**Noname manuscript No.**
(will be inserted by the editor)

# Modular Translation Validation of a Full-sized Synchronous Compiler using Off-the-shelf Verification Tools

**Van Chan Ngo · Jean-Pierre Talpin · Thierry Gautier · Loïc Besnard · Paul Le Guernic**

Received: date / Accepted: date

**Abstract** This work demonstrates a scalable, modular, and refinable methodology for translation validation applied to a mature (20 years old), large (500K lines of C), and open source (Eclipse/Polarsys IWG project POP) code generation suite, all by using off-the-shelf and open-source SAT/SMT verification tools (Yices), by adapting and optimizing the translation validation principle introduced by Pnueli et al. in 1998. In face of the enormous task at hand, the verification of a large compiler infrastructure, we devised to narrow down and isolate the problem to the very data-structures manipulated by the infrastructure at the successive phases of the compilation, in order to both optimize the whole verification process and make the implementation of a working prototype at all doable. Our presentation outlines the successive steps of this endeavour, from clock synthesis, static scheduling to target code production.

**Keywords** Formal Verification · Translation Validation · Certified Compiler · SAT/SMT Solver · Synchronous Languages · Graph Transformation

## 1 Introduction

Synchronous languages [6,28,35] offer formal semantic frameworks to design safety-critical systems, from their formal specification to their implementation as automatically generated code. Safety-critical systems are those systems whose reliability is crucial. Domains of applications include avionics and automotive systems, medical technology, telecommunication and control of industrial plants. In such systems, the violation of some constraints may lead to serious consequences, including loss of mission, environmental damage or even loss of life. Thus it is an essential task in the

Van Chan Ngo · Jean-Pierre Talpin · Thierry Gautier · Paul Le Guernic
INRIA, 35042 Rennes, France
E-mail: first.lastname@inria.fr

Loïc Besnard
CNRS/IRISA, 35042 Rennes, France
E-mail: loic.besnard@irisa.fr

development of safety-critical systems to be able to *prove* their reliability [31]. Since synchronous languages are based on formal semantic models, they provide much higher level of abstraction, expressivity, and clarity at source level rather than once compiled into C code for instance. That makes the application of formal methods much simpler to enforce safety properties. But there is a clear need that such safety properties, specified at source level, are ensured to be preserved through implementation. However, compilers of synchronous languages are large and complex programs which often consist of hundreds of thousands lines of code, divided into numerous packages. Moreover, compiler modules often interact in sophisticated ways, and the design and implementation of a compiler are substantial engineering tasks. Compilation involves analyzes, transformations, optimizations, some introducing new information, some refining or specializing the program's behavior to meet safety goals.

In this work, we demonstrate a *translation validation* approach, applied to a full-sized compiler of the polychronous data-flow language Signal. This compiler is a mature (20 years old), large (500k lines of C and C++), and open source (Eclipse/Polarsys IWG project POP) code generation suite for the Signal language[1]. Translation validation was introduced by Pnueli et al. in [51] as an approach to verify the correctness of translators (compilers, code generators). The main idea of translation validation is that instead of proving the correctness of a compiler, each of its individual translations (e.g., run of the code generator) is followed by a validation phase to check that the target program correctly implements the source specification. A translation validator consists of two objects:

- *The Model Builder* is a simple module that formally represents the semantics of the source and target programs of the translator (e.g., as labeled transition system or first-order logic formula).
- *The Analyzer* formalizes correctness as a refinement relation between the models of the source and target of the validator. The analyzer provides an automated proof method on the existence of the refinement between the formal models. If the analyzer succeeds, a *proof script* is created. If it does not, it generates a *counterexample*, which can be decompiled to help spot the error.

Translation validation does not modify or instrument the compiler. It treats it as a "black box". It only considers the input program and its compiled result. Hence, it is not affected by updates and modifications to the compiler, as long as its data structures remain the same. In general, the validator is much simpler and smaller than the compiler itself. Thus, the proof of correctness of the validator takes less effort than that of the compiler. Furthermore, verification is fully automated and scales to large programs.

The remainder of this article is organized as follows. Section 2 presents some related works about formal verification of compilers, including translation validation. Section 3 introduces the principles of our modular translation validation approach, applied to the Polychrony compiler of the Signal language. Then, Sections 4, 5 and 6 detail the three phases of this translation validation method, corresponding to the three main phases of the compiler. These are respectively preservation of

[1] `http://www.irisa.fr/Polychrony`

clock models, preservation of data dependencies and value-equivalence of variables. All these translation validation phases are illustrated using a simple representative Signal program. A few bugs have been detected in the Signal compiler thanks to the application of this method; they are described in Section 7. Finally, Section 8 concludes our work and outlines future directions.

## 2 Related Work

Formal verification of the correctness of a compiler can be based on the examination of the developed compiler's source code itself, meaning that a qualification process applies on the development of the compiler, the source of the compiler, and/or the compiler's output. Qualifying a compiler is rare because of the tremendous administrative effort involved. Qualification amounts to demonstrate the compliance with all recommendations and objectives specified in the certification standards for safety-critical software: DO-178C and its European equivalence ED-12. This provides users a way to demonstrate that the properties established in the source code still hold in the object code. In order to fulfil this objective, DO-178C recommends a proof that requirements at source-code level can be traceable down to the object code [36, 15], including the integration of software onto its hardware execution platform. Although DO-178C has been successful in industry, the cost of complying with it is significant: the activities on verification it incurs may well cost seven times more than the development effort needed [56].

A more traditional method is therefore to solely inspect or formally verify the compiler's output. This task requires less unitary effort, but has to be repeated every time target code is generated. For instance, Astrée [3, 10] is a special-purpose static program analyzer based on abstract interpretation to verify the absence of *run time errors* in the C code generated from SCADE programs. Another example, the SuperTest suite [1], is one of the most comprehensive test and validation suites to verify compilers. It contains a large collection of more than 3 millions test programs. The tool interprets the test set definition and test run parameters. Then, it feeds the tests to the compiler. It might run the resulting programs to assert that the compiler passed by the test.

One last resort is to formally verify the correctness of the compiler itself. There are two approaches, in general, to prove the correctness of a compiler using formal methods. One approach consists of specifying the intended behavior of the compiler in a specification language as a formal model and of building a proof to show that the compiler behaves exactly as prescribed by requirements. The second approach consists of examining the source and compiled programs in order to prove that, for each run of the compiler, the semantics of the source program is preserved. Many correctness proofs of compiler implementations based on the two above approaches have been carried out, formal verification of the compiler itself [59, 53, 16, 13] or the verification of its compiled code [51, 50, 55, 43, 62, 52, 33, 37].

A recent and typical example of compiler correctness proof is that of [13]. In this example, the correctness of the whole *Iterated Register Coalescing* (IRC) algorithm [27] is formally verified. The verification process works in cooperation with the proof assistant Coq [18]. The input of IRC is an *interference graph* and a *palette* of colors, the output is the colored graph. The interference graph is first defined in a purely functional language, Gallina, and implemented in the Coq prover. Then IRC is written in Gallina. The implementation of the abstract interference graph and the operations of the algorithm are formally proved to be correct. The verified program is translated automatically into OCaml code that can be plugged in the CompCert compiler to provide correct register allocation.

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. This brings two main drawbacks of the formal verification of the compiler itself approach. 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*. A pioneering contribution to this area was the work of Pnueli et al. [51,50] to prove the correctness of the code generator from Signal programs to C programs. Pnueli et al. formalized the semantics of a Signal program and the generated C code in terms of *Synchronous Transition Systems* (STS). A STS consists of the set of states, the set of initial sates and a transition relation. The running of a program is represented by a *computation* of STS which is an infinite sequence of states $\sigma = \langle s_0, s_1, s_2, \ldots \rangle$ such that $s_0$ is an initial state and $s_{i+1}$ is the successor state of $s_i$, for all $i \in \mathbb{N}$. The set of all possible computations represents the semantics of the program. Given a computation $\sigma$, an *observation* is an infinite sequence of values by applying the *observation function* on each state of $\sigma$. Then, the authors formalize the concept of “correct translation” as a refinement between two STSs which expresses that the semantics of the source program is preserved at the compiled program, meaning that for any observation of the STS of the compiled program, it is also the observation of the STS of the source program. The refinement is generated as a set of verification conditions, and it is proved by the use of a solver such as SMT solver. Technically, the drawback of this approach is that it does not capture explicitly the clock semantics and in some cases, the code generator eliminates the use of local register variables in the generated code and then, the mapping cannot be established. Additionally, for a large program, the formula is very large, including numerical expressions that induce some inefficiency. Moreover, the whole calculation of a synchronous program or the generated code is considered as one atomic transition in STS, thus it does not capture the data dependencies between signals.

Zuck et al. [63,62,41,51] introduce a methodology to validate optimizations by generating a set of verification conditions and using a theorem prover. The main idea of their work is that the validator generates a set of verification conditions based on an invariant for intra-procedural optimizations. The invariant for an intra-procedural optimization is composed of:

– a relation between the nodes in the control-flow graphs;
– a relation between the program states (e.g., contents of registers, stacks...);

– invariants for the individual input and output programs.

This set of verification conditions indicates the program equivalence for finite slices of program executions. That implies that the optimized program is a correct refinement of the input program.

A representative example is the CompCert project [17]. The CompCert compiler is a formally verified compiler for the C language. The compiler is mostly written in the functional programming language Gallina. The implementation is formally verified and automatically translated into OCaml code by Coq. Some representative works of the project are carried out by Blazy et al. [12,11,21] and Leroy et al. [40, 38,39]. For instance, the work of Leroy [39] describes the correctness proof of the code generation, the back-end of the CompCert compiler, from a low-level, imperative intermediate language C-minor into optimized PowerPC assembly code, using the Coq proof assistant.

Inspired by the work of the CompCert compiler, the formal development of a code generator based on a correct-by-construction components method is carried out in the GeneAuto project [26,30,29]. The GeneAuto code generator takes as input a functional description of a system specified in a high-level modeling language (e.g., Simulink, Stateflow) and generates C code as output.

Another related work which adopts the translation validation approach in verification of optimizations, Tristan et al. [61,60], recently proposed a framework for translation validation of an LLVM optimizer. For a function and its optimized counterpart, they construct a shared value-graph. The graph is *normalized* (it is reduced). After the normalization, if the outputs of two functions are represented by the same subgraph, they can safely conclude that both functions are equivalent.

On the other hand, Tate et al. [58] propose 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. For translation validation purpose, our normalization process in Section 6 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., if a compiler performs constant folding optimization, then we can add rewrite rules for constant expressions such that a three nodes subgraph $(1 + 2)$ is replaced by a single node 3).

Gamatié et al. [23,24,20] introduce an approach to statically analyze Signal programs for efficient code generation. The main idea of their work is that the clocks and clock relations are formalized as first-order logic formulas with the help of interval-Boolean abstraction technique. This work aims to remove the dead-code segments (e.g., segment of code to compute a data-flow which is always absent). The dead-code segments are identified by detecting the existence of empty clocks, mutual exclusion of two or more clocks, or clock inclusions. The reasoning on the logic formulas is done using a SMT solver. With the interval abstraction, the analysis of clock hierarchy is more precise and more efficient when dealing with numerical expressions.

## 3 Modular Translation Validation

Our approach is to scale translation validation not only in a modular fashion, by decomposing the problem into the successive transformations performed by the compiler on the intermediate representation of a program [47], but by narrowing it further to the actual data structure that is being used to represent the transformation problem and the actual algorithmic operations performed on it. In all cases, we show that translation validation is amenable to simple SAT/SMT verification (the semantic inclusion of one data structure into another) instead of the more general problem of simulation-based conformance-checking of the transformed program w.r.t. the input program [44,46, 45].

### 3.1 The Signal Compiler

Historically related to the synchronous programming paradigm, the polychronous model of computation, implemented in the data-flow language Signal and its infrastructure Polychrony, stands apart by the capability to model multi-clocked systems. A source Signal program is the synchronous composition of discrete equations on signals, e.g. $x := y + 1 \mid y := x\$1$ defines $x$ by $y + 1$ at all times (discrete logical instants) and $y$ by the value of $x$ delayed by 1 evaluation tick. Its compilation may be seen as a sequence of morphisms that refine and rewrite the source specification with information gained from analysis. C or Java code production, e.g., $y = x;\ x = y + 1$, is performed on ultimately transformed program.

We briefly recall here the primitive operators of the core language:
1/ *Stepwise Functions* on flows, $y := f(x_1, \ldots, x_n)$ for $f$ denoting a function on values ($y, x_1, \ldots, x_n$ are defined at the same logical instants: they have the same *clock*);
2/ *Delay* $y := x\$1$ `init` $a$ (where $a$ is a constant value): $y$ and $x$ are defined at the same logical instants and at any of these instants, $y$ holds the previous value of $x$;
3/ *Sampling* $y := x$ `when` $b$ ($b$ is a Boolean signal): $y$ holds the value of $x$ when $x$ is present and $b$ is present and `true`;
4/ *Merge* $y := x$ `default` $z$: $y$ holds the value of $x$ when $x$ is present, otherwise it holds the value of $z$ when $z$ is present.
The following notations (which are derived operators) are used to manipulate clocks, represented as signals of type `event`, `true` if and only if present: *clock* $z :=\hat{}x$ (present, and `true`, when $x$ is present); *synchronization* $x\ \hat{}=\ y$; *clock union* $z := x\ \hat{}+\ y$; *clock product* $z := x\ \hat{}*\ y$; *clock difference* $z := x\ \hat{}-\ y$; *clock selection* $z :=$ `when` $b$ (present, and `true`, when the Boolean signal $b$ is present and `true`).

A *process* is a system of equations. The *composition* of two processes $P$ and $Q$ is a process written $P \mid Q$, defined by the set of behaviors that satisfy both $P$ and $Q$ constraints. The *restriction* $P$ `where` $x$, where $x$ is a signal defined in a process $P$, is a process in which $x$ is considered as a local variable in $P$. The reader is referred to the Signal bibliography [7,22] for a more detailed description. The Signal compiler provided by Polychrony is an open source 500k lines big software. Schematically, the compilation process [5,8,42] can be divided into three phases depicted in Fig. 1 (top row).

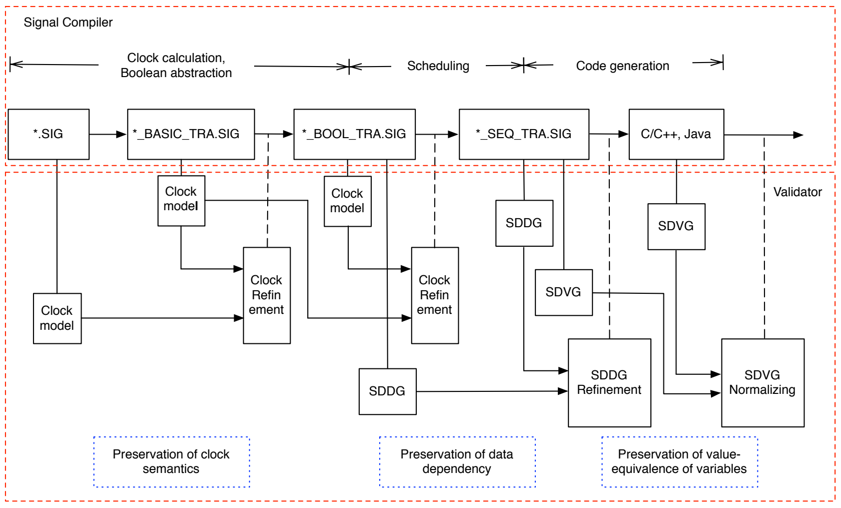

Fig. 1 Translation validation for the Signal compiler

– *Clock synthesis*. This stage determines the *clock* of all signals in the program and defines a Boolean abstraction of this program. The clock of a signal defines the instants at which the value of the signal shall be evaluated.
– *Static scheduling synthesis*. Based on the clock information and the Boolean abstraction obtained at the first stage, the compiler constructs a *Conditional Dependency Graph* (CDG), which represents the schedule of signals' evaluations.
– *Code generation*. Sequential C or Java code is directly generated from the structure of the clocked and scheduled Signal program.

## 3.2 Towards a Formally Verified Signal Compiler

Translation validation is decomposed into the proof of clock semantics preservation (*clock synthesis* phase) and that of data dependencies (*scheduling synthesis* phase). Value-equivalence of variables can then be checked at the *code generation* phase. Fig. 1 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, and constructs the corresponding formal model of the programs. Then, it checks the existence of the refinement relation to prove the preservation of the considered semantics. If the result is that this relation does not exist then a "compiler bug" message is emitted. Otherwise, the compiler continues its work.

Given a source program A, a compilation phase performed by the Signal compiler can be considered as a function $Cp^{sig}$ from a set of source programs to a set of compiled programs plus compilation error: $P_s \longrightarrow P_c \cup \{\texttt{Error}\}$. We denote the compiled program of A by $Cp^{sig}(\texttt{A}) = \texttt{C}$ and the compilation error by $Cp^{sig}(\texttt{A}) = \texttt{Error}$.

Consider a validator $Val$ which adopts the translation validation approach. The validator can be represented as a function from the set of pairs of a source program and its compiled program to the set of Boolean values: $P_s \times P_c \longrightarrow \mathbb{B}$. The validator that we want to build satisfies the following property:

$$\forall \mathtt{A} \in P_s, C \in P_c, Cp(\mathtt{A}) = \mathtt{C}, Val(\mathtt{A}, \mathtt{C}) = \mathtt{true} \Rightarrow Correct(\mathtt{A}, \mathtt{C})$$

where $Correct(\mathtt{A}, \mathtt{C})$ denotes the correctness property between the source program $\mathtt{A}$ and its compiled program $\mathtt{C}$. We now associate each run of the compiler, $Cp^{sig}$, with the validator $Val$. The function $Cp^{sig}_{Val}$ defines a formally verified compilation process from $P_s$ to $P_c$ plus compilation error. The derived Signal compiler, associated with the validator in Fig. 1, can be defined by the following function from $P_s$ to $P_c \cup \{\mathtt{Error}\}$.

$$Cp^{sig}_{Val}(\mathtt{A}) = \begin{cases} \mathtt{C} & \text{if } Cp^{sig}(\mathtt{A}) = \mathtt{C}_{clk}, Cp^{sig}(\mathtt{C}_{clk}) = \mathtt{C}_{dep}, Cp^{sig}(\mathtt{C}_{dep}) = \mathtt{C} \text{ and} \\ & Val(\mathtt{A}, \mathtt{C}) = \mathtt{true} \\ \mathtt{Error} & \text{if } Cp^{sig}(\mathtt{A}) = \mathtt{C}_{clk}, Cp^{sig}(\mathtt{C}_{clk}) = \mathtt{C}_{dep}, Cp^{sig}(\mathtt{C}_{dep}) = \mathtt{C} \text{ and} \\ & Val(\mathtt{A}, \mathtt{C}) = \mathtt{false} \\ \mathtt{Error} & \text{if } Cp^{sig}(\mathtt{A}) = \mathtt{Error} \text{ or } Cp^{sig}(\mathtt{C}_{clk}) = \mathtt{Error} \text{ or} \\ & Cp^{sig}(\mathtt{C}_{dep}) = \mathtt{Error} \end{cases}$$

$\mathtt{C}_{clk}$ and $\mathtt{C}_{dep}$ are the intermediate forms of the source program $\mathtt{A}$ as the outputs of the *clock synthesis* phase and *scheduling synthesis* phase, respectively. They have an explicit textual representation as Signal programs (with added synthesized definitions of clocks and explicit dependencies). $\mathtt{C}$ is the generated C program from the intermediate form $\mathtt{C}_{dep}$. $Val(\mathtt{A}, \mathtt{C}) = \mathtt{true}$ if and only if $Val(\mathtt{A}, \mathtt{C}_{clk}) = \mathtt{true}$, $Val(\mathtt{C}_{clk}, \mathtt{C}_{dep}) = \mathtt{true}$, and $Val(\mathtt{C}_{dep}, \mathtt{C}) = \mathtt{true}$.

## 4 Preservation of Clock Models

This section focuses on constructing a validator that proves the preservation of clock semantics in the *clock synthesis* phase of the Signal compiler. The clock semantics of the source program and its transformed counterpart are formally represented as *clock models*. A clock model is a first-order logic formula with *uninterpreted functions*. This formula deterministically characterizes the presence/absence status of all discrete data-flows (input, output and local variables of a program) at any logical instant. Given two clock models, a *correct transformation* between them is checked by the existence of a refinement relation, which expresses preservation of clock semantics.

### 4.1 Clock Model

In Signal, clocks play a much more important role than in other synchronous languages: they are used to express the underlying control (i.e., synchronization between signals). This differs in particular from Lustre, where all clocks are built by sampling the fastest clock. Consider the basic process $y := x$ `when` $b$ defined by the primitive operator *sampling*, where $x$ and $y$ are numerical signals, for instance, and $b$ is a Boolean one. To express the control, we need to represent the status of the signals $x$, $y$ and $b$. We use a Boolean variable $\hat{x}$ to capture the status of $x$: ($\hat{x} = \texttt{true}$) means $x$ is present, and ($\hat{x} = \texttt{false}$) means $x$ is absent. In the same way, the Boolean variable $\hat{y}$ captures the status of $y$. For the Boolean signal $b$, two Boolean variables, $\hat{b}$ and $\widetilde{b}$, will be used to represent its status: ($\hat{b} = \texttt{true} \wedge \widetilde{b} = \texttt{true}$) means $b$ is present and holds a value $\texttt{true}$; ($\hat{b} = \texttt{true} \wedge \widetilde{b} = \texttt{false}$) means $b$ is present and holds a value $\texttt{false}$; and ($\hat{b} = \texttt{false}$) means $b$ is absent. Hence, at any given instant, the clock relations of the equation above can be encoded by the formula: $\hat{y} \Leftrightarrow (\hat{x} \wedge \hat{b} \wedge \widetilde{b})$.

### 4.1.1 Abstraction

Let $X = \{x_1, \ldots, x_n\}$ be the set of all signals in a program $\mathtt{P}$, consisting of input, output, register (corresponding to *delay* operator), and local signals, denoted by $I$, $O$, $R$ and $L$, respectively. With each signal $x_i$, based on the encoding scheme proposed in [24], we attach a Boolean variable $\widehat{x_i}$ to encode its clock and a variable $\widetilde{x_i}$ of same type as $x_i$ to encode its value. The composition of processes corresponds to logical conjunctions. Thus the clock model of $\mathtt{P}$ will be a conjunction $\Phi(\mathtt{P}) = \bigwedge_{i=1}^{n} \phi(eq_i)$, whose atoms are $\widehat{x_i}$, $\widetilde{x_i}$, where $\phi(eq_i)$ is the abstraction of statement $eq_i$, and $n$ is the number of statements in the program. In the following, we present the abstraction corresponding to each Signal operator.

*Stepwise Functions.* The functions which apply on signal values in the stepwise functions are usual logic operators ($\texttt{not}$, $\texttt{and}$, $\texttt{or}$), numerical comparison functions ($<$, $>$, $=$, $<=$, $>=$, $/=$), and numerical operators ($+$, $-$, $*$, $/$). In our experience working with the Signal compiler, it performs very few arithmetical optimizations and leaves most of the arithmetical expressions intact. Moreover, for Signal programs from which deterministic code can be generated, every signal definition is determined explicitly by the input and register signals. This suggests that numerical expressions can be abstracted by *uninterpreted functions* [50,24]. By following the encoding procedure of [2], for every numerical comparison function and numerical operator (denoted by $\Box$) occurring in an equation, we perform the following rewriting: i) Replace each $x \Box y$ by a new variable $v^i_\Box$ of the same type as the return value. Two stepwise functions $x \Box y$ and $x' \Box y'$ are replaced by the same variable $v^i_\Box$ iff $x, y$ are identical to $x'$ and $y'$, respectively; ii) For every pair of newly added variables $v^i_\Box$ and $v^j_\Box$, $i \neq j$, corresponding to the non-identical occurrences $x \Box y$ and $x' \Box y'$, add the implication $(\widetilde{x} = \widetilde{x'} \wedge \widetilde{y} = \widetilde{y'}) \Rightarrow \widetilde{v^i_\Box} = \widetilde{v^j_\Box}$ into the abstraction $\Phi(\mathtt{P})$. The abstraction $\phi(y := f(x_1, ..., x_n))$ of stepwise functions is defined by induction as follows: $\phi(\texttt{true}) = \texttt{true}$ and $\phi(\texttt{false}) = \texttt{false}$; $\phi(y := x) = (\hat{y} \Leftrightarrow \hat{x}) \wedge (\hat{y} \Rightarrow (\widetilde{y} = \widetilde{x}))$; if $x$ is an $\texttt{event}$ signal, $\phi(y := x) = (\hat{y} \Leftrightarrow \hat{x}) \wedge (\hat{y} \Rightarrow (\widetilde{y} = \widetilde{x})) \wedge (\hat{x} \Rightarrow \widetilde{x})$; $\phi(y := \texttt{not}\ x) = (\hat{y} \Leftrightarrow \hat{x}) \wedge (\hat{y} \Rightarrow (\widetilde{y} \Leftrightarrow \neg\widetilde{x}))$; $\phi(y := x_1\ \texttt{and}\ x_2) = (\hat{y} \Leftrightarrow \widehat{x_1} \Leftrightarrow \widehat{x_2}) \wedge (\hat{y} \Rightarrow (\widetilde{y} \Leftrightarrow \widetilde{x_1} \wedge \widetilde{x_2}))$; $\phi(y := x_1\ \texttt{or}\ x_2) = (\hat{y} \Leftrightarrow \widehat{x_1} \Leftrightarrow \widehat{x_2}) \wedge (\hat{y} \Rightarrow (\widetilde{y} \Leftrightarrow \widetilde{x_1} \vee \widetilde{x_2}))$; $\phi(y := x_1 \Box x_2) = (\hat{y} \Leftrightarrow \widehat{v^i_\Box} \Leftrightarrow \widehat{x_1} \Leftrightarrow \widehat{x_2}) \wedge (\hat{y} \Rightarrow (\widetilde{y} = \widetilde{v^i_\Box}))$.

*Delay.* Considering the *delay* operator, $y := x\$1\ \texttt{init}\ a$, its encoding $\phi(y := x\$1\ \texttt{init}\ a)$ contributes to $\Phi(\mathtt{P})$ with the following conjunct: $(\hat{y} \Leftrightarrow \hat{x}) \wedge (\hat{y} \Rightarrow ((\widetilde{y} = m.x) \wedge (m.x' = \widetilde{x}))) \wedge (m.x_0 = a)$. This encoding requires that at any instant,

signals $x$ and $y$ have the same status (present or absent). To encode the value of the output signal as well, we introduce a memorization variable $m.x$ that stores the last value of $x$. The next value of $m.x$ is $m.x'$ and it is initialized to $a$ in $m.x_0$.
*Sampling.* The encoding of the *sampling* operator, $y := x$ `when` $b$, contributes to $\Phi(\mathtt{P})$ with the following conjunct: $(\hat{y} \Leftrightarrow (\hat{x} \wedge \hat{b} \wedge \tilde{b})) \wedge (\hat{y} \Rightarrow (\tilde{y} = \tilde{x}))$.
*Merge.* The encoding of the *merge* operator, $y := x$ `default` $z$, contributes to $\Phi(\mathtt{P})$ with the following conjunct: $(\hat{y} \Leftrightarrow (\hat{x} \vee \hat{z})) \wedge \hat{y} \Rightarrow ((\hat{x} \wedge (\tilde{y} = \tilde{x})) \vee (\neg\hat{x} \wedge (\tilde{y} = \tilde{z}))))$.
*Composition.* Consider the composition of two processes $\mathtt{P}_1$ and $\mathtt{P}_2$. Its abstraction $\phi(\mathtt{P}_1|\mathtt{P}_2)$ is defined as follows: $\phi(\mathtt{P}_1) \wedge \phi(\mathtt{P}_2)$.
*Clock Relations.* Given the above rules, we can obtain the following abstraction for derived operators on clocks. Here, $z$ is a signal of type `event`: $\phi(z := \hat{}x) = (\hat{z} \Leftrightarrow \hat{x}) \wedge (\hat{z} \Rightarrow \tilde{z})$; $\phi(x \ \hat{}= \ y) = \hat{x} \Leftrightarrow \hat{y}$; $\phi(z := x \ \hat{}+ \ y) = (\hat{z} \Leftrightarrow (\hat{x} \vee \hat{y})) \wedge (\hat{z} \Rightarrow \tilde{z})$; $\phi(z := x \ \hat{}* \ y) = (\hat{z} \Leftrightarrow (\hat{x} \wedge \hat{y})) \wedge (\hat{z} \Rightarrow \tilde{z})$; $\phi(z := x \ \hat{}- \ y) = (\hat{z} \Leftrightarrow (\hat{x} \wedge \neg\hat{y})) \wedge (\hat{z} \Rightarrow \tilde{z})$; $\phi(z :=$ `when` $b) = (\hat{z} \Leftrightarrow (\hat{b} \wedge \tilde{b})) \wedge (\hat{z} \Rightarrow \tilde{z})$.

*Example.* Consider the Signal program `DEC` (Listing 1), which counts through the output `N` from the value of input `FB` to `1`. It can be observed that the clock of the output signal is more frequent than that of the input. The diagram in Listing 2 illustrates one possible behavior of the program.

```
process DEC =
(? integer FB; ! integer N;)  % FB: input signal, N: output signal %
 (| FB ^= when (ZN<=1)  % ZN holds a value smaller than 1, FB is present %
  | N := FB default (ZN-1)
  | ZN := N$1 init 1    % ZN takes the previous value of N %
  |)
where integer ZN; end; % ZN is defined as local signal %
```

Listing 1 Signal program DEC

```
 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
```

Listing 2 An execution trace of DEC

Applying the abstraction rules above, the clock semantics of `DEC` is represented by the following formula $\Phi(\mathtt{DEC})$, where `ZN <= 1` and `ZN − 1` are replaced by two fresh variables `ZN1` and `ZN2`, and encoded by two uninterpreted function symbols $\mathtt{v}^1_{<=}$ and $\mathtt{v}^1_{-}$, respectively:

$$
\begin{aligned}
&(\widehat{FB} \Leftrightarrow \widehat{ZN1} \wedge \widetilde{ZN1}) \wedge (\widehat{ZN1} \Leftrightarrow \widehat{v^1_{<=}} \Leftrightarrow \widehat{ZN}) \wedge (\widehat{ZN1} \Rightarrow (\widetilde{ZN1} = \widetilde{v^1_{<=}}))\\
\wedge\ &(\widehat{ZN} \Leftrightarrow \widehat{N}) \wedge (\widehat{ZN} \Rightarrow (\widetilde{ZN} = m.N \wedge m.N' = \tilde{N})) \wedge (m.N_0 = 1)\\
\wedge\ &(\widehat{N} \Leftrightarrow \widehat{FB} \vee \widehat{ZN2}) \wedge (\widehat{N} \Rightarrow ((\widehat{FB} \wedge \tilde{N} = \widetilde{FB}) \vee (\neg\widehat{FB} \wedge \tilde{N} = \widetilde{ZN2})))\\
\wedge\ &(\widehat{ZN2} \Leftrightarrow \widehat{v^1_{-}} \Leftrightarrow \widehat{ZN}) \wedge (\widehat{ZN2} \Rightarrow (\widetilde{ZN2} = \widetilde{v^1_{-}}))
\end{aligned}
$$

In the following, we will denote input, output, register, memorization and local variables used in a clock model by $I_{clk}, O_{clk}, R_{clk}, M_{clk}$ and $L_{clk}$, respectively. Note that the memorization variables are introduced by the abstraction.

**Definition 1 (Clock configuration)** Consider a clock model $\Phi(\mathtt{P})$ over a set of variables $\hat{X}$. A *clock configuration* $\widehat{I}$ is an interpretation over $\hat{X}$ such that it is a model of the first-order logic formula $\Phi(\mathtt{P})$.

For instance, $(\widehat{\mathtt{FB}} \mapsto \mathtt{true}, \widehat{\mathtt{N}} \mapsto \mathtt{true}, \widehat{\mathtt{ZN}} \mapsto \mathtt{true}, \widetilde{\mathtt{FB}} \mapsto 6, \tilde{\mathtt{N}} \mapsto 6, \widetilde{\mathtt{ZN}} \mapsto 1)$ is a clock configuration of $\Phi(\mathtt{DEC})$.

*4.1.2 Concrete Clock Semantics*

We rely on the basic elements of *trace semantics* [34] to define the clock semantics of a Signal program. For each $x_i \in X$, we use $\mathbb{D}_{x_i}$ to denote its domain of values, and $\mathbb{D}^{\perp}_{x_i} = \mathbb{D}_{x_i} \cup \{\perp\}$ extends this domain of values with the *absent value* $\perp$, where $\perp \notin \mathbb{D}_{x_i}$. The extended domain of values of $X$ is defined as $\mathbb{D}^{\perp}_{X} = \bigcup_{i=1}^{n} \mathbb{D}_{x_i} \cup \{\perp\}$.

**Definition 2 (Clock events, clock traces)** Given a non-empty set $X$, the set of clock events on $X$, denoted by $\mathcal{E}c_X$, is the set of all possible interpretations $I$ over $X$. The set of clock traces on $X$, denoted by $\mathcal{T}c_X$, is defined by the set of functions $T_c$ defined from the set $\mathbb{N}$ of natural numbers to $\mathcal{E}c_X$, denoted by $T_c : \mathbb{N} \longrightarrow \mathcal{E}c_X$.

An interpretation $I$ is an assignment of values from $X$ to $\mathbb{D}^{\perp}_{X}$. The assignment $I(x) = \perp$ means that $x$ holds no value, while $I(x) = v$ means that $x$ holds the value $v$. Natural numbers represent instants $t = 0, 1, 2, \ldots$ A trace $T_c$ is a chain of clock events. We denote the interpreted value of a variable $x_i$ at instant $t$ by $T_c(t)(x_i)$.

**Definition 3 (Restriction clock trace)** Given a non-empty set $X$, a subset $X_1 \subseteq X$, and a clock trace $T_c$ being defined on $X$, the restriction of $T_c$ onto $X_1$ is denoted by $X_1.T_c$. It is defined as $X_1.T_c : \mathbb{N} \longrightarrow \mathcal{E}c_{X_1}$ such that $\forall t \in \mathbb{N}, \forall x \in X_1, X_1.T_c(t)(x) = T_c(t)(x)$.

Let $X$ be the set of all signals in a program $\mathtt{P}$. We write $[[\mathtt{P}]]_c$ to denote the clock semantics of $\mathtt{P}$, which is defined as the set of all possible clock traces on $X$. For any subset $X_1 \subseteq X$, the set of all restriction clock traces on $X_1$ defines the clock semantics of $\mathtt{P}$ on $X_1$, denoted by $([[\mathtt{P}]]_c)_{\backslash X_1}$.

Let $\Phi(\mathtt{P})$ be the clock model of the program $\mathtt{P}$. We now define the *concrete clock semantics* of a clock model based on the notion of clock configuration. Given a clock configuration $\hat{I}$, the set of clock events according to $\hat{I}$ is the set of interpretations $I$ such that for every signal $x_i$, if $x_i$ holds a value then $\widehat{x_i}$ has the value $\mathtt{true}$ ($x_i$ is present), and $\widetilde{x_i}$ holds the same value as $x_i$. Otherwise, $\widehat{x_i}$ has the value $\mathtt{false}$ (meaning $x_i$ is absent). The set of clock events according to $\hat{I}$ and the set of all clock events of $\Phi(\mathtt{P})$ are computed as follows:

$$\begin{aligned} S_{\mathcal{E}c_X}(\hat{I}) \quad &= \{I \in \mathcal{E}c_X |\ \forall x_i \in X, (I(x_i) = \hat{I}(\widetilde{x_i}) \wedge \hat{I}(\widehat{x_i}) = \mathtt{true}) \\ &\quad \vee (I(x_i) = \perp \wedge \hat{I}(\widehat{x_i}) = \mathtt{false})\} \\ S_{\mathcal{E}c_X}(\Phi(\mathtt{P})) &= \textstyle\bigcup_{\hat{I} \models \Phi(\mathtt{P})} S_{\mathcal{E}c_X}(\hat{I}) \end{aligned}$$

With a set of clock events $S_{\mathcal{E}c_X}(\Phi(\mathtt{P}))$, the concrete clock semantics of $\Phi(\mathtt{P})$ is defined by the following set of clock traces: $\Gamma(\Phi(\mathtt{P})) = \{T_c \in \mathcal{T}c_X |\ \forall t, T_c(t) \in S_{\mathcal{E}c_X}(\Phi(\mathtt{P}))\}$. For any subset $X_1 \subseteq X$, the concrete clock semantics of $\Phi(\mathtt{P})$ on $X_1$ is defined as $\Gamma(\Phi(\mathtt{P}))_{\backslash X_1} = \{X_1.T_c |\ T_c \in \mathcal{T}c_X \text{ and } \forall t, T_c(t) \in S_{\mathcal{E}c_X}(\Phi(\mathtt{P}))\}$. Due to the lack of space, we do not present the proof of soundness of our abstraction.

## 4.2 Clock Model Translation Validation

We adopt the translation validation approach [51,50] to formally verify that the clock semantics is preserved for every transformation of the compiler. In order to apply the translation validation to the transformations, we capture the clock semantics of the original program and its transformed counterpart by means of clock models. We introduce a *refinement* relation which expresses the preservation of clock semantics, as relation on clock models.

### *4.2.1 Clock Refinement*

Let $\Phi(\mathtt{A})$ and $\Phi(\mathtt{C})$ be two clock models of programs $\mathtt{A}$ and $\mathtt{C}$, to which we refer respectively as a source program and its transformed counterpart produced by the compiler. We denote the sets of all signals in $\mathtt{A}$ and $\mathtt{C}$ by $X_A$ and $X_C$, respectively. The corresponding sets of variables which are used to construct the clock models are denoted by $\widehat{X_A}$ and $\widehat{X_C}$. Consider the finite set of common signals $X = X_A \cap X_C$ and the set of common variables which are used to construct the clock models, $\widehat{X} = \widehat{X_A} \cap \widehat{X_C}$. We say that $\mathtt{A}$ and $\mathtt{C}$ have the same clock semantics on $X$ if $\Phi(\mathtt{A})$ and $\Phi(\mathtt{C})$ have the same set of concrete restriction clock traces on $X$:

$$\forall T_c.(X.T_c \in \Gamma(\Phi(\mathtt{C}))_{\backslash X} \Leftrightarrow X.T_c \in \Gamma(\Phi(\mathtt{A}))_{\backslash X})$$

In fact, the compilation makes the transformed program more concrete. For instance, when the Signal compiler performs “endochronization”, which is used to generate sequential executable code [8], the signal with the fastest clock is considered as always present. Moreover, the compiler performs transformations and optimizations for removing or eliminating some redundant behaviors of the source program (e.g., elimination of subexpressions, trivial clock relations). Consequently, the above requirement is too strong to be practical. Hence, we have to relax it as follows:

$$\forall T_c.(X.T_c \in \Gamma(\Phi(\mathtt{C}))_{\backslash X} \Rightarrow X.T_c \in \Gamma(\Phi(\mathtt{A}))_{\backslash X})$$

It expresses that every restriction clock trace of $\Phi(\mathtt{C})$ is also a clock trace of $\Phi(\mathtt{A})$ on $X$, or $\Gamma(\Phi(\mathtt{C}))_{\backslash X} \subseteq \Gamma(\Phi(\mathtt{A}))_{\backslash X}$. We say that $\Phi(\mathtt{C})$ is a *correct clock transformation* of $\Phi(\mathtt{A})$, or $\Phi(\mathtt{C})$ is a *clock refinement* of $\Phi(\mathtt{A})$ on $X$, denoted by $\Phi(\mathtt{C}) \sqsubseteq_{clk} \Phi(\mathtt{A})$.

**Proposition 1** *The clock refinement is reflexive and transitive.*

*Proof* Proposition 1 is proved based on the clock refinement definition. $\Phi(\mathtt{P}) \sqsubseteq_{clk} \Phi(\mathtt{P})$ since $\Gamma(\Phi(\mathtt{P}))_{\backslash X} \subseteq \Gamma(\Phi(\mathtt{P}))_{\backslash X}$. For every clock trace $X.T_c \in \Gamma(\Phi(\mathtt{P}_1))_{\backslash X}$, $\Phi(\mathtt{P}_1) \sqsubseteq_{clk} \Phi(\mathtt{P}_2)$ on $X$ implies $X.T_c \in \Gamma(\Phi(\mathtt{P}_2))_{\backslash X}$. Since $\Phi(\mathtt{P}_2) \sqsubseteq_{clk} \Phi(\mathtt{P}_3)$ on $X$, we have $X.T_c \in \Gamma(\Phi(\mathtt{P}_3))_{\backslash X}$, or $\Phi(\mathtt{P}_1) \sqsubseteq_{clk} \Phi(\mathtt{P}_3)$ on $X$. □

### *4.2.2 Adaptation to the Signal Compiler*

We have to adapt the definition of the above general clock refinement to the case of the Signal compiler [8]. A first consideration is that programs take inputs from both their environment and register values. Then, they compute outputs to react with the environment. Programs may use also local variables to perform output calculations. However, from the outside, the natural observation of the programs is the snapshot of the values of the input and output signals. In our context, it is the snapshot of the *presence* of the input and output signals. For example, for the program DEC, the observation is the tuple of the presence of the signals (FB, N) at a given instant.

A second consideration is that local signals in the source program do not necessarily have counterparts in the transformed program. Conversely, all input and output signals are preserved in the transformations and are represented by identical names in the source and transformed programs. Moreover, all signals in the $R$ set are also preserved in the transformations. Therefore, it is natural to choose the snapshot of the presence of the input and output signals to be the observation for the transformed program.

These considerations let us adapt the above definition of clock refinement as follows. Let $X_A$ and $X_C$ be the sets of all signals in the source program A and its counterpart transformed program C. We write $X_{IO}$ to denote the set of common input and output signals. We say that C is a correct transformation of A if at any instant, the tuples of values representing the presence of the signals in $X_{IO}$ are the same in both programs. Formally, $\Phi(\mathtt{C}) \sqsubseteq_{clk} \Phi(\mathtt{A})$ on $X_{IO}$.

### *4.2.3 Proving Clock Refinement by SMT*

Our aim is proving that $\Phi(\mathtt{C})$ refines $\Phi(\mathtt{A})$ on $X_{IO}$. Let $\widehat{X_A}$, $\widehat{X_C}$, $\widehat{X_{IO}}$ be the set of variables which are used to construct $\Phi(\mathtt{A})$ and $\Phi(\mathtt{C})$, and the set of common variables between the two clock models. For every variable in the clock model $\Phi(\mathtt{C})$ except the common variables in $\widehat{X_{IO}}$, we added "c" as superscript to distinguish them from the variables in the clock model of the source program. The standard way of proving the existence of the clock refinement is based on the following elements:

- The identification of a mapping of variables that maps the non-input/output variables from the clock model $\Phi(\mathtt{C})$ to the non-input/output variables in the clock model $\Phi(\mathtt{A})$. This mapping expresses a relation $\widehat{X_C} \setminus \widehat{X_{IO}} \rightarrow \widehat{X_A} \setminus \widehat{X_{IO}}$; we denote it by: $\widehat{X_A} \setminus \widehat{X_{IO}} = \alpha(\widehat{X_C} \setminus \widehat{X_{IO}})$.
- The premises of a rule such that if the premises hold, then the conclusion, $\Phi(\mathtt{C})$ refines $\Phi(\mathtt{A})$, is `true`. The premise is presented in Fig. 2.

For a mapping of variables $\widehat{X_A} \setminus \widehat{X_{IO}} = \alpha(\widehat{X_C} \setminus \widehat{X_{IO}})$,
**Premise** $\forall \hat{I}$ over $\widehat{X_A} \cup \widehat{X_C}.(\hat{I} \models \Phi(\mathtt{C}) \Rightarrow \hat{I} \models \Phi(\mathtt{A}))$

---

**Conclusion** $\Phi(\mathtt{C}) \sqsubseteq_{clk} \Phi(A)$ on $X_{IO}$

Fig. 2 Rule CLKREF

The rule CLKREF indicates that for any interpretation $\hat{I}$ over $\widehat{X_A} \cup \widehat{X_C}$ such that the mapping of variables $\alpha$ (as a relation) is evaluated to `true` and $\hat{I}$ is a clock configuration of $\Phi(\mathtt{C})$, then it is also a clock configuration of $\Phi(\mathtt{A})$. Then there exists a clock refinement for $(\Phi(\mathtt{C}), \Phi(\mathtt{A}))$. The rule CLKREF is sound based on the following theorem.

**Theorem 1** *For a mapping of variables $\widehat{X_A} \setminus \widehat{X_{IO}} = \alpha(\widehat{X_C} \setminus \widehat{X_{IO}})$, if the formula $\Phi(\mathtt{C}) \Rightarrow \Phi(\mathtt{A})$ is valid, then $\Phi(\mathtt{C}) \sqsubseteq_{clk} \Phi(\mathtt{A})$ on $X_{IO}$.*

*Proof* To prove it, we have to show that for every interpretation $\hat{I}$ over $\hat{X} = \widehat{X_A} \cup \widehat{X_C}$ such that $\alpha$ is evaluated to $\mathtt{true}$, if $\hat{I} \models (\Phi(\mathtt{C}) \Rightarrow \Phi(\mathtt{A}))$, then $\Gamma(\Phi(\mathtt{C}))_{\setminus X_{IO}} \subseteq \Gamma(\Phi(\mathtt{A}))_{\setminus X_{IO}}$. Given $X_{IO}.T_c \in \Gamma(\Phi(\mathtt{C}))_{\setminus X_{IO}}$, it means that $\forall t, T_c(t) \in S_{\mathcal{E}c_X}(\Phi(\mathtt{C}))$. Since for every interpretation $\hat{I}$, $\hat{I} \models \Phi(\mathtt{C})$ implies that $\hat{I} \models \Phi(\mathtt{A})$), thus $S_{\mathcal{E}c_X}(\Phi(\mathtt{C})) \subseteq S_{\mathcal{E}c_X}(\Phi(\mathtt{A}))$ under the variable mapping. We get $T_c(t) \in S_{\mathcal{E}c_X}(\Phi(\mathtt{A}))$ for every $t$. Therefore, we have $T_c \in \Gamma(\Phi(\mathtt{A}))$. □

*Mapping of variables.* Consider a variable $x \in \widehat{X_A} \setminus \widehat{X_{IO}}$, the mapping $\alpha_x$ defines the value of $x$ in the clock model $\Phi(\mathtt{A})$ $\alpha$-related to the value represented in the clock model $\Phi(\mathtt{C})$. We need to describe the mappings $\alpha_x$ for $x^c \in \widehat{X_C} \setminus \widehat{X_{IO}} = M_{clk} \cup R_{clk} \cup L_{clk}$ (memorization, register and local variables). Recall that for every register signal $s$, we introduce memorization variables $m.s$, $m.s'$ in the clock model $\Phi(\mathtt{A})$, and the corresponding memorization variables $m.s^c$, $m.s'^c$ in the clock model $\Phi(\mathtt{C})$. Therefore, we define the following instance of the $\alpha$ mapping for each register signal $s$: $\widetilde{s} = \widetilde{s^c} \Rightarrow m.s = m.s^c \wedge m.s' = m.s'^c$.

For example, in the program DEC, the mapping for the variables $m.N$, $m.N'$, $m.N^c$ and $m.N'^c$ will be given by the formula: $\widetilde{N} = \widetilde{N^c} \Rightarrow m.N = m.N^c \wedge m.N' = m.N'^c$.

It remains to define the instance of the mapping $\alpha$ for variables $\hat{l}, \widetilde{l} \in R_{clk} \cup L_{clk}$ in the clock model $\Phi(\mathtt{A})$ which correspond to register or local signals named $l$ in the program. In a *deterministic* Signal program, a signal $l$ is defined by an equation $l := eq$, and if we follow the definitions of all output and local signals in this equation and apply successive substitutions, then we get that the equation is constructed only from the input and register signals. Note that the compiler will detect non-deterministic programs, for which this property is not respected. Equivalently, in the corresponding clock model $\Phi(\mathtt{A})$, the output, register and local variables are determinately defined from the input $I$ and memorization $M$ variables. The definition is written in the clock model in the form $\hat{l} \Leftrightarrow \hat{f} \wedge (\hat{l} \Rightarrow \widetilde{l} = \widetilde{f})$ or $\hat{l} \Leftrightarrow \hat{f} \wedge (\hat{l} \Rightarrow \widetilde{l} = \widetilde{f}) \wedge \widetilde{f_0}$, where $\hat{f}$, $\widetilde{f}$ and $\widetilde{f_0}$ are the formulas which define respectively the clock, the value, and the initial value of the signal $l$ in the clock model $\Phi(\mathtt{A})$. Therefore, we define the following instance of the $\alpha$ mapping in the clock model, corresponding to each register or local signal $l$: $\hat{l} \Leftrightarrow \hat{f} \wedge (\hat{l} \Rightarrow \widetilde{l} = \widetilde{f})$ or $\hat{l} \Leftrightarrow \hat{f} \wedge (\hat{l} \Rightarrow \widetilde{l} = \widetilde{f}) \wedge \widetilde{f_0}$.

For example, in the program DEC, the mapping for the variables $\widehat{ZN}$ and $\widetilde{ZN}$ in the clock model $\Phi(\mathtt{DEC})$ corresponding to the local variable $ZN$ will be given by the formula: $(\widehat{ZN} \Leftrightarrow \widehat{N}) \wedge (\widehat{ZN} \Rightarrow (\widetilde{ZN} = m.N \wedge m.N' = \widetilde{N})) \wedge (m.N_0 = 1)$.

Thus, the mapping of variables $\widehat{X_A} \setminus \widehat{X_{IO}} = \alpha(\widehat{X_C} \setminus \widehat{X_{IO}})$ is expressed as the following formula:

$$\bigwedge_{m.s \in M} (\widetilde{s} = \widetilde{s^c} \Rightarrow (m.s = m.s^c \wedge m.s' = m.s'^c)) \wedge \bigwedge_{\hat{l}, \widetilde{l} \in R \cup L} (\hat{l} \Leftrightarrow \hat{f} \wedge (\hat{l} \Rightarrow \widetilde{l} = \widetilde{f})) \text{ or}$$
$$\bigwedge_{m.s \in M} (\widetilde{s} = \widetilde{s^c} \Rightarrow (m.s = m.s^c \wedge m.s' = m.s'^c)) \wedge \bigwedge_{\hat{l}, \widetilde{l} \in R \cup L} (\hat{l} \Leftrightarrow \hat{f} \wedge (\hat{l} \Rightarrow \widetilde{l} = \widetilde{f}) \wedge \widetilde{f_0})$$

*SMT solving.* To solve the validity of the formula $(\Phi(\mathtt{C}) \Rightarrow \Phi(\mathtt{A}))$ in Theorem 1 under the mapping of variables, a SMT solver is needed since this formula involves non-Boolean variables and *uninterpreted functions* (using a SAT solver would not be sufficient). A SMT solver decides the satisfiability of arbitrary logic formulas of linear real and integer arithmetic, scalar types, other user-defined data structures, and uninterpreted functions. If the formula belongs to the decidable theory, the solver gives two types of answers: `sat` when the formula has a model (there exists an interpretation that satisfies it); or `unsat`, otherwise. In our case, we will ask the solver to check whether the formula $\neg(\Phi(\mathtt{C}) \wedge \widehat{X_A} \setminus \widehat{X_{IO}} = \alpha(\widehat{X_C} \setminus \widehat{X_{IO}}) \Rightarrow \Phi(\mathtt{A}))$ is unsatisfiable, since this formula is unsatisfiable iff $\models (\Phi(\mathtt{C}) \wedge \widehat{X_A} \setminus \widehat{X_{IO}} = \alpha(\widehat{X_C} \setminus \widehat{X_{IO}}) \Rightarrow \Phi(\mathtt{A}))$. In our translation validation, the clock models which are constructed from Boolean or numerical variables and uninterpreted functions belong to a part of first-order logic which has a *small model* property according to [14]. The numerical variables are involved only in some implications with uninterpreted functions such as $(\widetilde{x} = \widetilde{x'} \wedge \widetilde{y} = \widetilde{y'}) \Rightarrow \widetilde{v^i_\square} = \widetilde{v^j_\square}$. In addition, the formula is quantifier-free. This means that the check of satisfiability can be established by examining a certain finite cardinality of models. Therefore, the formula can be solved efficiently and this significantly improves the scalability of the solver.

The flow and main components of the validator are depicted in Fig. 3. First, it takes

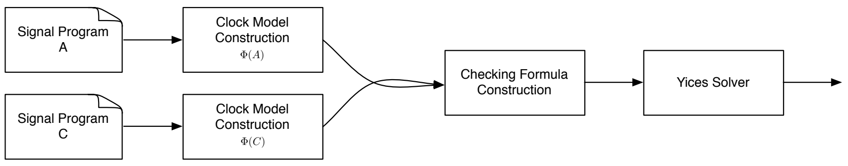
Signal Program A
Signal Program C
Clock Model Construction $\Phi(A)$
Clock Model Construction $\Phi(C)$
Checking Formula Construction
Yices Solver

Fig. 3 Clock model translation validation

the input program `P` and its transformed counterpart `P_BASIC_TRA` (refer to Fig. 1), and constructs the corresponding clock models. Indicatively, the transformed program `DEC_BASIC_TRA` obtained from the source program `DEC` as result of the first phase of the Signal compiler is illustrated in Listing 3. Its clock model is twice as large as that of the program `DEC` (Section 4.1.1).

```
(| CLK := CLK_N ^- CLK_FB |)
 | (| CLK_N := CLK_N ^+ CLK_FB
    | CLK_N ^= N ^= ZN
    |)
 | (| CLK_FB := when (ZN<=1)
    | CLK_FB ^= FB
    | CLK_12 := when (not (ZN<=1))
    |)
 | (| N := (FB when CLK_FB) default ((ZN-1) when CLK)
    | ZN := N$1 init 1
    |)
|)
```

Listing 3 `DEC_BASIC_TRA` in Signal

These clock models are combined as the formula ($\Phi$(P_BASIC_TRA) $\Rightarrow$ $\Phi$(P)). In the solving phase, it checks the validity of the formula $\Phi$(P_BASIC_TRA) $\Rightarrow$ $\Phi$(P). The result of this check can be exploited for the preservation of clock semantics of the transformations. If the formula is not valid then it emits a compiler bug. Otherwise, the compiler continues its work. The same procedure is applied for the next steps of the compiler. Finally, our verification process asserts that $\Phi$(P_BOOL_TRA) $\sqsubseteq_{clk}$ $\Phi$(P_BASIC_TRA) $\sqsubseteq_{clk}$ $\Phi$(P) along the transformations of the compiler. We delegate the checking of the clock refinement to a SMT solver. For our experiments, we consider the Yices [19] solver, which is one of the best solvers at the SMT-Comp competition [57].

## 5 Preservation of Data Dependencies

This section focuses on constructing a validator that proves the preservation of data dependencies in the *scheduling synthesis* phase of the Signal compiler. It describes how the preservation of data dependencies among signals can be proved by showing that for every pair of signals $x$ and $y$ in the source program, if there exists a data dependency between $x$ and $y$, then this dependency also exists in the compiled program. Data dependencies among signals are represented by a *Synchronous Data-flow Dependency Graph* (SDDG). Given two SDDGs of the source and compiled programs, a *refinement* relation between them is formally defined, which expresses the semantic preservation of data dependencies. Here again, we delegate checking the refinement to a SMT solver.

### 5.1 Synchronous Data-flow Dependency Graph

Consider the basic process $y := x$ `default` $z$ with its clock relations among signals; the “valid” statuses of the signals are: $x$ is present and $y$ is present; or $x$ is absent, $z$ is present, and $y$ is present; or $x$, $y$ and $z$ are absent. They can be represented by $\hat{y} \Leftrightarrow (\hat{x} \vee \hat{z})$ in our clock abstraction. Following these statuses, the data dependencies among signals in this process are depicted in Fig. 4, where labels on edges represent the conditions at which the corresponding dependencies are effective. For instance, when $\hat{x} =$ `true`, $y$ is defined by $x$; otherwise it is defined by $z$ when $\hat{x} =$ `false` and $\hat{z} =$ `true`. We can see that the graph in this figure has the following property: an edge cannot exist if one of its extremity nodes is not present (or the corresponding signal holds no value). In our example, this property can be expressed in the abstraction as: $\hat{x} \Rightarrow \hat{y} \wedge \hat{x}; \neg\hat{x} \wedge \hat{z} \Rightarrow \hat{y} \wedge \hat{z}$.

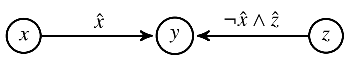
$$\begin{tikzcd} x \arrow[r, "\hat{x}"] & y & z \arrow[l, "\neg\hat{x} \wedge \hat{z}"'] \end{tikzcd}$$

Fig. 4 The SDDG of the *merge* operator

An SDDG associated with a program is a labeled directed graph [49] in which nodes are signal and clock variables and edges represent dependencies between nodes. Edges are labeled by clocks (represented by first-order logic formulas). An edge clock tells when the dependency is effective: when the clock is present. Formally, an SDDG is defined as follows:

**Definition 4** (SDDG) An SDDG is a labeled directed graph $G = \langle N, E, I, O, C, m_N, m_E \rangle$, where:

- $N$ is a finite set of nodes, each of which represents the equation defining a signal or a clock;
- $E \subseteq N \times N$ is the set of dependencies between nodes;
- $I \subseteq N$ is the set of input nodes;
- $O \subseteq N$ is the set of output nodes;
- $C$ is the set of first-order logic formulas, called *clock constraints*; the clock constraints are encoded using the abstraction described in Section 4.1;
- $m_N : N \longrightarrow C$ is a mapping labeling each node with a clock; it defines the existence condition of a node;
- $m_E : E \longrightarrow C$ is a mapping labeling each edge with a clock constraint; it defines the existence condition of an edge.

The clock labeling in an SDDG provides a dynamic dependency feature. This clock labeling enforces the following property: *an edge exists only if its two extremity nodes exist*. This can be translated in our abstraction as: $\forall (x, y) \in E, m_E(x, y) \Rightarrow (m_N(x) \wedge m_N(y))$. A dependency between two nodes (signals or clocks) $x$ and $y$ at a clock condition $m_E(x, y) = \hat{c}$ is denoted by $x \xrightarrow{\hat{c}} y$. A *dependency path* from $x$ to $y$ is any set of nodes $s = \{x_0, x_1, \ldots, x_k\}$ such that $x = x_0 \xrightarrow{\widehat{c_0}} x_1 \xrightarrow{\widehat{c_1}} \ldots \xrightarrow{\widehat{c_{k-1}}} x_k = y$.

In Table 1, we indicate the dependencies among signals for the core language (the subclocks $[c]$ and $[\neg c]$ are encoded as $\hat{c} \wedge \tilde{c}$ and $\hat{c} \wedge \neg\tilde{c}$, respectively, in our abstraction). The edges are labeled by clocks, which are represented by a first-order formula. The last line of the table expresses added dependencies for any dependency $x \xrightarrow{c} y$. The dependencies in this table respect in particular the mentioned implicit property of an SDDG: for instance, the encoding of the primitive operator *sampling* satisfies that $\hat{y} \Rightarrow \hat{x} \wedge \hat{y}$ and $\hat{y} \Rightarrow \hat{b} \wedge \hat{y}$.

Following the above construction rules, we can obtain the SDDG in Fig. 5 for the program `DEC`, where we introduce two fresh variables $ZN1$ and $ZN2$ to replace the expressions $ZN <= 1$ and $ZN - 1$, respectively. Note that for more clarity we omit some dependencies (for example, the signal $N$ depends on its clock $\widehat{N}$). The clocks which label the edges in the graph are encoded as the first-order formulas given as follows:

$$\begin{aligned}
\widehat{ZN1} &= \widehat{ZN2} = \widehat{ZN} = \widehat{N} \\
\widehat{FB} &= \widehat{ZN1} \wedge \widetilde{ZN1} \\
\widehat{N} &= \widehat{FB} \vee \widehat{ZN2}
\end{aligned}$$

| Operators | Encoding in SDDG |
|---|---|
| $x$ | $\hat{x} \xrightarrow{\hat{x}} x$, $m_N(\hat{x}) = \hat{x}$, $m_N(x) = \hat{x}$ |
| $c$ (Boolean signal) | $c \xrightarrow{[c]} [c]$, $m_N(c) = \hat{c}$, $m_N([c]) = [c]$, $c \xrightarrow{[\neg c]} [\neg c]$, $m_N(c) = \hat{c}$, $m_N([\neg c]) = [\neg c]$ |
| $y := f(x_1, \dots, x_n)$ | $x_1 \xrightarrow{\hat{y}} y, \dots, x_n \xrightarrow{\hat{y}} y$, $m_N(x_i) = \widehat{x_i}$, $m_N(y) = \hat{y}, i = 1, \dots, n$ |
| $y := x\$1$ `init` $a$ | $m_N(x) = \hat{x}, m_N(y) = \hat{y}$ |
| $y := x$ `when` $b$ | $x \xrightarrow{\hat{y}} y$, $m_N(x) = \hat{x}, m_N(y) = \hat{y}$, $b \xrightarrow{\hat{y}} \hat{y}$, $m_N(b) = \hat{b}$, $m_N(\hat{y}) = \hat{y}$ |
| $y := x$ `default` $z$ | $x \xrightarrow{\hat{x}} y$, $m_N(x) = \hat{x}$, $m_N(y) = \hat{y}$, $z \xrightarrow{\hat{z} \wedge \neg \hat{x}} y$, $m_N(z) = \hat{z}$, $m_N(y) = \hat{y}$ |
| $x \xrightarrow{c} y$ | $[c] \xrightarrow{[c]} y$, $m_N([c]) = [c]$, $m_N(y) = \hat{y}$ |

Table 1 The dependencies of the core language

$$\begin{tikzcd}
 & & ZN \arrow[d, "\widehat{ZN1}"] \arrow[dr, bend left, "\widehat{ZN2}"] & \\
\widehat{FB} \arrow[dd, "\widehat{FB}"] & \widehat{ZN1} \arrow[ur, "\widehat{ZN}"] \arrow[l, "\widehat{FB}"'] \arrow[r, "\widehat{ZN1}"] \arrow[drr, bend right, "\widehat{ZN2}"'] & ZN1 \arrow[ll, bend left, "\widehat{FB}"] & \\
 & & & ZN2 \arrow[ddll, "\neg\widehat{FB} \wedge \widehat{ZN2}"] \\
FB \arrow[dr, "\widehat{FB}"] & & & \\
 & N & &
\end{tikzcd}$$

Fig. 5 The SDDG of `DEC`

## 5.2 Translation validation of SDDG

We adopt the translation validation approach to prove the correctness of the compiler in the *scheduling synthesis* phase of the compilation process. Given two SDDGs, we first formalize the notion of "correct implementation" as a *dependency refinement* relation. This refinement expresses the semantic preservation of data dependencies. Then, as for clock models, we propose a method to implement our verification framework by the use of a SMT solver for checking the existence of the above refinement relation.

### *5.2.1 Definition of correct implementation: Dependency refinement*

Let SDDG(`A`) = $\langle N_A, E_A, I_A, O_A, C_A, m_{N_A}, m_{E_A} \rangle$ be the SDDG of the source program and SDDG(`C`) = $\langle N_C, E_C, I_C, O_C, C_C, m_{N_C}, m_{E_C} \rangle$ the SDDG of its transformed counterpart produced by the Signal compiler. Let $x$ and $y$ be two signals in both programs `A` and C. A dependency path from the signal $x$ to the signal $y$ in SDDG(C) is a *reinforcement* of the dependency path from $x$ to $y$ in SDDG(A) if the following condition holds: at any instant $t$, if the dependency path in SDDG(A) is effective (meaning that the conjunction of all the conditions associated with the path is evaluated to be `true` at $t$), then the corresponding dependency path in SDDG(C) is also effective. The formal definition of reinforcement is given as follows.

**Definition 5 (Reinforcement)** Let $dp_1 = \langle x_0, x_1, \ldots, x_n \rangle$ and $dp_2 = \langle x'_0, x'_1, \ldots, x'_m \rangle$ be two dependency paths in SDDG(A) and SDDG(C), respectively, where $x = x_0 = x'_0$, $y = x_n = x'_m$ and $\widehat{c_i}$, $\widehat{c'_j}$ denote the clock constraints $m_{E_A}(x_i, x_{i+1})$ and $m_{E_C}(x'_j, x'_{j+1})$. It is said that $dp_2$ is a *reinforcement* of $dp_1$ iff the following formula is valid:

$$\models \bigwedge_{i=0}^{n-1} \widehat{c_i} \Rightarrow \bigwedge_{j=0}^{m-1} \widehat{c'_j}$$

We write $dp_2 \preceq_{dep} dp_1$ to denote the fact that $dp_2$ is a reinforcement of $dp_1$. The assertion $\models \bigwedge_{i=0}^{n-1} \widehat{c_i} \Rightarrow \bigwedge_{j=0}^{m-1} \widehat{c'_j}$ indicates that if, at any instant, the dependency path $dp_1$ in SDDG(A) is effective, then the dependency path $dp_2$ in SDDG(C) is also effective. In the special case when $m = n = 1$, $x \xrightarrow{\widehat{c'_0}} y$ is a reinforcement of $x \xrightarrow{\widehat{c_0}} y$ iff $\models \widehat{c_0} \Rightarrow \widehat{c'_0}$.

Consider a dependency path from $x$ to $y$ in an SDDG graph. Assume that there exists a path from $y$ to $x$, that makes a dependency cycle between $x$ and $y$. We say that such a cycle is a deadlock iff the dependencies of $x$ to $y$ and vice-versa are effective at the same time. The formal definition of deadlock is given as follows.

**Definition 6 (Deadlock)** Let $dp = \langle x_0, x_1, \ldots, x_n, x_0 \rangle$ be a cycle in an SDDG graph. The dependency cycle $dp$ stands for a deadlock if the conjunction of all the clock constraints associated with the cycle is satisfiable, meaning that there exists some interpretation that makes the conjunction formula $m_E(x_0, x_1) \wedge m_E(x_1, x_2) \wedge \ldots \wedge m_E(x_n, x_0)$ `true`.

Obviously, a dependency cycle does not stand for a deadlock if the conjunction of all the clock constraints associated with the cycle, in which the dependencies are effective, is identically `false`: that means that the dependencies of the cycle cannot be present at the same time. It can be expressed as:

$$M \not\models \bigwedge_{i=0}^{n} \widehat{c_i},$$

where the $\widehat{c_i}$ are the clock constraints associated with the dependencies of the cycle. It indicates that there is no interpretation that makes the conjunction of all the clock constraints associated with the cycle `true`. Based on the above definition of deadlock, an SDDG is *deadlock-free* if every dependency cycle in the graph does not stand for a deadlock.

**Definition 7 (Deadlock-consistency)** Let $dp_1$ and $dp_2$ be two dependency paths from the signal $x$ to the signal $y$ in SDDG(`A`) and SDDG(`C`), respectively. The dependency path $dp_2$ is *deadlock-consistent* with $dp_1$ if the following condition is satisfied: if there is no dependency cycle between $x$ and $y$ in SDDG(`A`) which is a deadlock, then all dependency cycles between $x$ and $y$ in SDDG(`C`) are not deadlocks.

Let $dp_1 = \langle x_0, x_1, \ldots, x_n \rangle$ and $dp_2 = \langle x'_0, x'_1, \ldots, x'_m \rangle$ be two dependency paths in SDDG(`A`) and SDDG(`C`), respectively, where $x = x_0 = x'_0$, $y = x_n = x'_m$ and $\widehat{c_i}$, $\widehat{c'_j}$ denote the clock constraints $m_E(x_i, x_{i+1})$ and $m_E(x'_j, x'_{j+1})$. The Definition 7 can be expressed as follows. For any dependency path $dp_1^{inv} = \langle x_0^{inv}, x_1^{inv}, \ldots, x_p^{inv} \rangle$, where $x_0^{inv} = y$, $x_p^{inv} = x$, that forms a cycle between $x$ and $y$ in SDDG(`A`), then for every dependency path $dp_2^{inv} = \langle x_0^{inv'}, x_1^{inv'}, \ldots, x_q^{inv'} \rangle$, where $x_0^{inv'} = y$, $x_q^{inv'} = x$, that forms a cycle between $x$ and $y$ in SDDG(`C`), it satisfies:

$$\models (\bigwedge_{i=0}^{n-1} \widehat{c_i} \wedge \bigwedge_{j=0}^{p-1} \widehat{c_j^{inv}}) \Leftrightarrow \texttt{false} \Rightarrow (\bigwedge_{k=0}^{m-1} \widehat{c'_k} \wedge \bigwedge_{l=0}^{q-1} \widehat{c_l^{inv'}}) \Leftrightarrow \texttt{false}$$

We write $dp_2 \asymp_{dep} dp_1$ to denote the fact that $dp_2$ is deadlock-consistent with $dp_1$. Deadlock-consistency expresses the fact that if there is a cycle between two signals $x$ and $y$ in the graph of the source program such that it does not stand for a deadlock, then in the graph of the compiled program, every cycle between $x$ and $y$ must not stand for a deadlock. In the special case where $m = n = p = q = 1$, $x \xrightarrow{\widehat{c'_0}} y$ is deadlock-consistent with $x \xrightarrow{\widehat{c_0}} y$ if $\models (c_0 \wedge c_0^{inv} \Leftrightarrow \texttt{false}) \Rightarrow (c'_0 \wedge c_0^{inv'} \Leftrightarrow \texttt{false})$.

Recall that SDDG(`A`) and SDDG(`C`) are two Synchronous Data-flow Dependency Graphs, to which we refer respectively as the data dependency representations of the source program and its transformed counterpart produced by the Signal compiler. We assume that they have the same set of nodes, $N_A = N_C$. Let $x$ and $y$ be two signals, we say that `C` preserves the data dependencies among signals in `A` if the following conditions are satisfied:

- For every dependency path from the signal $x$ to the signal $y$ in `A`, there exists a dependency path from $x$ to $y$ in `C`.
- If there is no deadlock in `A`, then `C` introduces no deadlock. In other words, if $A$ is deadlock-free, then it is required that `C` is deadlock-free.

These conditions can be expressed in terms of the Synchronous Data-flow Dependency Graphs as follows:

- For every dependency path from the signal $x$ to the signal $y$ in SDDG(`A`) at a clock constraint $\widehat{c_1}$, then there exists a dependency path from $x$ to $y$ at a clock constraint $\widehat{c_2}$ in SDDG(`C`) such that the dependency path in SDDG(`C`) is effective whenever the dependency path in SDDG(`A`) is effective.
- If SDDG(`A`) is deadlock-free, then SDDG(`C`) is also deadlock-free.

If the SDDGs satisfy the above conditions, we say that SDDG(`C`) is a *dependency refinement* of SDDG(`A`) on $N_A$ and `C` is a correct implementation of `A`. We write $SDDG(\texttt{C}) \sqsubseteq_{dep} SDDG(\texttt{A})$ to denote the fact that there exists a dependency refinement relation between SDDG(C) and SDDG(A). We formalize the definition of dependency refinement as follows.

**Definition 8 (Dependency refinement)** Let SDDG(A) and SDDG(C) be two Synchronous Data-flow Dependency Graphs, SDDG(C) is a *dependency refinement* of SDDG(A) if:

1. for every dependency path $dp_1 = \langle x_0, x_1, \dots, x_n \rangle$ in SDDG(A), there exists a dependency path $dp_2 = \langle x'_0, x_1, \dots, x'_m \rangle$ in SDDG(C) such that $dp_2 \preceq_{dep} dp_1$;
2. for any two dependency paths $dp_1 = \langle x_0, x_1, \dots, x_n \rangle$ in SDDG(A) and $dp_2 = \langle x'_0, x'_1, \dots, x'_m \rangle$ in SDDG(C) such that $x_0 = x'_0$ and $x_n = x'_m$, then $dp_2 \asymp_{dep} dp_1$.

Based on the definitions of reinforcement, deadlock-consistency and dependency refinement relations, important properties are stated in Proposition 2.

**Proposition 2** *The reinforcement, deadlock-consistency and dependency refinement are reflexive and transitive:*

1. $\forall dp, dp \preceq_{dep} dp$
2. $\forall dp, dp \asymp_{dep} dp$
3. $\forall SDDG(P), SDDG(P) \sqsubseteq_{dep} SDDG(P)$
4. *If* $dp_1 \preceq_{dep} dp_2$ *and* $dp_2 \preceq_{dep} dp_3$ *then* $dp_1 \preceq_{dep} dp_3$
5. *If* $dp_1 \asymp_{dep} dp_2$ *and* $dp_2 \asymp_{dep} dp_3$ *then* $dp_1 \asymp_{dep} dp_3$
6. *If* $SDDG(P_1) \sqsubseteq_{dep} SDDG(P_2)$ *and* $SDDG(P_2) \sqsubseteq_{dep} SDDG(P_3)$ *then* $SDDG(P_1) \sqsubseteq_{dep} SDDG(P_3)$

*Proof* The proof is based on the definitions of reinforcement, deadlock-consistency and dependency refinement.
*Reinforcement:* For every dependency path $dp$, we always have $dp \preceq_{dep} dp$.
Assume that $dp_1 \preceq_{dep} dp_2$ and $dp_2 \preceq_{dep} dp_3$, we have $(\models \bigwedge_{i=0}^{n-1} \widehat{c_i} \Rightarrow \bigwedge_{j=0}^{m-1} \widehat{c'_j})$ and $(\models \bigwedge_{j=0}^{m-1} \widehat{c'_j} \Rightarrow \bigwedge_{k=0}^{p-1} \widehat{c''_k})$. Thus, $(\models \bigwedge_{i=0}^{n-1} \widehat{c_i} \Rightarrow \bigwedge_{k=0}^{p-1} \widehat{c''_k})$, or $dp_1 \preceq_{dep} dp_3$.
*Deadlock-consistency:* For every dependency path $dp$, we always have $dp \asymp_{dep} dp$.
Because $dp_1 \asymp_{dep} dp_2$ and $dp_2 \asymp_{dep} dp_3$, we have $\models (\bigwedge_{i=0}^{n-1} \widehat{c_i} \wedge \bigwedge_{j=0}^{p-1} \widehat{l_j}) \Leftrightarrow \texttt{false} \Rightarrow (\bigwedge_{u=0}^{m-1} \widehat{c'_u} \wedge \bigwedge_{v=0}^{q-1} \widehat{l'_u}) \Leftrightarrow \texttt{false}$ and $\models (\bigwedge_{u=0}^{m-1} \widehat{c'_u} \wedge \bigwedge_{v=0}^{q-1} \widehat{l'_u}) \Leftrightarrow \texttt{false}) \Rightarrow (\bigwedge_{t=0}^{r-1} \widehat{c''_t} \wedge \bigwedge_{z=0}^{s-1} \widehat{l''_z}) \Leftrightarrow \texttt{false}$. Therefore, $\models (\bigwedge_{i=0}^{n-1} \widehat{c_i} \wedge \bigwedge_{j=0}^{p-1} \widehat{l_j}) \Leftrightarrow \texttt{false} \Rightarrow (\bigwedge_{t=0}^{r-1} \widehat{c''_t} \wedge \bigwedge_{z=0}^{s-1} \widehat{l''_z}) \Leftrightarrow \texttt{false}$, or $dp_1 \asymp_{dep} dp_3$.
*Dependency refinement:* For every dependency path $dp \in$ SDDG($P$), we have $dp \preceq_{dep} dp$ and $dp \asymp_{dep} dp$, thus SDDG($P$) $\sqsubseteq_{dep}$ SDDG($P$).
For every dependency path $dp_3 \in$ SDDG($P_3$), there exists a dependency path $dp_2 \in$ SDDG($P_2$) such that $dp_2 \preceq_{dep} dp_3$. Then, there exists a dependency path $dp_1 \in$ SDDG($P_1$) such that $dp_1 \preceq_{dep} dp_2$. Following the transitivity of the reinforcement, we have $dp_1 \preceq_{dep} dp_3$. In the same way, for every dependency path $dp_3 \in$ SDDG($P_3$), any dependency path $dp_1 \in$ SDDG($P_1$) satisfies $dp_1 \asymp_{dep} dp_3$. Therefore, SDDG($P_1$) $\sqsubseteq_{dep}$ SDDG($P_3$). □

### 5.2.2 Adaptation to the Signal compiler

Again, we have to adapt the above definition of dependency refinement to the case of the Signal compiler. Relying on the same considerations we made previously in Section 4.2.2 about clocks, we note that in our context, the natural observation from the outside is the snapshot of the dependencies among the input and output signals. For example, for the program DEC, the observation is the dependencies between the signals $(FB, N)$ at a given instant. And it is natural to choose also the observation for the transformed program as the snapshot of the dependencies among the input and output signals.

This makes us adapt the definition of clock refinement as follows. Let SDDG(A) and SDDG(C) be two Synchronous Data-flow Dependency Graphs such that they have same set of input and output nodes, $I_A = I_C$ and $O_A = O_C$. We say that C is a correct implementation of A if at any instant, the dependencies among the signals in $I_A \cup O_A$ are also the dependencies among the signals in $I_C \cup O_C$. Formally, $SDDG(\mathtt{C}) \sqsubseteq_{dep} SDDG(\mathtt{A})$ on $I_A \cup O_A$.

### 5.2.3 Proving Dependency Refinement by SMT

We propose a method to check the existence of a refinement between two SDDGs ùin Definition 8 using a SMT solver [19,57]. Let SDDG(A) and SDDG(C) be the Synchronous Data-flow Dependency Graphs of given source and compiled programs. The set of all common input and output signals between A and C is represented by the common set of input and output nodes in the graphs, $I_A \cup O_A$. For all signals or clock variables in SDDG(C) except the common input and output signals, we add "c" as superscript to distinguish them from the signals in SDDG(A). The mapping of variables that maps the non input/output signals from SDDG(A) to the non input/output signals in SDDG(C) is constructed as described in Section 4.2.3. Our aim is proving that SDDG(C) refines SDDG(A) on $I_A \cup O_A$.

To check the existence of the dependency refinement, we apply graph traversals of SDDG(A) and SDDG(C) to verify the following conditions:

- for every path $dp_1$ from $x$ to $y$ such that $x, y \in I_A \cup O_A$ in SDDG(A), there exists a reinforcement $dp_2$ from $x$ to $y$ in SDDG(C);
- for every path $dp_1$ from $x$ to $y$ such that $x, y \in I_A \cup O_A$ in SDDG(A), any path $dp_2$ from $x$ to $y$ in SDDG(C) is deadlock-consistent with the path $dp_1$.

Consider two dependency paths $dp_1 = \langle x_0, x_1, \ldots, x_n \rangle$ and $dp_2 = \langle x'_0, x'_1, \ldots, x'_m \rangle$ in SDDG(A) and SDDG(C), respectively, where $\widehat{c_i}$ and $\widehat{c'_j}$ denote the clock constraints $m_E(x_i, x_{i+1})$ and $m_E(x'_j, x'_{j+1})$. From Definition 5, $dp_2$ is a reinforcement of $dp_1$ iff the following formula is valid:

$$\bigwedge_{i=0}^{n-1} \widehat{c_i} \Rightarrow \bigwedge_{j=0}^{m-1} \widehat{c'_j} \qquad (1)$$

In the same way, to verify deadlock-consistency between $dp_1$ and $dp_2$, we have to check the validity of the following formula:

$$(\bigwedge_{i=0}^{n-1} \widehat{c_i} \wedge \bigwedge_{j=0}^{p-1} \widehat{c_j^{inv}}) \Leftrightarrow \mathtt{false} \Rightarrow (\bigwedge_{k=0}^{m-1} \widehat{c'_k} \wedge \bigwedge_{l=0}^{q-1} \widehat{c_l^{inv'}}) \Leftrightarrow \mathtt{false} \qquad (2)$$

To solve the validity of the above formulas under the mapping of variables, a SMT solver is needed since these formulas involve non-Boolean variables and uninterpreted functions as in our abstraction of clock semantics. In our case, we shall ask the solver to check whether the formula ¬(1) is unsatisfiable, since this formula is unsatisfiable iff $\models$ (1). We do the same for the formula (2), meaning that we ask the solver to check whether the formula ¬(2) is unsatisfiable.

As for clock semantics, satisfiability can be established by examining a certain finite cardinality of models, and it can be solved efficiently [2,9]. Notice that we do not provide here any specific algorithm to find cycles in a directed graph; interested readers can refer to any research on this problem (e.g. the work of D.B. Johnson [32]).

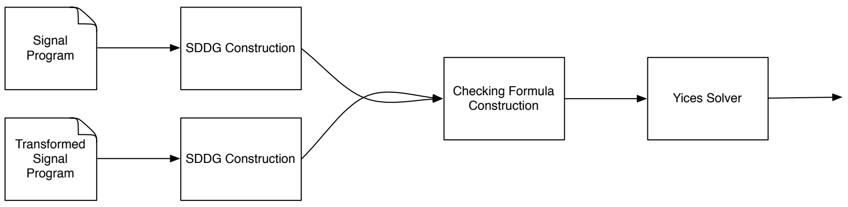

Fig. 6 SDDG translation validation

The main components of the validator are depicted in Fig. 6. It works as follows. First, it takes the input program `A` and the counterpart transformed program `C`. It constructs the corresponding Synchronous Data-flow Dependency Graphs. Then, it establishes the first-order logic formulas corresponding to formula (1) and formula (2), and the mapping of variables from the transformed program to the input program. Finally, in the solving phase, it checks the validity of the formulas in the previous step to indicate the dependency refinement. As illustration, the SDDG of the program `DEC_SEQ_TRA` resulting from the scheduling synthesis phase of the Signal compiler applied to the program `DEC` is depicted in Fig. 7 (`FB1` and `ZN3` are fresh variables, `FB1` replaces `FB when C_FB`). Considering the SDDGs of `DEC` (obtained after the clock synthesis phase) and of `DEC_SEQ_TRA`, checking the dependency refinement amounts to finding all dependency paths from `FB` to `N` to generate the formulas (1) and (2). In this case:

$$\widehat{FB} \Rightarrow (\widehat{FB1^c} \wedge \widehat{FB1^c})$$

$$\begin{tikzcd}
 & & & & ZN1^c \arrow[dl, "\widehat{C\_FB^c}"'] \arrow[ddrrr, bend left, "\widehat{C\_CLK}\_12^c"] & & & \\
FB \arrow[dr, "\widehat{FB1^c}"'] & & \widehat{FB1^c} \arrow[dl, "\widehat{FB1^c}"] & C\_FB^c \arrow[l, "\widehat{FB1^c}"'] & & ZN^c \arrow[ul, "\widehat{ZN1^c}"'] \arrow[dr, "\widehat{ZN2^c}"'] & & \\
 & FB1^c \arrow[ddrr, "\widehat{FB1^c}"'] & & & & & ZN2^c \arrow[dll, "\widehat{ZN3^c}"] & C\_CLK\_12^c \arrow[ddll, "\widehat{ZN3^c}"] \\
 & & & & ZN3^c \arrow[dl, "\neg\widehat{FB1^c} \wedge \widehat{ZN3^c}"] & & & \\
 & & & N & & \widehat{ZN3^c} \arrow[ul, "\widehat{ZN3^c}"'] & &
\end{tikzcd}$$

Fig. 7 The SDDG of DEC_SEQ_TRA

## 6 Value-equivalence of Variables

Following clock and dependency validation, the next step focuses on proving that every output signal in the source program and the corresponding variable in the generated C program are assigned the same values at all times. Still in a translation validation approach, we use a *value-graph* as 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. Let $\mathtt{C}_{dep}$ and $\mathtt{C}$ be the intermediate forms of the source program $\mathtt{A}$ as the results of the *scheduling synthesis* phase and the C code generation, respectively. $Cp^{sig}$ denotes the unverified Signal compiler, which compiles $\mathtt{C}_{dep}$ into either $\mathtt{C} = Cp^{sig}(\mathtt{A})$ or a compilation error. We now associate $Cp$ with a validator, checking that at any time, for any output signal $x$ in $\mathtt{C}_{dep}$ and the corresponding variable $x^c$ in $\mathtt{C}$, they have the same values (denoted by $\widetilde{x} = \widetilde{x^c}$). We denote this property by $\mathtt{C} \sqsubseteq_{val} \mathtt{C}_{dep}$.

The main components of the validator are depicted in Fig. 8. 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 (normalization). The set of rewrite rules reflects the general rules of inference of the operators and 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` (it reflects 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. In the best case, when semantics has been preserved, this check has constant time complexity $O(1)$. In fact, it is expected that, if there is no compiler bug, transformations and optimizations are semantics-preserving.

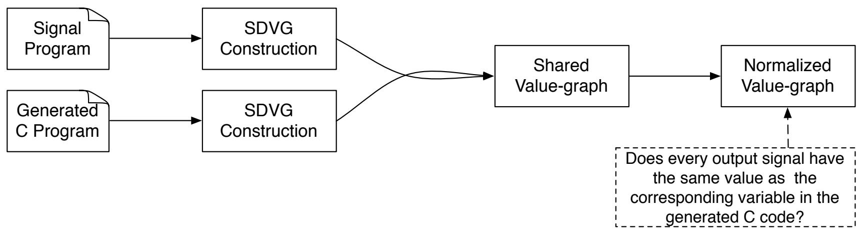

Fig. 8 SDVG translation validation

## 6.1 Synchronous Data-flow Value-Graph

Let $X$ be the set of variables used to denote the signals, clocks and variables in a Signal program and its generated C code, and $F$ the set of function symbols. Here, $F$ contains usual logic operators (`not`, `and`, `or`), numerical comparison functions (<, >, =, <=, >=, /=), numerical operators (+, -, *, /), and gated $\phi$-function [48]. A gated $\phi$-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 9** A SDVG (Synchronous Data-flow Value-Graph) associated with a Signal program and its generated C code is a directed graph $G = \langle N, E, l_N, m_N \rangle$, 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; $l_N : N \longrightarrow X \cup F$ 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. The mapping $m_N$ defines a set of equivalent clocks, signals and variables.

A subgraph rooted at a given node is used to describe the computation of the corresponding element labeled (with $l_N$) at this node. A node $s$ labeled by $y$ with the set of clocks, signals or variables $m_N(s) = \{x_0, \ldots, x_n\}$ will be denoted as a node with label $\{x_0, \ldots, x_n\}\ y$.

### *6.1.1 SDVG of a Signal Program*

Let $P$ be a Signal program, we write $X = \{x_1, \ldots, x_n\}$ to denote the set of all signals in $P$, consisting of input, output, state (corresponding to delay operator) and local signals, denoted by $I, O, S$ and $L$, respectively. Recall that for each $x_i \in X$, $\mathbb{D}_{x_i}$ denotes its domain of values and $\mathbb{D}^{\perp}_{x_i}$ its domain of values extended with the absent value. Each signal $x_i$ is associated with a Boolean variable $\widehat{x_i}$ to encode its clock at a given instant

$t$ and $\widetilde{x_i}$ with the same type as $x_i$ to encode its value. Formally, the abstract values representing the clock and value of a signal can be denoted using a gated $\phi$-function: $x_i = \phi(\widehat{x_i}, \widetilde{x_i}, \perp)$.

Assume that the computations of signals in processes $P_1$ and $P_2$ are represented as shared value-graphs $G_1$ and $G_2$, respectively. Then the value-graph $G$ of the synchronous composition process $P_1|P_2$ can be defined as $G = \langle N, E, l_N, m_N \rangle$ obtained from $G_1$ and $G_2$ by replacing any node labeled by some $x$ by the subgraph that is rooted by the node labeled by $x$ in $G_1$ or $G_2$. Every identical subgraph is reused, in other words, 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.

Thus the SDVG of a Signal program is obtained from the subgraphs associated with each primitive operator.

*Stepwise Functions* $y := f(x_1, \ldots, x_n)$. The computation of $y$ can be represented by the following gated $\phi$-function: $y = \phi(\hat{y}, f(\widetilde{x_1}, \widetilde{x_2}, \ldots, \widetilde{x_n}), \perp)$, where $\hat{y} \Leftrightarrow \widehat{x_1} \Leftrightarrow \widehat{x_2} \Leftrightarrow \ldots \Leftrightarrow \widehat{x_n}$. The graph representation of the stepwise functions is depicted in Fig. 9 (left). Note that in the graph, the node labeled by $\{\widehat{x_1}, \ldots, \widehat{x_n}\}\ \hat{y}$ means that $m_N(\hat{y}) = \{\widehat{x_1}, \ldots, \widehat{x_n}\}$. In other words, the subgraph representing the computation of $\hat{y}$ is also the computation of $\widehat{x_1}, \ldots,$ and $\widehat{x_n}$.

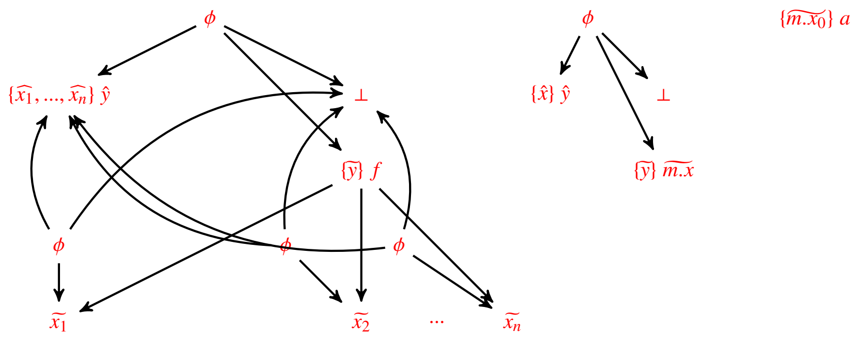

Fig. 9 The graphs of $y := f(x_1, \ldots, x_n)$ and $y := x\$1$ `init` $a$

*Delay* $y := x\$1$ `init` $a$. 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 respectively the last value of $x$ and the initial value of $y$. The graph representation is depicted in Fig. 9 (right).

*Sampling* $y := x$ `when` $b$. The computation of $y$ can be represented by the following node: $y = \phi(\hat{y}, \widetilde{x}, \perp)$, where $\hat{y} \Leftrightarrow (\hat{x} \wedge \hat{b} \wedge \widetilde{b})$. Fig. 10 (right) shows its graph representation.

*Merge* $y := x$ `default` $z$. The computation of $y$ can be represented by the following node: $y = \phi(\hat{y}, \phi(\hat{x}, \widetilde{x}, \widetilde{z}), \perp)$, where $\hat{y} \Leftrightarrow (\hat{x} \vee \hat{z})$. The graph representation is depicted in Fig. 10 (left). Note that in the graph, the clock $\hat{y}$ is represented by the subgraph of $\hat{x} \vee \hat{z}$.

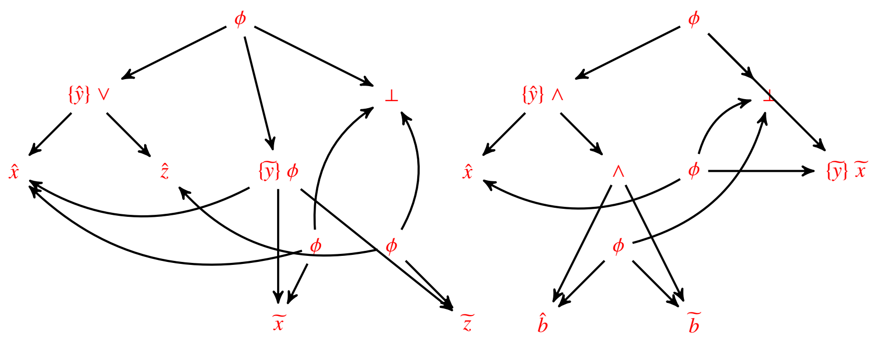

Fig. 10 The graphs of $y := x$ `default` $z$ and $y := x$ `when` b

*Restriction.* The graph representation of the restriction process $P$ `where` $x$ is the same as the graph of $P$.
*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 $\phi$-function $z = \phi(\hat{z}, true, \perp)$, 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` $b$. The clock relation $x\ \hat{}{=}\ y$ is represented by a single node graph labeled by $\{\hat{x}\}\ \hat{y}$.

### 6.1.2 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 denote $X_A = \{x_1, \dots, x_n\}$ the set of all signals in `A`, and $X_C = \{x_1^c, \dots, x_m^c\}$ the set of all variables in `C`. We add "c" as superscript for the variables, to distinguish them from the signals in `A`. As described in [8,42,25,4], the generated C code of `A` consists of the following files:

– `A_main.c` is the implementation of the *main function*. It opens the IO communication channels by calling functions provided in `A_io.c`, 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. The generated C code in the step function consists only of assignments and `if-then` statements. For each signal named $x$ in `A`, there is 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 read operation, which gets the value of $x$ from the environment. If $x$ 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 implies that we have to detect these instants, at which the value of a variable in the C code is not updated. In this case, it will be assigned the absent value, $\perp$. Thus, the computation of a variable $x^c$ can fully be represented by a gated $\phi$-function $x^c = \phi(C\_x^c, \widetilde{x^c}, \perp)$, where $\widetilde{x^c}$ denotes the newly updated value of the variable.

A particular case is that of the computation of signals whose clock is the *master clock*, which ticks every time the step function is called. In the generated C code, they are implemented using the following forms:

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

This is also the case for some local C variables introduced by the Signal compiler. The computation of such variables can be represented by a single node graph labeled by $\{\widetilde{x^c}\}\ x^c$. That means that 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 considered 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 $\phi$-function: $y^c = \phi(C\_y^c, m.x^c + 1, \perp)$.

## 6.2 Translation Validation of SDVG

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 this normalization, for any output signal $x$ and its corresponding variable $x^c$ in the generated C code, we check whether $x$ and $x^c$ point to the same node in the resulting graph (formally, there exists one node $n$ such that $x, x^c \in m_N(n)$). The normalizing procedure can be adapted with any future optimization of the compiler by updating the set of rewrite rules.

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 4. The normalizing algorithm indicates that we apply the rewrite rules to each graph node individually. When there are 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.

```
// 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 $\exists r \in R$ that can be applied on $G$) {
  while ($\exists r \in R$ that 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 $G$
```

Listing 4 The normalizing algorithm

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 the form of term rewrite rules, $t_l \rightarrow t_r$, 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 here a part of the rules.

*General Simplification Rules.* These rules are related to the general rules of inference of operators, denoted by the corresponding function symbols in $F$. When applying them, we will replace a subgraph rooted at some node by a smaller subgraph. In consequence of this replacement, we will reduce the number of nodes by eliminating some unnecessary structures. A first set of rules simplifies numerical and Boolean comparison expressions. In these rules, the subgraph $t$ represents some 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 labeled by the value true.

$$= (t, t) \rightarrow \texttt{true}$$
$$\neq (t, t) \rightarrow \texttt{false}$$

A second set of general simplification rules eliminates unnecessary nodes that represent the $\phi$-functions. For instance, we consider the following rules (where $c$ is a Boolean expression):

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

The first rule replaces a $\phi$-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 $\phi$-function can be replaced with the value of the condition $c$. Consider a $\phi$-function such that one of its branches is another $\phi$-function: the third rule removes the $\phi$-function in the branches if the conditions of the $\phi$-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 reflects 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. There is for example a set of rules for simplifying constant expressions such as:

$$\begin{aligned}
&+(cst_1, cst_2) \to cst, \text{ where } cst = cst_1 + cst_2\\
&\wedge(cst_1, cst_2) \to cst, \text{ where } cst = cst_1 \wedge cst_2\\
&\Box(cst_1, cst_2) \to cst
\end{aligned}$$

where $\Box$ denotes a numerical comparison function, 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 *rules of inference* for propositional logic. For example, we have a group of laws for rewriting formulas with `and` operator, such as:

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

*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.

A 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.

Consider an 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 $m.z^c$. A second rule in our synchronous rules 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 read 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.

## 6.3 Illustrative Example

Let us illustrate the verification process (Fig. 8) on the program `DEC` and its generated C code `DEC_step()` (Listing 5).

```
1   EXTERN logical DEC_step() {
2     C_FB = N <= 1;
3     if (C_FB) {
4       if (!r_DEC_FB(&FB)) return FALSE; % read input FB %
5     }
6     if (C_FB) N = FB; else N = N - 1;
7     w_DEC_N(N); % write output N %
8     DEC_step_finalize();
9     return TRUE;
10  }
```

Listing 5  Generated C code of `DEC`

In a 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. 11. Note that in the C program, the variable $N^c$ ("c" is added as

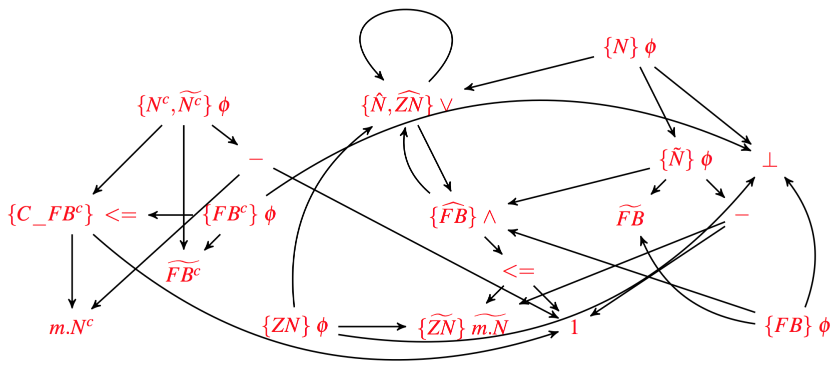

Fig. 11 The shared value-graph of `DEC` and `DEC_step`

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`.

In a second step, we shall normalize the above initial graph. Below is a potential normalization scenario, meaning that it 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 in different order.

1. 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`.
2. By rule $\wedge(\texttt{true}, x) \to x$, the node $\{\widehat{FB}\} \wedge$ is rewritten into $\{\widehat{FB}\}$ <=.
3. The $\phi$-function node representing the computation of $N$ is removed and $N$ points to the node $\{\widetilde{N}\}\ \phi$.
4. The $\phi$-function node representing the computation of $ZN$ is removed and $ZN$ points to the node $\{\widetilde{ZN}\}\ \widetilde{m.N}$.
5. 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 $\{\widetilde{FB}\}\ \widetilde{FB^c}$.
6. 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 $\{\widetilde{m.N}\}\ m.N^c$.

Fig. 12 depicts an intermediate resulting graph of this normalization scenario (after step 4), and Fig. 13 is the final normalized graph when we cannot perform any more normalization. In the final step, we check that the value of the output signal and its

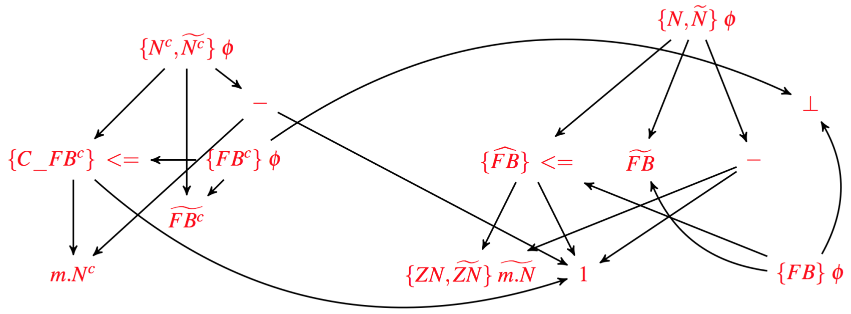

Fig. 12 Intermediate resulting value-graph of `DEC` and `DEC_step`

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.

## 7 Detected Bugs

So far, our validator has revealed three previously unknown bugs in the implementation of the Signal compiler. The first bug was introduced when multiple constraints

$$\begin{tikzcd}
 & & \{N^c, \widetilde{N^c}, N, \widetilde{N}\}\ \phi \arrow[dl] \arrow[d] \arrow[dr] & & \{FB^c, FB\}\ \phi \arrow[dlll] \arrow[dll] \arrow[d] \\
 & \{C\_FB^c, \widehat{FB}\}\ <= \arrow[dl] \arrow[d] & \{\widetilde{FB^c}\}\ \widetilde{FB} & - \arrow[dlll] \arrow[dll] & \perp \\
\{\widetilde{m.N}, ZN, \widetilde{ZN}\}\ m.N^c & 1 & & &
\end{tikzcd}$$

Fig. 13 The final normalized graph of `DEC` and `DEC_step`

condition a clock. The seconde problem is the wrong implementation of `xor` operator. The last one is a syntax error in the generated C code from a Signal program, in which a constant signal is used. In the following, we show how the validator captures these classes of bugs through concrete examples.

## 7.1 Clock with Multiple Constraints

We consider a clock $x$ that is defined in the Signal program `P` in Listing 6. The clock of `x` is conditioned by two constraints, `y` $\leq 9$ and `y` $\geq 1$. That means that `x` is present if and only if the signal `y` is present and holds a value in $[1,9]$.

```
% P.SIG %
| x ^= when (y <= 9)
| x ^= when (y >= 1)

% P_BASIC_TRA.SIG %
| CLK_x := when (y <= 9)
| CLK := when (y >= 1)
| CLK_x ^= CLK
| CLK ^= XZX_24

% P_BOOL_TRA.SIG %
| when Tick ^= C_z ^= C_CLK
| when C_z ^= x ^= z
| C_z := y <= 9
| C_CLK := y >= 1
```

Listing 6 Bug Example: Clock with Multiple Constraints

The clock calculation of the transformed programs `P_BASIC_TRA` and `P_BOOL_TRA` is also given in Listing 6. In the transformed program `P_BASIC_TRA`, the compiler introduces a fresh signal `XZX_24` that is synchronized with `CLK` and `CLK_x`. The introduction of the signal `XZX_24` and the synchronization between `CLK`, `CLK_x`, and `XZX_24` cause an incorrect specification of clocks: the signal `x` might be absent when `XZX_24` is absent, which is not the case in `P`, nor in `P_BOOL_TRA`).

This bug was caught by our validator from the *clock synthesis* phase of the compiler. It found that there does not exist a refinement between the clock models of P_BASIC_TRA and P_BOOL_TRA, or $\Phi$(P_BOOL_TRA) $\not\sqsubseteq_{clk}$ $\Phi$(P_BASIC_TRA).

## 7.2 XOR Operator

The second problem was the wrong implementation of the xor operator as shown in the program in Listing 7. In this program, the Boolean signal b3 is defined by the Boolean expression (true xor true) and b1, meaning that b3 is synchronized with b1 and whenever it is present it holds the value false. However, in the transformed program P_BASIC_TRA, b1 and b3 are synchronized but whenever b3 is present, it has the same value as b1.

```
% P.SIG %
| b3 := (true xor true) and b1

% P_BASIC_TRA.SIG %
| CLK_b1 := ^b1
| CLK_b1 ^= b1 ^= b3 | b3 := b1
```

Listing 7 Bug Example: Xor Operator

The validator has detected this bug in the *clock synthesis* phase of the compiler. It found that there does not exist a refinement between the clock models of P and P_BASIC_TRA, or $\Phi$(P_BASIC_TRA) $\not\sqsubseteq_{clk}$ $\Phi$(P).

## 7.3 Constant Signal

The last detected problem was not found by the translation validation itself but was indirectly discovered when trying to apply it on the *code generation* phase of the Signal compiler. It occurred in a program in which a *merge* operator with a constant signal was used, such as y := 1 default x.

```
% Version with bug %
if (C_y) {
   y = 1; else y = x;
   w_ClockError_y(y);
}

% Version without bug %
if (C_y) {
  if (C_y) y = 1; else y = x;
  w_ClockError_y(y);
}
```

Listing 8 Bug Example: Constant Signal

In this case, the compiler dealt wrongly with the *clock context* of a constant signal by introducing a syntax error in the generated C code. The bug and its fix are given in Listing 8. This bug was captured when the validator tried to construct the shared SDVG graph of the transformed Signal program at the end of the *static scheduling synthesis* phase and of the generated C program.

## 8 Conclusion

We have presented an approach to compiler verification that adopts translation validation to build a formally verified compiler verifier within the existing Signal toolset. Our approach focuses on the transformations performed by the compiler using the simplest structures to represent them: SAT/SMT formulas represent the refinement of clock models and the reinforcement of data-flow graphs, value-graphs are used to represent the production of target code patterns from and a specification's syntax tree. This reduces the whole process of proving a refinement relation between the source specification and the generated code to a couple of SMT SAT-checking on formulas of minimal size, and to a symbolic rewriting on a reduced graph to check value equivalence. Our validator does not modify or instrument the compiler. It treats it as a "black box" (as long as there is no error in it). It only considers an input program and its transformed result. Hence, it is not affected by an update or modification made to this or that compilation stage, as long as its principle and data structure remain the same. The validator is much simpler and smaller than the compiler itself. Proving its correctness (the model builder, the verifier) would take a lot less effort than for the compiler as well. Verification is fully automated and scales to large programs very well by employing state-of-the-art verification tools and by minimizing the representation of the problem to solve. For that purpose, we represent the desired program semantics using a scalable abstraction and we use efficient SMT libraries [19] to achieve the expected goals: traceability and formal evidence. We believe that this approach provides a, both technically and economically, attractive alternative to developing a certified compiler. The individual modules designed in the context of this project are being integrated in the open-source environment of the Eclipse project POP with the Polarsys Industry Working Group [54].

## References

1. ACE: Ace supertest suite. http://www.ace.nl/compiler/supertest.html (2013)
2. Ackerman, W.: Solvable cases of the Decision Problem. Studies in logic and the foundations of mathematics. North-Holland, Amsterdam (1954)
3. Astrée: The static program analyzer. http://www.astree.ens.fr/ (2014)
4. Aubry, P., Le Guernic, P., Machard, S.: 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)
5. Benveniste, A., Le Guernic, P.: Hybrid dynamical systems theory and the Signal language. IEEE Transactions on Automatic Control **35(5)**, 535–546 (1990)
6. Berry, G.: The foundations of Esterel. In Proof, Language and Interaction: Essays in Honor of Robin Milner, MIT Press (2000)
7. Besnard, L., Gautier, T., Le Guernic, P.: SIGNAL V4-INRIA version: Reference Manual (2010). http://www.irisa.fr/Polychrony/documentation.php
8. Besnard, L., Gautier, T., Le Guernic, P., Talpin, J.P.: Compilation of polychronous data-flow equations. In: Synthesis of Embedded Software, pp. 1–40. Springer (2010)
9. Biere, A., Heule, M., van Maaren, H., Walsh, T.: Handbook of Satisfiability: Volume 185 Frontiers in Artificial Intelligence and Applications. IOS Press, Amsterdam, The Netherlands (2009)
10. Blanchet, B., Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., Rival, X.: A static analyzer for large safety-critical software. In: Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation, PLDI '03, pp. 196–207 (2003)

11. Blazy, S.: Which C semantics to embed in the front-end of a formally verified compiler? In: Tools and Techniques for Verification of System Infrastructure (2008)
12. Blazy, S., Dargaye, Z., Leroy, X.: Formal verification of a C compiler front-end. In: Proceedings of the 14th International Conference on Formal Methods, Lecture Notes in Computer Science (LNCS 4085), pp. 460–475. Springer (2006)
13. Blazy, S., Robillard, B., Appel, A.: Formal verification of coalescing graph-coloring register allocation. In: 19th European Symposium On Programming (ESOP 2010), Lecture Notes in Computer Science (LNCS 6012), pp. 145–164. Springer (2010)
14. Börger, E., Grädel, E., Gurevich, Y.: The Classical Decision Problem. Spinger-Verlag (1996)
15. Brown, D., Delseny, H., Hayhurst, K., Wiels, V.: Guidance for using formal methods in a certification context. In: Embedded Real Time Software and Systems (ERTS$^2$) (2010)
16. Chirica, L.M., Martin, D.F.: Towards compiler implementation correctness proofs. ACM TOPLAS (1986)
17. CompCert: CompCert C verified compiler. http://compcert.inria.fr/partners.html (2014)
18. Coq-Inria: Coq proof assistant. http://coq.inria.fr/ (2014)
19. Dutertre, B., de Moura, L.: Yices SMT solver. http://yices.csl.ri.com (2009)
20. Feautrier, P., Gamatié, A., Gonnord, L.: Enhancing the compilation of synchronous data-flow programs with combined numerical-boolean abstraction. In CSI Journal of Computing **1(4)**, 86–99 (2012)
21. França, R.B., Blazy, S., Favre-Félix, D., Leroy, X., Pantel, M., Souyris, J.: Formally verified optimizing compilation in ACG-based flight control software. In: Embedded Real Time Software and Systems (ERTS$^2$) (2012)
22. Gamatié, A.: Designing Embedded Systems with the Signal Programming Language. Springer (2009)
23. Gamatié, A., Gautier, T., Le Guernic, P.: Towards static analysis of Signal programs using interval techniques. In: Synchronous Languages, Applications, and Programming (SLAP'06) (2006)
24. Gamatié, A., Gonnord, L.: Static analysis of synchronous programs in Signal for efficient design of multi-clocked embedded systems. In: ACM SIGPLAN/SIGBED Conference on Languages, Compilers, Tools and Theory for Embedded Systems, LCTES'2011 (2011)
25. Gautier, T., Le Guernic, P.: Code generation in the SACRES project. In: Towards System Safety, Proceedings of the Safety-critical Systems Symposium, pp. 127–149 (1999)
26. GeneAuto: GeneAuto project. www.geneauto.org (2014)
27. George, L., Appel, A.W.: Iterated register coalescing. TOPLAS **18(3)**, 300–324 (1996)
28. Halbwachs, N.: A synchronous language at work: the story of Lustre. In: 3th ACM-IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE'05) (2005)
29. Izerrouken, N., Kai, O.S.Y., Pantel, M., Thirioux, X.: Use of formal methods for building qualified code generator for safer automotive systems. In: Proceedings of the 1st Workshop on Critical Automotive applications: Robustness and Safety, pp. 53–56 (2010)
30. Izerrouken, N., Pantel, M., Thirioux, X.: Machine checked sequencer for critical embedded code generator. In: International Conference on Formal Engineering Methods (ICFEM 2009), pp. 521–540 (2009)
31. Jackson, D.: A direct path to dependable software. Communications of the ACM **52**(4), 78–88 (2009)
32. Johnson, D.B.: Finding all the elementary circuits of a directed graph. SIAM Journal on Computing **4**(1), 77–84 (1975)
33. Klein, G., Nipkow, T.: A machine-checked model for a Java-like language, virtual machine and compiler. TOPLAS **28**(4), 619–695 (2006)
34. Le Guernic, P., Gautier, T.: Data-flow to von Neumann: the Signal approach. In: Advanced Topics in Data-Flow Computing, pp. 413–438 (1991)
35. Le Guernic, P., Le Borgne, M., Gautier, T., Le Maire, C.: Programming real-time applications with Signal. Proceedings of the IEEE **79**(9), 1321–1336 (1991)
36. Ledinot, E., Pariente, D.: Formal methods and compliance to the DO-178C/ED12C standard in aeronautics. In: Static Analysis of Software, pp. 207–272. John Wiley & Sons (2012)
37. Leinenbach, D., Paul, W., Petrova, E.: Towards the formal verification of a C0 compiler: code generation and implementation correctness. In: Third IEEE International Conference on Software Engineering and Formal Methods (SEFM 2005) (2005)
38. Leroy, X.: Formal certification of a compiler back-end, or: Programming a compiler with a proof assistant. In: Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 42–54 (2006)
39. Leroy, X.: A formally verified compiler back-end. Journal of Automated Reasoning **43**(4), 363–446 (2009)

40. Leroy, X., Blazy, S.: Formal verification of a C-like memory model and its uses for verifying program transformations. Journal of Automated Reasoning **41**(1), 1–31 (2008)
41. Leviathan, R., Pnueli, A.: Validating software pipelining optimizations. In: Int. Conf. on Compilers, Architecture, and Synthesis for Embedded Systems (CASES 2002), pp. 280–287 (2006)
42. Maffeïs, O., Le Guernic, P.: 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, pp. 547–566 (1994)
43. Necula, G.C.: Translation validation for an optimizing compiler. In: Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation (PLDI'00), pp. 83–94 (2000)
44. Ngo, V.C., Talpin, J.P., Gautier, T.: Efficient deadlock detection for polychronous data-flow specifications. In: Electronic System Level Synthesis Conference (ESLsyn 2014) (2014)
45. Ngo, V.C., Talpin, J.P., Gautier, T.: Translation validation for synchronous data-flow specification in the Signal compiler. In: Formal Techniques for Distributed Objects, Components and Systems (FORTE 2015), LNCS 9039. Springer (2015)
46. Ngo, V.C., Talpin, J.P., Gautier, T., Le Guernic, P.: Translation validation for clock transformations in a synchronous compiler. In: International Conference on Fundamental Approaches to Software Engineering (FASE 2015), LNCS 9033. Springer (2015)
47. Ngo, V.C., Talpin, J.P., Gautier, T., Le Guernic, P., Besnard, L.: Formal verification of compiler transformations on polychronous equations. In: Proceedings of 9th International Conference on Integrated Formal Methods (IFM 2012), LNCS 7321. Springer (2012)
48. Ottenstein, K.J., Ballance, R.A., MacCabe, A.B.: 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, pp. 257–271 (1990)
49. Ottenstein, K.J., Ottenstein, L.M.: The program dependence graph in a software development environment. ACM SIGSOFT Software Engineering Notes **9(3)**, 177–184 (1984)
50. Pnueli, A., Shtrichman, O., Siegel, M.: Translation validation: From Signal to C. In: Correct System Design, Recent Insight and Advances, LNCS 1710, pp. 231–255. Springer-Verlag (2000)
51. Pnueli, A., Siegel, M., Singerman, E.: Translation validation. In: Proceedings of the 4th International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS '98), LNCS 1384, pp. 151–166. Springer-Verlag (1998)
52. Poetzsch-Heffter, A., Gawkowski, M.: Towards proof generating compilers. Electronic Notes in Theoretical Computer Science **132**(1) (2005)
53. Polak, W.: Compiler Specification and Verification, *Lecture Notes In Computer Science*, vol. 124. Springer-Verlag (1981)
54. Polarsys project POP: Polychrony on Polarsys. `https://www.polarsys.org/projects/polarsys.pop` (2015)
55. Rinard, M.: Credible compilers. Tech. Rep. 776, Massachusetts Institute of Technology (1999)
56. Rushby, J.: New challenges in certification for aircraft software. In: Proceedings of the Ninth ACM International Conference On Embedded Software (EMSOFT) (2011)
57. Stump, A., Deters, M.: SMT-Comp. `http://www.smtcomp.org/2009` (2009)
58. Tate, R., Stepp, M., Tatlock, Z., Lerner, S.: Equality saturation: A new approach to optimization. In: Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 264–276 (2009)
59. Thatcher, J.W., Wagner, E.G., Wright, J.B.: More on advice on structuring compilers and proving them correct. In: Semantics-Directed Compiler Generation, *Lecture Notes In Computer Science*, vol. 94 (1980)
60. Tristan, J.B., Govereau, P., Morrisett, G.: Evaluating value-graph translation validation for LLVM. In: Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation (2011)
61. Tristan, J.B., Leroy, X.: A simple, verified validator for software pipelining. In: Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 83–92 (2010)
62. Zuck, L., Pnueli, A., Fang, Y., Goldberg, B.: VOC: A translation validator for optimizing compilers. Electronic Notes in Theoretical Computer Science **65(2)** (2002)
63. Zuck, L., Pnueli, A., Leviathan, R.: Validation of optimizing compilers. Tech. Rep. MCS01-12, Weizmann Institute of Science (2001)