graph* (SDVG). The validator proves that every output signal and its counterpart variable have the same values by transforming the SDVG graph.

Keywords: Value-Graph, Graph Transformation, Formal Verification, Translation Validation, Certified Compiler, Synchronous Programs.

1 Introduction

A compiler is a large and very complex program which often consists of hundreds of thousands, if not millions, lines of code, and is divided into multiple sub-systems and modules. In addition, each compiler implements a particular algorithm in its own way. Consequently, that results in two main drawbacks regarding the formal verification of the compiler itself. First, constructing the specifications of the actual compiler implementation is a long and tedious task. Second, the correctness proof of a compiler implementation, in general, cannot be reused for another compiler.

To deal with these drawbacks of formally verifying the compiler itself, one can prove that the source program and the compiled program are semantically equivalent, which is the approach of *translation validation* [12,13,2]. The principle of translation validation is as follows: for a given input sample, the source and the compiled programs will give corresponding *execution traces*. These traces are equivalent if they have the same *observation*. An observation is a sequence (finite or infinite) of values (e.g., values of variables, arguments, returned values,...). The compilation is correct if for any input, the source and the compiled programs have observationally equivalent execution traces.

In this work, to adopt the translation validation approach, we use a value-graph as a common semantics to represent the computation of variables in the source and compiled programs. The "correct transformation" is defined by the assertion that every output variable in the source program and the corresponding variable in the compiled program have the same values.

SIGNAL [4,6] 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 $\perp$). Given two signals, they are *synchronous* iff 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. Equations and processes are concurrent.

A Synchronous Data-flow Value-Graph symbolically represents the computation of the output signals in a SIGNAL program and their counterparts in its generated C code. The same structures are shared in the graph, meaning that they are represented by the same subgraphs. Suppose that we want to show that an output signal and its counterpart have the same values. We simply need to check that whether they are represented by the same subgraphs, meaning they are label the same node. We manage to realize this check by transforming the graph using some rewrite rules, which is called *normalizing* process.

Let A and C be the source program and its generated C code. Cp denotes the unverified SIGNAL compiler which compiles A into $C = Cp(A)$ or a compilation error. We now associate Cp with a validator checking that for any output signal x in A and the corresponding variable x^c in C, they have the same values (denoted by $\widetilde{x} = \widetilde{x^c}$). We denote this fact by $C \sqsubseteq_{val} A$.

```
if (Cp(A) is Error) return Error;
else {
   if (C ⊑val A) return C;
   else return Error;
}
```

The main components of the validator are depicted in Fig. 1. It works as follows. First, a shared value-graph that represents the computation of all signals and variables in both programs is constructed. The value-graph can be considered as a generalization of symbolic evaluation. Then, the shared value-graph is transformed by applying graph rewrite rules (the normalization). The set of rewrite rules reflexes the general rules of inference 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, the validator compares the values of the output signals and the corresponding variables in the C code. For every output signal and its corresponding variable, the validator checks whether they point to the same node in the graph, meaning that their computation is represented by the same subgraph. Therefore, in the best case, when semantics has been preserved, this check 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.

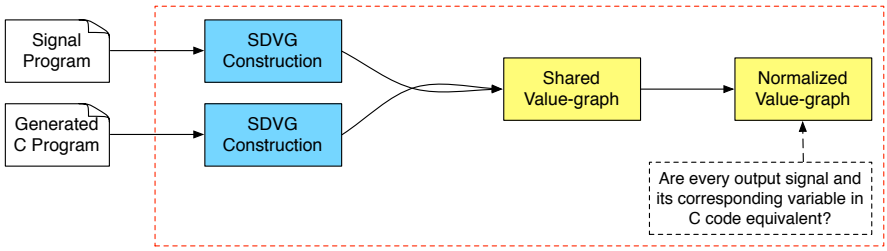

Fig. 1: SDVG Translation Validation Architecture

This work is a part of the whole work of the SIGNAL compiler formal verification. Our approach is that we separate the concerns and prove each analysis and transformation stage of the compiler separately with respect to ad-hoc data-structures to carry the semantic information relevant to that phase.

The preservation of the semantics can be decomposed into the preservation of clock semantics at the *clock calculation and Boolean abstraction* phase, the preservation of data dependencies at the *static scheduling* phase, and value-equivalence of variables at the *code generation* phase.

Fig. 2 shows the integration of this verification framework into the compilation process of the SIGNAL compiler. For each phase, the validator takes the source program and its compiled counterpart, then constructs the corresponding formal models of both programs. Finally, it checks the existence of the *refinement* relation to prove the preservation of the considered semantics. If the result is that the relation does not exist then a "compiler bug" message is emitted. Otherwise, the compiler continues its work.

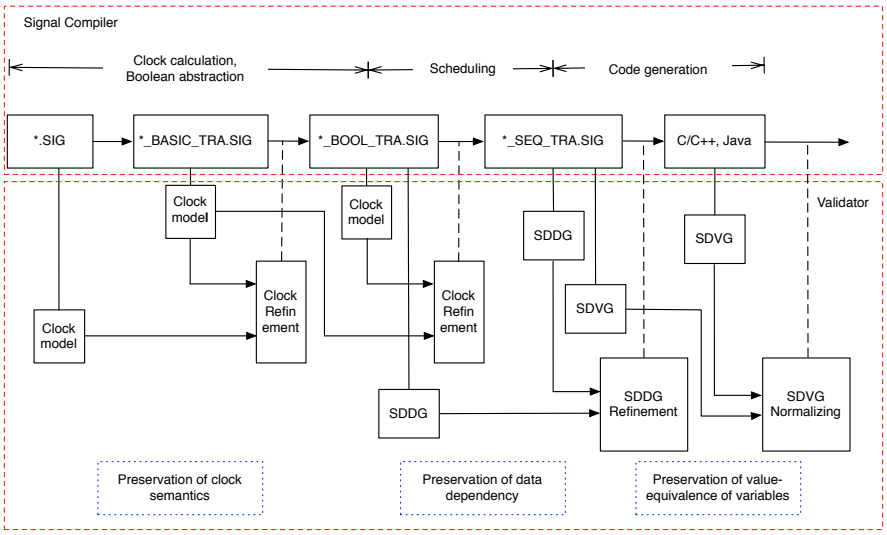

Fig. 2: The Translation Validation for the SIGNAL Compiler

The remainder of this paper is organized as follows. In Section 2, we consider the formal definition of SDVG and the representation of a SIGNAL program and its generated C code as a shared SDVG. Section 3 addresses the mechanism of the verification process based on the normalization of a SDVG. Section 4 illustrates the concept of SDVG and the verification procedure. Section 5 terminates this paper with some related work, a conclusion and an outlook to future work.

2 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 code, and F be the set of function symbols. In our consideration, F contains usual logic operators (`not`, `and`, `or`), numerical comparison functions (`<`, `>`, `=`, `<=`, `>=`, `/=`), numerical operators (`+`, `-`, `*`, `/`), and gated ϕ-function [3]. A gated ϕ-function such as $x = \phi(c, x_1, x_2)$ represents a branching in a program, which means x takes the value of x_1 if the condition c is satisfied, and the value of x_2 otherwise. A constant is defined as a function symbol of arity 0.

Definition 1. *A* SDVG *associated with a* SIGNAL *program and its generated C code is a directed graph* $G = \langle N, E, I, O, l_N, m_N \rangle$ *where* N *is a finite set of nodes that represent clocks, signals, variables, or functions.* $E \subseteq N \times N$ *is the set of edges that describe the computation relations between nodes.* $I \subseteq N$ *is the set of input nodes that are the input signals and their corresponding variables in the generated C code.* $O \subseteq N$ *is the set of output nodes that are the output signals and their corresponding variables in the generated C code.* $l_N : N \longrightarrow X \cup F$ *is a mapping labeling each node with an element in* $X \cup F$*.* $m_N : N \longrightarrow \mathcal{P}(N)$ *is a mapping labeling each node with a finite set of clocks, signals, and variables. It defines the set of equivalent clocks, signals and variables.*

A subgraph rooted at a node is used to describe the computation of the corresponding element labelled at this node. In a graph, for a node labelled by y, the set of clocks, signals or variables $m_N(y) = \{x_0, ..., x_n\}$ is written as a node with label $\{x_0, ..., x_n\}$ y.

2.1 SDVG of SIGNAL Program

Let P be a SIGNAL program, we write $X = \{x_1, ..., x_n\}$ to denote the set of all signals in P which consists of input, output, state (corresponding to delay operator) and local signals, denoted by I, O, S and L, respectively. For each $x_i \in X$, $\mathbb{D}_{x_i}$ denotes its domain of values, and $\mathbb{D}_{x_i}^{\perp} = \mathbb{D}_{x_i} \cup \{\perp\}$ is the domain of values with the absent value. Then, the domain of values of X with absent value is defined as follows: $\mathbb{D}_X^{\perp} = \bigcup_{i=1}^{n} \mathbb{D}_{x_i} \cup \{\perp\}$. For each signal x_i, it is associated with a Boolean variable $\widehat{x_i}$ to encode its clock at a given instant t (`true`: x_i is present at t, `false`: x_i is absent at t), and $\widetilde{x_i}$ with the same type as x_i to encode its value. Formally, the abstract values to represent the clock and value of a signal can be represented by a gated ϕ-function, $x_i = \phi(\widehat{x_i}, \widetilde{x_i}, \perp)$.

Assume that the computation of signals in processes P_1 and P_2 is represented as shared value-graphs G_1 and G_2, respectively. Then the value-graph G of the synchronous combination process $P_1|P_2$ can be defined as $G = \langle N, E, I, O, m_N \rangle$ in which for any node labelled by x, we replace it by the subgraph that is rooted by the node labelled by x 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 P can be constructed as a combination of the sub-value-graphs of its equations.

A SIGNAL program is built through a set of primitive operators. Therefore, to construct the SDVG of a SIGNAL program, we construct a subgraph for each primitive operator. In the following, we present the value-graph corresponding to each SIGNAL primitive operator.

Stepwise Functions Consider the equation using the stepwise functions $y := f(x_1, ..., x_n)$, it indicates that if all signals 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: $y = \phi(\hat{y}, f(\widetilde{x_1}, \widetilde{x_2}, ..., \widetilde{x_n}), \perp)$, where $\hat{y} \Leftrightarrow \widehat{x_1} \Leftrightarrow \widehat{x_2} \Leftrightarrow ... \Leftrightarrow \widehat{x_n}$ (since by definition they are *synchronous*). The graph representation of the stepwise functions is depicted in Fig. 3. Note that in the graph, the node labelled by $\{\widehat{x_1}, ..., \widehat{x_n}\}\ \hat{y}$ means that $m_N(\hat{y}) = \{\widehat{x_1}, ..., \widehat{x_n}\}$. In other words, the subgraph representing the computation of $\hat{y}$ is also the computation of $\widehat{x_1}$, ..., and $\widehat{x_n}$.

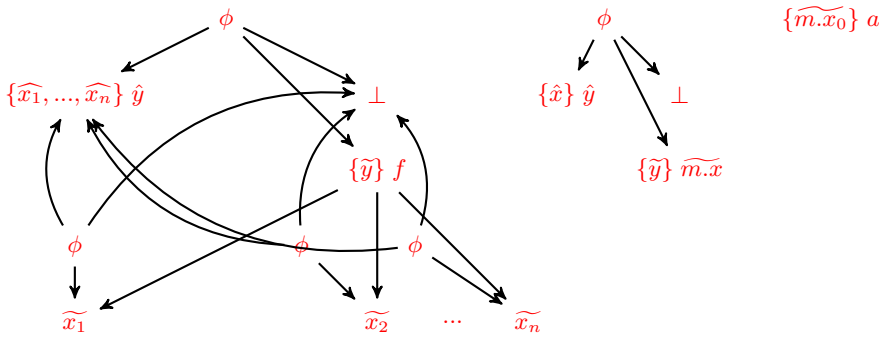

Fig. 3: The graphs of $y := f(x_1, ..., x_n)$ and $y := x\$1$ init a

Delay Consider the equation using the delay operator $y := x\$1$ 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. The computation of y can be represented by the following nodes: $y = \phi(\hat{y}, \widetilde{m.x}, \perp)$ and $\widetilde{m.x_0} = a$, where $\hat{y} \Leftrightarrow \hat{x}$. $\widetilde{m.x}$ and $\widetilde{m.x_0}$ are the last value of x and the initialized value of y. The graph representation is depicted in Fig. 3.

Merge Consider the equation 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: $y = \phi(\hat{y}, \phi(\hat{x}, \widetilde{x}, \widetilde{z}), \bot)$, where $\hat{y} \Leftrightarrow (\hat{x} \vee \hat{z})$. The graph representation is depicted in Fig. 4. Note that in the graph, the clock $\hat{y}$ is represented by the subgraph of $\hat{x} \vee \hat{z}$.

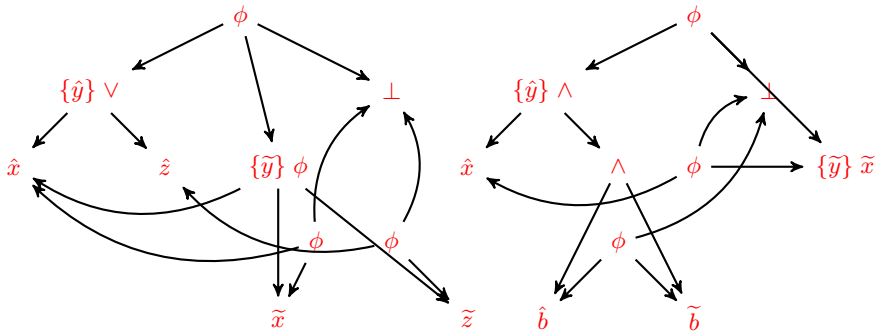

Fig. 4: The graphs of $y := x$ default z and $y := x$ when b

Sampling Consider the equation 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: $y = \phi(\hat{y}, \widetilde{x}, \bot)$, where $\hat{y} \Leftrightarrow (\hat{x} \wedge \hat{b} \wedge \widetilde{b})$. Fig. 4 shows its graph representation.

Restriction The graph representation of restriction process $P_1 \backslash x$ is the same as the graph of P_1.

Clock Relations Given the above graph representations of the primitive operators, we can obtain the graph representations for the derived operators on clocks as the following gated ϕ-function $z = \phi(\hat{z}, true, \bot)$, where $\hat{z}$ is computed as $\hat{z} \Leftrightarrow \hat{x}$ for $z := \hat{x}$, $\hat{z} \Leftrightarrow (\hat{x} \vee \hat{y})$ for $z := x \hat{+}\ y$, $\hat{z} \Leftrightarrow (\hat{x} \wedge \hat{y})$ for $z := x \hat{*}\ y$, $\hat{z} \Leftrightarrow (\hat{x} \wedge \neg\hat{y})$ for $z := x \hat{-}\ y$, and $\hat{z} \Leftrightarrow (\hat{b} \wedge \widetilde{b})$ for $z :=$ when b. For the clock relation $x \hat{=}\ y$, it is represented by a single node graph labelled by $\{\hat{x}\}\ \hat{y}$.

2.2 SDVG of Generated C Code

For constructing the shared value-graph, the generated C code is translated into a subgraph along with the subgraph of the SIGNAL program. Let A be a SIGNAL program and C its generated C code, we write $X_A = \{x_1, ..., x_n\}$ to denote the set of all signals in A, and $X_C = \{x_1^c, ..., x_m^c\}$ to denote the set of all variables in

C. We added "c" as superscript for the variables, to distinguish them from the signals in A.

As described in [5,8,7,1], the generated C code of A consists of the following files:

- `A_main.c` is 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` is the implementation of the initialization function and the step function. The initialization function is called once to provide initial values to the program variables. The step function, which contains also the step initialization and finalization functions, is responsible for the calculation of the outputs to interact with the environment. This function, which is called repeatedly in an infinite loop, is the essential part of the concrete code.
- `A_io.c` is the implementation of the *IO communication functions*. The IO functions are called to setup communication channels with the environment.

The scheduling and the computations are done inside the step function. Therefore, it is natural to construct a graph of this function in order to prove that its variables and the corresponding signals have the same values. To construct the graph of the step function, the following considerations need to be studied.

The generated C code in the step function consists of only the assignment and `if-then` statements. For each signal named x in A, it has a corresponding Boolean variable named C_x in the step function. Then the computation of x is implemented by a conditional `if-then` statement 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 an 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 always have some value) to implement the SIGNAL program A which uses volatile variables. As a result, there is a difference in the types of a signal in the SIGNAL program and of the corresponding variable in the C code. When a signal has the absent value, $\perp$, at a given instant, the corresponding C variable always has a value. This consideration implies that we have to detect when a variable in the C code such that whose value is not updated. In this case, it will be assigned the absent value, $\perp$. Thus, the computation of a variable, called x^c, can fully be represented by a gated ϕ-function $x^c = \phi(C\_x^c, \widetilde{x^c}, \perp)$, where $\widetilde{x^c}$ denotes the newly updated value of the variable.

In the generated C code, the computation of the variable whose clock is the *master clock*, which ticks every time the step function is called, and the computation of some local variables (introduced by the SIGNAL compiler) are implemented using the following forms:

```
if (C_x) {
  computation(x);
} else computation(x);
// or without if-then
computation(x)
```

It is obvious that x is always updated when the step function is invoked. The computation of such variables can be represented by a single node graph labelled by $\{\widetilde{x^c}\}$ x^c. That means the variable x^c is always updated and holds the value $\widetilde{x^c}$.

Considering the following code segment, we observe that the variable x is involved in the computation of the variable y before the updating of x.

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

In this situation, we refer to the value of x as the previous value, denoted by $m.x^c$. It happens when a *delay* operator is applied on the signal x in the SIGNAL program. The computation of y is represented by the following gated ϕ-function: $y^c = \phi(C\_y^c, m.x^c + 1, \perp)$.

3 Translation Validation of SDVG

In this section, we introduce the set of rewrite rules to transform the shared value-graph resulting from the previous step. This procedure is called *normalizing*. At the end of the normalization, for any output signal x and its corresponding variable x^c in the generated C code, we check whether x and x^c label the same node in the resulting graph. We also provide a method to implement the representation of synchronous data-flow value-graph and adapt the normalizing procedure with any future optimization of the compiler.

3.1 Normalizing

Once a shared value-graph is constructed for the SIGNAL program and its generated C code, if the values of an output signal and its corresponding variable in the C code are not already 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 1.1.

Listing 1.1: Value-graph Normalization

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

```
while (∃s ∈ S or ∃r ∈ R that can be applied on G) {
   while (∃r ∈ R that can be applied on G) {
   for (n ∈ 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 that can be applied to the resulting graph, we maximize the shared nodes, reusing the identical subgraphs. The process terminates when there exists no more sharing or rules that can be applied.

We classify our set of rewrite rules into three basic types: *general simplification rules*, *optimization-specific rules* and *synchronous rules*. In the following, we shall present the rewrite rules of these types, and we assume that all nodes in our shared value-graph are typed. We write a rewrite rule in form of term rewrite rules, $t_l \to t_r$, meaning that the subgraph represented by t_l is replaced by the subgraph represented by t_r when the rule is applied. Due to the lack of space, we only present a part of these rules, the full set of rules is shown in the appendix.

General Simplification Rules The general simplification rules contain the rules which are related to the general rules of inference 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 usual logic operators, 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, we will reduce the number of nodes by eliminating some unnecessary structures. The first set of rules simplifies numerical and Boolean comparison expressions. In these rules, the subgraph t represents a structure of value computing (e.g., the computation of expression $b = x \neq \texttt{true}$). These rules are self explanatory, for instance, with any structure represented by a subgraph t, the expression $t = t$ can always be replaced with a single node subgraph labelled by the value `true`.

$$
\begin{aligned}
= (t, t) &\to \texttt{true} \\
\neq (t, t) &\to \texttt{false}
\end{aligned}
$$

The second set of general simplification rules eliminates unnecessary nodes in the graph that represent the ϕ-functions, where c is a Boolean expression. For instance, we consider the following rules.

$$
\begin{aligned}
\phi(\texttt{true}, x_1, x_2) &\to x_1 \\
\phi(c, \texttt{true}, \texttt{false}) &\to c \\
\phi(c, \phi(c, x_1, x_2), x_3) &\to \phi(c, x_1, x_3)
\end{aligned}
$$

The first rule replaces a ϕ-function with its left branch if the condition always holds the value `true`. The second rule operates on Boolean expressions represented by the branches. When the branches are Boolean constants and hold different values, the ϕ-function can be replaced with the value of the condition c. Consider a ϕ-function such that one of its branches is another ϕ-function. The third rule removes the ϕ-function in the branches if the conditions of the ϕ-functions are the same.

Optimization-specific Rules Based on the optimizations of the SIGNAL compiler, we have a number of optimization-specific rules in a way that reflexes the effects of specific optimizations of the compiler. These rules do not always reduce the graph or make it simpler. One has to know specific optimizations of the compiler when she wants to add them to the set of rewrite rules. In our case, the set of rules for simplifying constant expressions of the SIGNAL compiler such as:

$$
\begin{array}{l}
+(cst_1, cst_2) \rightarrow cst, \text{ where } cst = cst_1 + cst_2 \\
\wedge(cst_1, cst_2) \rightarrow cst, \text{ where } cst = cst_1 \wedge cst_2 \\
\square(cst_1, cst_2) \rightarrow cst
\end{array}
$$

where $\square$ denotes a numerical comparison function, and the Boolean value cst is the evaluation of the constant expression $\square(cst_1, cst_2)$ 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 inference* for propositional logic. For example, we have a group of laws for rewriting formulas with `and` operator, such as:

$$
\begin{array}{ll}
\wedge(x, \texttt{true}) & \rightarrow x \\
\wedge(x, \Rightarrow (x, y)) & \rightarrow x \wedge y
\end{array}
$$

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.

The first rule is that if a variable in the generated C code is always updated, then we require that the corresponding signal in the source program is present at every instant, 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 $\tilde{x}$ (resp. the variable x^c and its updated value $\widetilde{x^c}$ in the generated C code) point to the same node in the shared value-graph. Every reference to x and $\tilde{x}$ (resp. x^c and $\widetilde{x^c}$) point to the same node.

We consider the equation $pz := z\$1$ `init` 0. We use the variable $\widetilde{m.z}$ to capture the last value of the signal z. In the generated C program, the last value of the variable z^c is denoted by $m.z^c$. The second rule is that it is required that the last values of a signal and the corresponding variable in the generated C code are the same. That means $\widetilde{m.z} = m.z^c$.

Finally, we add rules that mirror the relation between input signals and their corresponding variables in the generated C code. First, 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 and the value of the variable x^c after the reading statement must be equivalent. That means $\widetilde{x^c}$ and $\widetilde{x}$ are represented by the same subgraph in the graph. Second, if the clock of x is also read from the environment as a parameter, then the clock of the input signal x is equivalent to the condition in which the variable x^c is updated. It means that we represent $\hat{x}$ and C_x^c by the same subgraph. Consequently, every reference to $\hat{x}$ and C_x^c (resp. $\widetilde{x}$ and $\widetilde{x^c}$) points to the same node.

4 Illustrative Example

Let us illustrate the verification process in Fig. 1 on the program `DEC` in Listing 1.2 and its generated C code `DEC_step()` in Listing 1.3.

In the first step, we shall compute the shared value-graph for both programs to represent the computation of all signals and their corresponding variables. This graph is depicted in Fig. 5.

Listing 1.2: DEC in Signal

```
process DEC=
(? integer FB;
 ! integer N)
 (| FB ^= when (ZN<=1)
  | N := FB default (ZN-1)
  | ZN := N$1 init 1
  |)
where integer ZN init 1
end;
```

Listing 1.3: Generated C code of DEC

```
EXTERN logical DEC_step() {
   C_FB = N <= 1;
   if (C_FB) {
      if (!r_DEC_FB(&FB)) return FALSE; // read input FB
   }
   if (C_FB) N = FB; else N = N - 1;
   w_DEC_N(N); // write output N
   DEC_step_finalize();
   return TRUE;
}
```

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 (line (6)). In lines (2) and (6), the references to the variable N^c are the references to the last value of N^c 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`.

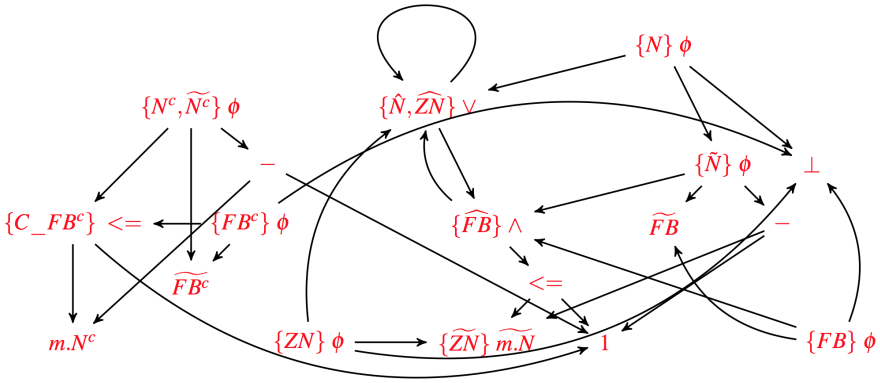

Fig. 5: The shared value-graph of DEC and DEC_step

In the second step, we shall normalize the above initial graph. Below is a potential normalization scenario, meaning that there might have more than one normalization scenario, and the validator can choose one of them. For example, given a set of rules that can be applied, the validator can apply these rules with different order. Fig. 6 depicts the intermediate resulting graph of this normalization scenario, and Fig. 7 is the final normalized 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 by the variable N^c being always updated. The node $\{\hat{N}, \widehat{ZN}\}\ \vee$ is rewritten into `true`.
- By rule $\wedge(\texttt{true}, x) \rightarrow x$, the node $\{\widehat{FB}\}\ \wedge$ is rewritten into $\{\widehat{FB}\}$ <=.
- The ϕ-function node representing the computation of N is removed and N points to the node $\{\widetilde{N}\}\ \phi$.
- The ϕ-function node representing the computation of ZN is removed and ZN points to the node $\{\widetilde{ZN}\}\ \widetilde{m.N}$.
- The nodes $\widetilde{FB^c}$ and $\widetilde{FB}$ are rewritten into a single node $\{\widetilde{FB}\}\ \widetilde{FB^c}$. All references to them are replaced by references to $\{\widetilde{FB}\}\ \widetilde{FB^c}$.
- The nodes $m.N^c$ and $\widetilde{m.N}$ are rewritten into a single node $\{\widetilde{m.N}\}\ m.N^c$. All references to them are replaced by references to $\{\widetilde{m.N}\}\ m.N^c$.

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 are equivalent since they point to the same node in the final normalized graph.

5 Conclusion and Related Work

There is a wide range of works for value-graph representations of expression evaluations in a program. For example, in [16], Weise et al. present a nice summary of the various types of value-graph. In our context, the value-graph is used to

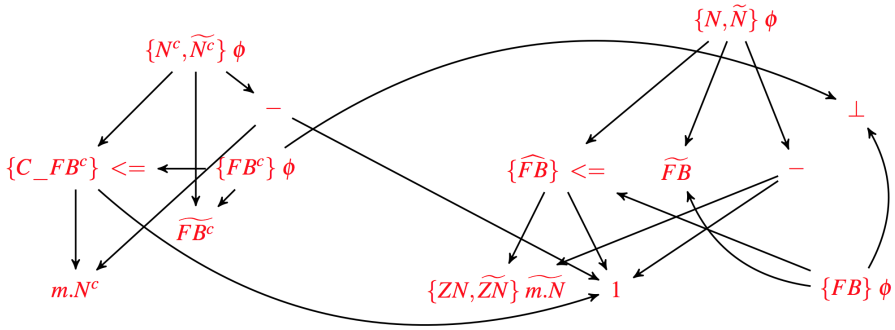

Fig. 6: The resulting value-graph of DEC and DEC_step

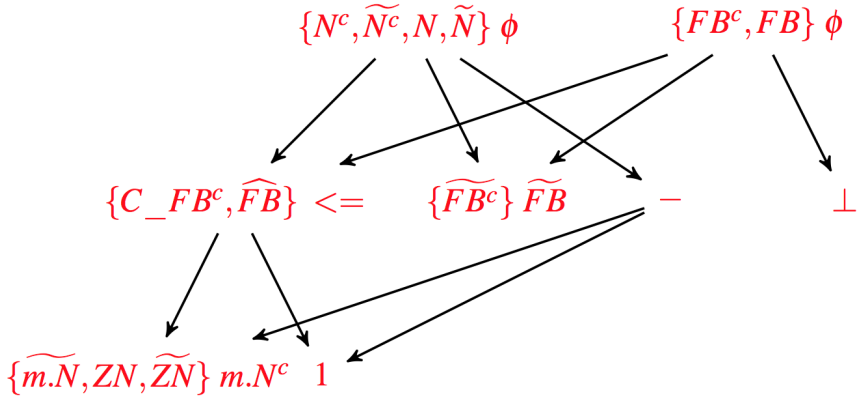

Fig. 7: The final normalized graph of DEC and DEC_step

represent the computation of variables in both source program and its generated C code in which the identical structures are shared. We believe that this representation will reduce the required storage and make the normalizing process more efficient. Another remark is that the calculation of clocks as well as the special value, the absent value, are also represented in the shared 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 construct a shared value-graph. The graph is *normalized* (the graph is reduced). After the normalization, if the outputs of two functions are represented by the same sub-graph, they can safely conclude that both 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. Then they transform the graph by adding equivalent terms through a process called *equality saturation*. After the saturation, if both value-graphs are the same, they can conclude that the return

value of two given functions are the same. However, for translation validation purpose, our normalization process is more efficient and scalable since we can add rewrite rules into the validator that reflect 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 three nodes subgraph $(1+2)$ is replaced by a single node 3).

The present paper provides a verification framework to prove the value-equivalence of variables and applies this approach to the synchronous data-flow compiler SIGNAL. With the simplicity of the graph normalization, we believe that translation validation of synchronous data-flow value-graph for the industrial compiler SIGNAL is feasible and efficient. Moreover, the normalization process can always be extended by adding new rewrite rules. That makes the translation validation of SDVG scalable and flexible.

We have considered sequential code generation. A possibility is to extend this framework to use with other code generation schemes including cluster code with static and dynamic scheduling, modular code, and distributed code. One path forward is the combination of this work and the work on data dependency graph in [9,10,11]. That means that we use synchronous data-flow dependency graphs and synchronous data-flow value-graphs as a common semantic framework to represent the semantics of the generated code. The formalization of the notion of "correct transformation" is defined as the refinements between two synchronous data-flow dependency graphs and in a shared value-graph as described above.

Another possibility is that we use a SMT solver to reason on the rewriting rules. For example, we recall the following rules:

$$\phi(c_1, \phi(c_2, x_1, x_2), x_3) \rightarrow \phi(c_1, x_1, x_3) \text{ if } c_1 \Rightarrow c_2$$
$$\phi(c_1, \phi(c_2, x_1, x_2), x_3) \rightarrow \phi(c_1, x_2, x_3) \text{ if } c_1 \Rightarrow \neg c_2$$

To apply these rules on a shared value-graph to reduce the nested ϕ-functions (e.g., from $\phi(c_1, \phi(c_2, x_1, x_2), x_3)$ to $\phi(c_1, x_1, x_3)$), we have to check the validity of first-order logic formulas, for instance, we check that $\models (c_1 \Rightarrow c_2)$ and $\models c_1 \Rightarrow \neg c_2$. We consider the use of SMT to solve the validity of the conditions as in the above rewrite rules to normalize value-graphs.

References

1. P. Aubry,P. Le Guernic and S. Machard. *Synchronous Distribution of Signal Programs*. In Proceedings of the 29th Hawaii International Conference on System Sciences, IEEE Computer Society Press. (1)656-665, 1996.
2. S. Blazy. *Which C Semantics to Embed in the Front-end of a Formally Verified Compiler?*. Tools and Techniques for Verification of System Infrastructure, TTVSI. 2008.
3. R. Ballance, A. Maccabe and K. Ottenstei, *The Program Dependence Web: A Representation Supporting Control, Data, and Demand Driven Interpretation of Imperative Languages*. In Proc. of the SIGPLAN'90 Conference on Programming Language Design and Implementation. 257-271, 1990.

4. A. Benveniste and P. Le Guernic, *Hybrid dynamical systems theory and the Signal language*. IEEE Transactions on Automatic Control. 35(5):535-546, May 1990.
5. L. Besnard, T. Gautier, P. Le Guernic and J-P. Talpin, *Compilation of Polychronous Data-flow Equations*. In Synthesis of Embedded Software Springer. 1-40, 2010.
6. T. Gautier, P. Le Guernic and L. Besnard, *Signal, a declarative language for synchronous programming of real-time systems*. Proc. 3rd. Conf. on Functional Programming Languages and Computer Architecture, LNCS 274, May 1990.
7. T. Gautier and P. Le Guernic. *Code Generation in the SACRES Project*. In Towards System Safety, Proceedings of the Safety-critical Systems Symposium. 127-149, 1999.
8. O. Maffeïs and P. Le 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. (863)547-566, 1994.
9. V.C. Ngo, J-P. Talpin, T. Gautier, P. Le Guernic and L. Besnard. *Formal Verification of Compiler Transformations on Polychronous Equations*. In Proceedings of 9th International Conference on Integrated Formal Methods IFM 2012, Springer Lecture Notes in Computer Science. 2012.
10. V.C. Ngo, J-P. Talpin, T. Gautier, P. Le Guernic and L. Besnard. *Formal Verification of Synchronous Data-flow Program Transformations Toward Certified Compilers*. In Frontiers of Computer Science, Special Issue on Synchronous Programming, Springer. 2013.
11. V.C. Ngo. *Formal Verification of a Synchronous Data-flow Compiler: from Signal to C*. Ph.D Thesis - http://tel.archives-ouvertes.fr/tel-01058041. 2014.
12. A. Pnueli, M. Siegel and E. Singerman. *Translation Validation*. In B. Steffen, editor, 4th Intl. Conf. TACAS'98. NCS 1384:151-166, 1998.
13. A. Pnueli, O. Shtrichman and M. Siegel. *Translation validation: From Signal to C*. In Correct Sytem Design Recent Insights and Advances. LNCS 1710:231-255, 2000.
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. 264-276, 2009.
16. D. Weise, R.F. Crew, M.D. Ernst and B. Steensgaard, *Value Dependence Graphs: Representation without Taxation*. In 21th Principles of Programming Languages. 297-310, 1994.

Appendix A: The Signal Language

5.1 Language Features

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`.

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 $\forall t \in C_x \cap C_b \wedge b(t) =$ `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 $\{t \in C_b | b(t) =$ `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).

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.

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:

```
process DEC=
(? integer FB;
 ! integer N)
 (| FB ^= when (ZN<=1)
  | N := FB default (ZN-1)
  | ZN := N$1 init 1
  |)
where integer ZN init 1
end;
```

Let us comment this program: ? `integer` FB;! `integer` N: FB, N are respectively input and output signals of type *integer*; $FB \hat{} =$ `when` $(ZN <= 1)$: FB is accepted (or it is present) only when ZN becomes less than or equal to 1; $N := FB$ `default` $(ZN - 1)$: N is set to FB when its previous value is less than or equal to 1, otherwise it is decremented by 1; $ZN := N\$1$ `init` 1: defines ZN as always carrying the previous value of N (the initial value of ZN is 1); `where integer` ZN `init` 1: 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
C_{FB}	t_0						t_6			t_9
C_{ZN}	t_0	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9
C_N	t_0	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9

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;
```

Appendix B: Rewrite Rules

General Simplification Rules

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

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 $b = x \neq \texttt{true}$). 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 subsets as follows.

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

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.

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

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

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

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 remove the ϕ-function in the branch if one of the following conditions is 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 its 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 its negation (as in the rules 24 to 27).

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

```
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 shall 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 $\phi(=(a,b),d,0)$. Replacing the definition of a, b and d, and normalizing this graph, we get:

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

Optimization-specific Rules Based on the optimizations of the SIGNAL compiler, we have a number of *optimization-specific rules* in a way that reflexes the effects of specific optimizations of the compiler. These rules do not always reduce the graph or make it simpler. One has to know specific optimizations of the compiler when she wants to add them to the set of rewrite rules. In our case, the set of rules for simplifying constant expressions of the SIGNAL compiler is 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:

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

– Specific rules for constant expressions with numerical comparison functions:

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

where $\square = <, >, =, <=, >=, /=$, and the Boolean value cst is the evaluation of the constant expression $\square(cst_1, cst_2)$ 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 inference* for propositional logic. For example, we have a group of laws for rewriting formulas with `and` operator, such as:

$$\wedge(x, \texttt{false}) \rightarrow \texttt{false}$$
$$\wedge(x, \texttt{true}) \rightarrow x$$
$$\wedge(x, \Rightarrow (x, y)) \rightarrow x \wedge y$$

We consider the following SIGNAL program and its generated C code, the input signal x is present when the other Boolean input signal cx holds the value `true`.

```
/* Signal equation */
| x ^= when cx
/* Generated C code */
if (C_cx)
{
   if (cx)
   {
      if (!r_P_x(&x)) return FALSE;
   }
}
```

The computation of x is represented by $x = \phi(\hat{x}, \widetilde{x}, \perp)$, where $\hat{x} \Leftrightarrow \widehat{cx} \wedge \widetilde{cx}$. 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 $x = \phi(C\_cx, \phi(cx, \widetilde{x}, \perp), \perp)$. This observation makes us add the following rewrite rule into the systems to mirror the above rewriting of the SIGNAL compiler.

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

Synchronous Rules 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. Hence, we have the following rule that mirrors the above situation.

$$x^c \mapsto \phi(\texttt{true}, \widetilde{x^c}, \perp) \rightarrow x \mapsto \phi(\texttt{true}, \tilde{x}, \perp) \tag{38}$$

We write $x \mapsto \phi(\texttt{true}, \tilde{x}, \perp)$ to denote that x points to the subgraph rooted at the node labeled by ϕ-function. The rule 38 indicates that if a variable in the generated C code is always updated, then we require that the corresponding signal in the source program is present at every instant, 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 $\tilde{x}$ (resp. the variable x^c and its updated value $\widetilde{x^c}$ in the generated C code) point to the same node in the shared value-graph. Every reference to x and $\tilde{x}$ (resp. x^c and $\widetilde{x^c}$) point to the same node. A second synchronous rule mirrors the semantics of the *delay* operator. For instance, we

consider the equation $pz := z\$1$ `init` 0. We use the variable $\widetilde{m.z}$ to capture the last value of the signal z. In the generated C program, 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 $\widetilde{m.z} = m.z^c$.

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

The rule 39 indicates that for any signal x which is involved in a *delay* operator, and its corresponding variable in the generated C program, then it is required that the last values of x and x^c are the same. Therefore, every reference to $m.x^c$ and $\widetilde{m.x}$ points to the same node. Finally, we add rules that mirror the relation between input signals and their corresponding variables in the generated C code. First, 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 and the value of the variable x^c after the reading statement must be equivalent. That means $\widetilde{x^c}$ and $\tilde{x}$ are represented by the same subgraph in the graph. Second, if the clock of x is also read from the environment as a parameter, then the clock of the input signal x is equivalent to the condition in which the variable x^c is updated. It means that we represent $\hat{x}$ and C_x^c by the same subgraph.

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

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

Consequently, every reference to $\hat{x}$ and C_x^c (resp. $\tilde{x}$ and $\widetilde{x^c}$) points to the same node.