RESEARCH ARTICLE

Formal Verification of Synchronous Data-flow Program Transformations Toward Certified Compilers

Van Chan Ngo (✉)[1]**, Jean-Pièrre Talpin** [1]**, Thierry Gautier** [1]**,**
Paul Le Guernic [1]**, and Loïc Besnard** [2]

1   INRIA Rennes-Bretagne Atlantique, Rennes 35042, France
2   IRISA/CNRS, Rennes 35042, France

Abstract   Translation validation was invented in the 90's by Pnueli et al. as a technique to formally verify the correctness of code generators. Rather than certifying the code generator or exhaustively qualifying it, translation validators attempt to verify that program transformations preserve semantics. In this work, we adopt this approach to formally verify that the clock semantics and data dependence are preserved during the compilation of the Signal compiler. Translation validation is implemented for every compilation phase from the initial phase until the latest phase where the executable code is generated, by proving the transformation in each phase of the compiler preserves the semantics.

We represent the clock semantics, the data dependence of a program and its transformed counterpart as first-order formulas which are called *Clock Models* and *Synchronous Dependence Graphs* (SDGs), respectively. We then introduce *clock refinement* and *dependence refinement* relations which express the preservations of clock semantics and dependence, as a relation on clock models and SDGs, respectively. Our validator does not require any instrumentation or modification of the compiler, nor any rewriting of the source program.

1   Introduction

The synchronous languages such as Esterel, Lustre, and Signal [8, 17, 24] have been introduced and successfully used to design and implement embedded and critical real-time systems. They have associated compilers, which transform, optimize, and generate code in some general-purpose programming language. Their compilation involves many analyzes, and program transformations. Some transformations may introduce additional information or constraints, to refine the meaning, and/or specialize the behavior of the original program, such as optimization or static scheduling. Thus, the complexity of these compilers increases the risk that their large-scale use may yield bugs. In a safety-critical framework, it is naturally required that the compiler must be formally verified as well to ensure that the source program semantics is preserved.

To circumvent compiler bugs, one can entirely rewrite the compiler with a theorem proving tool such as Coq [14], or check that it is compliant to DO-178C documents [38]. However, these solutions yield a situation where any change of the compiler (e.g. further optimization and update) means redoing the proof. Another approach, which provides ideal separation between the tool under verification and its checker, is trying to verify that the output and the input have the same semantics. In this aim, *translation validation* was introduced in the 90's by Pnueli et al. [35, 36], as a technique to formally verify correctness of code generators. Translation validators can be used to ensure that program transformations do not

introduce semantic discrepancies, or to help debugging the compiler implementation. Some other works have adopted the translation validation approach in verification of transformations, and optimizations. In [13, 31], the programs before and after the transformations and optimizations of a C compiler are represented in a common intermediate form, then the preservation of semantics is checked by using symbolic execution and the Coq proof assistant.

A compiler generally involves several phases during its compilation process. For instance, the Signal compiler, in its first two phases: *calculates the clock information*, makes *Boolean abstraction*, and makes *static scheduling*. The final phase is the executable code generation. One can try to prove globally that the input program and its final transformed program have the same semantics. However, we believe that a better approach consists in separating the concerns and proving for each phase the preservation of different kinds of semantic properties. In this case of the Signal compiler, the preservation of the semantics can be decomposed into the preservation of clock semantics, data dependence, and value-equivalence of variables. As first contribution to this work, this paper focuses on proving the preservation of clock semantics in the first phases of the Signal compiler. The preservation of clock semantics described in the present contribution will be used to verify the value-equivalence between data-flows in the source program and its generated code. Thanks to clock semantics preservation, the evaluation of a *normalizing* value-graph [41], used for that purpose, will be more efficient and faster. Moreover, the encoding of clock information considered here will be reused in order to represent the dependence graph of the synchronous programs for studying the preservation of data dependence which is considered as the other contribution of this work.

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 the program) manipulated by the specification at a given instant. Given two clock models, a *correct transformation* relation between them is defined, which expresses the semantic preservation of clock information. In the implementation, we apply our translation validation to the first two transformation steps of the compiler.

With the similar approach, the dependence in the source program and its transformed counterpart is represented by the formal structure, called *Synchronous Dependence Graph*

(SDG). A SDG for a given program is a labelled directed graph in which each node is a signal or a clock and each edge represents a dependence between nodes. Each edge is labeled by a clock expression called *clock constraint* at which the dependence between two extremity nodes is effective. Given two SDGs, a *correct transformation* relation between them is defined which expresses the semantic preservation of data dependence. In implementation, a SMT-solver is used for checking the existence of the correct transformation relations. We apply this validation to the second transformation steps of the compiler, *static scheduling*.

At a high level, our tool works as follows. For each transformation, it takes the input program and its transformed counterpart, and constructs the corresponding clock models, SDGs. Then it delegates the existence checking of the correct transformation relation to a solver. If the result is that the relation does not exist then a "compiler bug" message is emitted. Otherwise, the compiler continues its work. We believe that our validator must have the following features to be effective and realistic. First, we do not modify or instrument the compiler, and we treat the compiler as a "black box". Hence the validator is not affected by some future update or modification of the compiler. We only need some additional information about the mapping between original names and potential new names of local variables. Our approach consists in applying formal methods to the compiler transformations themselves in order to automatically generate formal evidence that the clock semantics of the source program is preserved during program transformations, as per applicable qualification standard. Second, it is important that the validator can be scaled to large programs. For this purpose, we represent the desired program semantics using a scalable abstraction and we use efficient SMT (Satisfiability Modulo Theory) libraries [15] to achieve the expected goals: traceability and formal evidence. Moreover, this approach provides an attractive alternative to develop a certified compiler for a synchronous language. Since in general the validator is much smaller and easier to verify than the compiler it validates.

The remainder of this paper is organized as follows. Section 2 introduces the Signal language. Section 3 presents the abstraction that represents the clock constraints in terms of first-order logic formulas. The definition and properties of SDGs are detailed in Section 4. In Section 5, we consider the definitions of correct transformation on clocks and SDGs which formally prove the conformance between the original specification and its compiled counterpart w.r.t the clock semantic and the data dependence. It also addresses the application of the verification process to the Signal compiler, and

its integration into the Polychrony toolset [34] in Section 6. Section 7 presents some related works, concludes our work and outlines future directions.

2 The Signal Language

Signal [10, 22] is a polychronous data-flow language that allows the specification of multi-clocked systems. Signal handles unbounded series of typed values $(x(t))_{t \in \mathbb{N}}$, called *signals*, denoted as x. Each signal is implicitly indexed by a logical *clock* indicating the set of instants at which the signal is present, noted C_x. At a given instant, a signal may be present where it holds a value, or absent (denoted by #).

2.1 Language Features

Syntax. In Signal, a process (written P or Q) consists of the synchronous composition (noted $P|Q$) of equations over signals x, y, z, written $x := y f z$ or $x := f(y, z)$. The process P/x restricts the lexical scope of the signal x to the process P. An equation $x := y f z$ defines the output signal x by the result of the application of operator f to its inputs y, z.

$$P, Q ::= x := y f z \mid P|Q \mid P/x$$

Semantic Domains. For a set of values (a type) $\mathbb{D}$ we define its extended set $\mathbb{D}_\# = \mathbb{D} \cup \{\#\}$, where $\# \notin \mathbb{D}$ is a special symbol used to denote the absence of an occurrence in the signal. $\mathbb{D}_\#$ is flat. We denote by $\mathbb{D}^\infty = \mathbb{D}^* \cup \mathbb{D}^\omega$ the set of finite and infinite sequences of "values" in $\mathbb{D}_\#$. ϵ denotes the empty sequence. All signal functions $f : \mathbb{D}_1^\infty \times ... \times \mathbb{D}_n^\infty \to \mathbb{D}_{n+1}^\infty$ are defined using the following conventions: $x, y, z...$ are signals, $v_1, ..., v_n$ are values in $\mathbb{D}_i$ (cannot be #), $v_1^\#, ..., v_n^\#$ are values in $\mathbb{D}_{i\#}$, and $x.y$ is the concatenation of two sequences x and y. Signal functions are total, strict and continuous functions over domains [3] (w.r.t prefix order) that satisfy the following general rules:

- $f(\#.x_1, ..., \#.x_n) = \#.f(x_1, ..., x_n)$
- $f(x_1, ..., x_n) = \epsilon$ when for some i, $x_i = \epsilon$

A function is *synchronous* iff it satisfies:

- $f(v_1^\#.x_1, ..., v_n^\#.x_n) = \epsilon$ when $v_i^\# = \#$ and $v_j^\# \neq \#$ for some i, j

Stepwise Extension ($y := f(x_1, ..., x_n)$). Given $n > 0$ and a n-ary total function $f : \mathbb{D}_1 \times ... \times \mathbb{D}_n \to \mathbb{D}_{n+1}$, the *stepwise extension* of f denoted F is the synchronous function that satisfies:

- $F(v_1.x_1, ..., v_n.x_n) = f(v_1, ..., v_n).F(x_1, ..., x_n)$

Previous Value ($y := x\$1 \text{ init } a$). The $\$: \mathbb{D}_i \times \mathbb{D}_i^\infty \to \mathbb{D}_i^\infty$ is the synchronous (state) function that satisfies

- $\$(v_{-1}, v.x) = v_{-1}.\$(v, x)$

Deterministic Merge ($y := x \text{ default } z$). The $\text{default}: \mathbb{D}_{i\#}^\infty \times \mathbb{D}_{i\#}^\infty \to \mathbb{D}_{i\#}^\infty$ signal function is recursively defined by

- for $v \in \mathbb{D}_i$, $\text{default}(v.x, v^\#.z) = v.\text{default}(x, z)$.
- $\text{default}(\#.x, v^\#.z) = v^\#.\text{default}(x, z)$.

Boolean Sampling ($y := x \text{ when } b$). The $\text{when}: \mathbb{D}_{i\#}^\infty \times \mathbb{B}_\#^\infty \to \mathbb{D}_{i\#}^\infty$ signal function is recursively defined by

- for $b^\# \in \mathbb{B}_\#$, $b^\# \neq \text{true} \text{ when}(v^\#.x, b^\#.b) = \#.\text{when}(x, b)$
- $\text{when}(v^\#.x, \text{true}.b) = v^\#.\text{when}(x, b)$

A network of strict, continuous signal functions that satisfies the Kahn conditions is a strict, continuous signal function or Kahn Process Network [26].

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

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

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

Let us comment this program: $?$ integer FB; $!$ integer N: FB, N are respectively input and output signals of type *integer*; $FB \hat{} = \text{ when } (ZN <= 1)$: FB is accepted (or it is present) only when ZN becomes less than or equal to 1; $N := FB \text{ default } (ZN - 1)$: N is set to FB when its previous value is less than or equal to 1, otherwise it is decremented by 1; $ZN := N\$1 \text{ init } 1$: defines ZN as always

carrying the previous value of *N* (the initial value of *ZN* is 1); `where integer ZN init 1`: indicates that *ZN* is a local signal whose initial value is 1. Note that the clock of the output signal is more frequent than that of the input. This is illustrated in the following possible trace:

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

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

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

2.2   Clock Constraints and Dependence

The above basic processes induce implicitly the clock constraints and dependence between the signals. Table 1 shows these clock constraints for the primitive operators. In this table, the sub-clock [c] (resp. [¬c]) is defined as $\{t \in C_c | c(t) = $ `true`$\}$ (resp. $\{t \in C_c | c(t) = $ `false`$\}$). Notice that a clock can be viewed as a signal with type *event* (which has only one value, `true`, when it is present), thus the condition C_c means that the signal *c* is present.

Let *x, y* be two signals and *c* an *event* or Boolean signal, if at any instant *t* such that $t \in C_x \cap C_y \cap C_c$ and $c(t) = $ `true`, setting a value to *y* cannot precede the availability of *x*, then we say that *y* depends on *x* at the condition *c*. We use $x \xrightarrow{c} y$ to denote the fact that there is a dependence between *y* and *x* at the condition *c*. Table 1 shows the dependence for the core language. In particular, the following dependence applies equally: i) Any signal is preceded by its clock. ii) For a Boolean signal *c*, [c] and [¬c] depend on *c*. iii) Any

	Dependence	Clock Constraint
x	$C_x \xrightarrow{C_x} x$	
c (Boolean signal)	$c \xrightarrow{C_c} [c]$ $c \xrightarrow{C_c} [\neg c]$	
$x \xrightarrow{c} y$	$[c] \xrightarrow{C_c} y$	
$y := f(x_1, ..., x_n)$	$x_1 \xrightarrow{C_y} y$ ... $x_n \xrightarrow{C_y} y$	$C_y = C_{x_1}$ ... $C_y = C_{x_n}$
$y := x\$1$ init *a*	$y \xrightarrow{C_y} x$	$C_y = C_x$
$y := x$ when *b*	$x \xrightarrow{C_y} y$ $b \xrightarrow{C_y} C_y$	$C_y = C_x \cap [b]$
$y := x$ default *z*	$x \xrightarrow{C_x} y$ $z \xrightarrow{C_z \backslash C_x} y$	$C_y = C_x \cup C_z$

Table 1   The Clock Constraints and Dependence

dependence $x \xrightarrow{c} y$ implies implicitly a dependence $[c] \xrightarrow{C_c} y$. As an example, for the basic process corresponding to the primitive operator *Boolean sampling*, the clock constraints and dependence between signals are given by:

- The clock of *y* is the intersection of the clock of *x* and the sub-clock [b].
- The signal *y* depends on the signal *x* whenever *y* is present.
- The clock C_y depends on the Boolean signal *b* whenever *y* is present.

2.3   Compilation of Signal Programs

The Signal compiler [9] consists of a sequence of code transformations. Some transformations are optimizations that rewrite the code to eliminate inefficient expressions. The compilation process may be seen as a sequence of morphisms rewriting Signal programs to Signal programs. The final steps (C or Java code generation) are simple morphisms over the ultimately transformed program. For convenience, the transformations of the compiler are divided into three phases as depicted in Figure 1. The optimized final program *_SEQ_TRA is translated directly to executable code. Signal programs
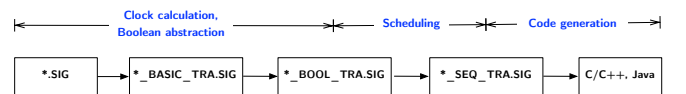

Fig. 1   The compilation of Signal compiler

which are produced in the first two phases (*clock calculation,*

Boolean abstraction and *static scheduling*) have the following features:

- The transformed programs are also written in Signal language.
- The clocks of all signals have been calculated and the overall set of clocks is organized as a *clock hierarchy* which is a set of clock trees [9]. When there is a single clock tree, the process has a fastest rated clock and it is said *endochronous*. When there are several clock trees, the process may be *endochronized* with an explicit parameterization, adding a fastest clock, *Tick*.
- In the successive transformations of the compiler, clocks are first represented as *event* signals related through clock specific Signal operators (this is reflected in the `*_BASIC_TRA` intermediate form); then clocks are transformed into Boolean signals defined with Boolean operators (this is reflected in the `*_BOOL_TRA` intermediate form).
- The scheduling information is represented in the `*_SEQ_TRA` intermediate form.
- The arithmetic expressions are leaved intact.

As an example, the body of the intermediate form `DEC_BASIC_TRA` obtained by compiling the above `DEC` process is as follows:

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

3 Clock Models

In this section, we describe the timing semantics of a program in terms of a first-order logic formula. Let us consider the semantics of the sampling operator $y := x$ when b. At any instant, the signal y holds the value of x if the following conditions are satisfied: x holds a value, and b is present and holds the value `true`; otherwise, it holds no value. Thus, to represent the underlying control conditions, we need to model the statuses *present* with value `true` or `false` and *absent* for the signal b, and the statuses *present* and *absent* for the signal x. This section explores a method to construct the control

model of a program as an abstraction of the clock semantics, called *clock model*, which is the computational model of our translation validation approach.

3.1 Illustrative Example

In Signal, clocks play a much more important role than in other synchronous languages, they are used to express the underlying control (i.e., the synchronization between signals) for any conditional definition. This differs from Lustre, where all clocks are built by sampling the fastest clock. For instance, we consider again the basic process corresponding to the primitive operator *Boolean sampling*, where x and y are numerical signals, and b is a Boolean signal: $y := x$ when b. To express the control, we need to represent the status of the signals x, y and b at a given instant. In this example, we use a Boolean variable $\hat{x}$ to capture the status of x: ($\hat{x} = $ `true`) means x is present, and ($\hat{x} = $ `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 $\bar{b}$ are used to represent its status: ($\hat{b} = $ `true` $\wedge$ $\bar{b} = $ `true`) means b is present and holds a value `true`; ($\hat{b} = $ `true` $\wedge$ $\bar{b} = $ `false`) means b is present and holds a value `false`; and ($\hat{b} = $ `false`) means b is absent.

Hence, at a given instant, the implicit control relations of the basic process above can be encoded by the following formula:

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

3.2 Abstraction

Let $X = \{x_1, ..., x_n\}$ be the set of all signals in program P. With each signal x_i, we attach a Boolean variable $\widehat{x_i}$ to encode its clock and a variable $\overline{x_i}$ of same type as x_i to encode its value. Formally, the abstract values which represent the clock semantics of the program can be computed using the following functions:

- $\hat{\ }$: $X \longrightarrow \mathbb{B}$ associates a signal with a Boolean value;
- $\bar{\ }$: $X \longrightarrow \mathbb{D}$ associates a signal with a value of same type as the signal.

The composition of Signal processes corresponds to logical conjunctions. Thus clock model of P will be a conjunction $\Phi(P) = \bigwedge_{i=1}^{n} \phi(eq_i)$ whose atoms are $\widehat{x_i}, \overline{x_i}$, where $\phi(eq_i)$ is the abstraction of statement eq_i (statement using the Signal primitive operators), and n is the number of statements in the program. In the following, we present the abstraction corresponding to each Signal operator.

3.2.1   Stepwise Extensions

The functions which apply on signal values in the primitive *stepwise extensions* are usual logic operators (*not, and, 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. Every variable is determinable by the inputs, memorizable values, otherwise program can not be compiled. This suggests that most of the implications will hold independently of the features of the numerical comparison functions and numerical operators and we can replace the operations by *uninterpreted functions*. By following the encoding procedure of [1], for every numerical comparison functions and numerical operator (denoted by $\square$) occurring in an equation, we perform the following rewriting:

- Replace each $x \square y$ by a new variable $v_\square^i$ of a type equal to that of the value returned by $\square$. Two stepwise functions $x \square y$ and $x' \square y'$ are replaced by the same variable $v_\square^i$ iff x, y are identical to x' and y', respectively.
- For every pair of newly added variables $v_\square^i$ and $v_\square^j$, $i \neq j$, corresponding to the non-identical occurrences $x \square y$ and $x' \square y'$, add the implication $(\overline{x} = \overline{x'} \wedge \overline{y} = \overline{y'}) \Rightarrow \overline{v_\square^i} = \overline{v_\square^j}$ into the abstraction $\Phi(P)$.

The abstraction $\phi(y := f(x_1, ..., x_n))$ of stepwise functions is defined by induction as follows:

- $\phi(\texttt{true}) = \texttt{true}$ and $\phi(\texttt{false}) = \texttt{false}$.
- $\phi(y := x) = (\hat{y} \Leftrightarrow \hat{x}) \wedge (\hat{y} \Rightarrow (\bar{y} \Leftrightarrow \bar{x}))$ if x and y are Boolean. $\phi(y := x) = (\hat{y} \Leftrightarrow \hat{x}) \wedge (\hat{y} \Rightarrow (\bar{y} \Leftrightarrow \bar{x})) \wedge (\hat{x} \Rightarrow \bar{x})$ if x is an *event* signal.
- $\phi(y := x_1 \ and \ x_2) = (\hat{y} \Leftrightarrow \widehat{x_1} \Leftrightarrow \widehat{x_2}) \wedge (\hat{y} \Rightarrow (\bar{y} \Leftrightarrow \overline{x_1} \wedge \overline{x_2}))$.
- $\phi(y := x_1 \ or \ x_2) = (\hat{y} \Leftrightarrow \widehat{x_1} \Leftrightarrow \widehat{x_2}) \wedge (\hat{y} \Rightarrow (\bar{y} \Leftrightarrow \overline{x_1} \vee \overline{x_2}))$.
- $\phi(y := x_1 \ \square \ x_2) = (\hat{y} \Leftrightarrow \widehat{v_\square^i} \Leftrightarrow \widehat{x_1} \Leftrightarrow \widehat{x_2}) \wedge (\hat{y} \Rightarrow (\bar{y} = \overline{v_\square^i}))$.

3.2.2   Previous Value

Considering the *previous value* operator, $y := x\$1 \ \texttt{init} \ a$, its encoding $\phi(y := x\$1 \ \texttt{init} \ a)$ contributes to $\Phi(P)$ with the following conjunct:

- if x, y and a are Boolean:
$(\hat{y} \Leftrightarrow \hat{x})$
$\wedge (\hat{y} \Rightarrow ((\bar{y} \Leftrightarrow m.x) \wedge (m.x' \Leftrightarrow \bar{x})))$
$\wedge (m.x_0 \Leftrightarrow a)$
- if x, y and a are non-Boolean:
$(\hat{y} \Leftrightarrow \hat{x})$

This encoding requires that at any instant, signals x and y have the same status (present or absent). If the signals are Boolean, it encodes the value of the output signal as well. Here, 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 $m.x_0$.

3.2.3   Deterministic Merge

The encoding of the *deterministic merge* operator, $y := x$ default z, contributes to $\Phi(P)$ with the following conjunct:

- if x, y and z are Boolean:
$(\hat{y} \Leftrightarrow (\hat{x} \vee \hat{z}))$
$\wedge \hat{y} \Rightarrow ((\hat{x} \wedge (\bar{y} \Leftrightarrow \bar{x}))$
$\vee (\neg\hat{x} \wedge (\bar{y} \Leftrightarrow \bar{z}))))$
- if x, y and z are non-Boolean:
$\hat{y} \Leftrightarrow (\hat{x} \vee \hat{z})$

3.2.4   Boolean Sampling

The encoding of the *Boolean sampling* operator, $y := x$ when b, contributes to $\Phi(P)$ with the following conjunct:

- if x and y are Boolean:
$(\hat{y} \Leftrightarrow (\hat{x} \wedge \hat{b} \wedge \bar{b}))$
$\wedge (\hat{y} \Rightarrow (\bar{y} \Leftrightarrow \bar{x}))$
- if x and y are non-Boolean:
$\hat{y} \Leftrightarrow (\hat{x} \wedge \hat{b} \wedge \bar{b})$

3.2.5   Composition

Consider the composition of two processes P_1 and P_2. Its abstraction $\phi(P_1|P_2)$ is defined as follows:

- $\phi(P_1) \wedge \phi(P_2)$

3.2.6   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*:

- $\phi(z := \hat{\ }x) = (\hat{z} \Leftrightarrow \hat{x}) \wedge (\hat{z} \Rightarrow \bar{z})$
- $\phi(x\hat{\ }= \ y) = \hat{x} \Leftrightarrow \hat{y}$
- $\phi(z := x\hat{\ }+ \ y) = (\hat{z} \Leftrightarrow (\hat{x} \vee \hat{y})) \wedge (\hat{z} \Rightarrow \bar{z})$
- $\phi(z := x\hat{\ }* \ y) = (\hat{z} \Leftrightarrow (\hat{x} \wedge \hat{y})) \wedge (\hat{z} \Rightarrow \bar{z})$
- $\phi(z := x\hat{\ }- \ y) = (\hat{z} \Leftrightarrow (\hat{x} \wedge \neg\hat{y})) \wedge (\hat{z} \Rightarrow \bar{z})$
- $\phi(z := \ $ when $b) = (\hat{z} \Leftrightarrow (\hat{b} \wedge \bar{b})) \wedge (\hat{z} \Rightarrow \bar{z})$

3.2.7   Nested processes

Assume that a process P has a sub-process P_1, the abstraction $\Phi(P)$ is given by:

- $\phi(P) \wedge \phi(P_1)$
- For every equation in process P that involves an invocation of a sub-process such as $(y_1, ..., y_n) := P_1(x_1, ..., x_m)$, the following conjuncts are added, where i_h, o_k are the inputs and outputs of P_1:
$$\bigwedge_{k=1}^{n} (\widehat{y_k} \Leftrightarrow \widehat{o_k} \wedge \overline{y_k} \Leftrightarrow \overline{o_k}) \wedge \bigwedge_{h=1}^{m} (\widehat{x_h} \Leftrightarrow \widehat{i_h} \wedge \overline{x_h} \Leftrightarrow \overline{i_h})$$

Applying the abstraction rules above, the clock semantics of the Signal program `DEC` is represented by the following first-order logic formula $\Phi(\text{DEC})$, where $ZN <= 1$ is replaced by $v_{<=}^1$ and $ZN - 1$ is replaced by v_{-}^1.

$$(\widehat{FB} \Leftrightarrow \widehat{v_{<=}^1} \wedge \overline{v_{<=}^1})$$
$$\wedge (\widehat{v_{<=}^1} \Leftrightarrow \widehat{ZN})$$
$$\wedge (\widehat{ZN} \Leftrightarrow \widehat{N})$$
$$\wedge (\widehat{N} \Leftrightarrow \widehat{FB} \vee \widehat{v_{-}^1})$$
$$\wedge (\widehat{v_{-}^1} \Leftrightarrow \widehat{ZN})$$

3.3 Concrete Clock Semantics

Let $X_{\mathbb{B}} \subseteq X$ be the set of all Boolean or *event* signals. We rely on the basic elements of *trace semantics* [21] to define the *clock semantics* of a synchronous program.

Definition 1. *(Clock events) 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 for X and $\overline{I}$ for $X_{\mathbb{B}}$. The interpretations I, $\overline{I}$ are respectively mappings from X^n to $\mathbb{B}^n$ and from $X_{\mathbb{B}}^m$ to $\mathbb{B}^m$, where $I(x) = $ `true` if x holds a value while $I(x) = $ `false` if it holds no value; and $\overline{I}(x) = $ `true` if x holds the value* `true`*, $\overline{I}(x) = $ `false`*, otherwise.*

For example, consider a program whose variables are $X = \{x, b\}$ where b is Boolean variable, the set of clock events is $\mathcal{E}c_X = \{(x \mapsto_I$ `false`$, b \mapsto_I$ `false`$, b \mapsto_{\overline{I}}$ `false`$), (x \mapsto_I$ `fa − lse`$, b \mapsto_I$ `true`$, b \mapsto_{\overline{I}}$ `false`$), (x \mapsto_I$ `false`$, b \mapsto_I$ `true`$, b \mapsto_{\overline{I}}$ `true`$), (x \mapsto_I$ `true`$, b \mapsto_I$ `false`$, b \mapsto_{\overline{I}}$ `false`$), (x \mapsto_I$ `true`$, b \mapsto_I$ `true`$, b \mapsto_{\overline{I}}$ `false`$), (x \mapsto_I$ `true`$, b \mapsto_I$ `true`$, b \mapsto_{\overline{I}}$ `true`$)\}$. Then at a given instant, the signals clock information is one of these clock events. By convention, the set of clock events of the empty set is defined as the empty set $\mathcal{E}c_\emptyset = \emptyset$.

Definition 2. *(Clock traces) Given a non-empty set 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 $T_c : \mathbb{N} \longrightarrow \mathcal{E}c_X$.*

The natural numbers represent the instants $t = 0, 1, 2, ...$ A trace Tc is a chain of clock events. We denote the interpreted value (`true` or `false`) of a variable x_i at instant t by

$T_c(t)(x_i)$, and $\overline{T_c(t)(x_i)}$ if $x_i \in X_{\mathbb{B}}$. Considering the above example, we have $T_c : (0, (x \mapsto_I$ `false`$, b \mapsto_I$ `false`$, b \mapsto_{\overline{I}}$ `false`$)), (1, (x \mapsto_I$ `false`$, b \mapsto_I$ `true`$, b \mapsto_{\overline{I}}$ `false`$)), ...$ as one of the possible clock traces on X, and $T_c(0)(x) = T_c(0)(b) = \overline{T_c(0)(b)} = $ `false`.

Definition 3. *(Clock trace restriction) 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 $\forall t \in \mathbb{N}, \forall x \in X_1, X_1.T_c(t)(x) = T_c(t)(x)$ and $\overline{X_1.T_c(t)(x)} = \overline{T_c(t)(x)}$ if $x \in X_{\mathbb{B}}$.*

We write $[[P]]_c$ to denote the clock semantics of program P which is defined as a set of possible clock traces.

Let $\hat{X} = \{\widehat{x_1}, ..., \widehat{x_n}, \overline{x_1}, ..., \overline{x_n}\} \cup \widehat{V} \cup \overline{V}$ be a finite set of variables that are used to construct the abstraction, where V is a set of newly added variables in *uninterpreted functions* replacement. Considering an interpretation $\hat{I}$ over $\hat{X}$, it is called a *clock configuration* iff it is a *model* of the first-order logic formula $\Phi(P)$. For example, $(\widehat{FB} \mapsto$ `false`$, \widehat{N} \mapsto$ `true`$, \widehat{ZN} \mapsto$ `true`$)$ is a clock configuration of $\Phi(\text{DEC})$, but $(\widehat{FB} \mapsto$ `false`$, \widehat{N} \mapsto$ `true`$, \widehat{ZN} \mapsto$ `false`$)$ is not one (we omit to write the interpretation for other variables).

Given a clock configuration $\hat{I}$, the set of clock events according to $\hat{I}$ and the set of all clock events of $\Phi(P)$ are computed as follows:

$$S_{sat}(\hat{I}) = \{I \in \mathcal{E}c_X | \forall i, I(x_i) = \hat{I}(\widehat{x_i}) \qquad (1)$$
$$\text{and } \overline{I}(x_i) = \hat{I}(\overline{x_i}) \text{ if } x_i \in X_{\mathbb{B}}\}$$
$$S_{sat}(\Phi(P)) = \bigcup_{\hat{I} \models \Phi(P)} S_{sat}(\hat{I}) \qquad (2)$$

With a set of clock events $S_{sat}(\Phi(P))$, the *concrete clock semantics* of $\Phi(P)$ is defined by the following set of clock traces:

$$\Gamma(\Phi(P)) = \{T_c \in \mathcal{T}c_X | \forall t, T_c(t) \in S_{sat}(\Phi(P))\} \qquad (3)$$

3.4 Soundness of the Abstraction

Table 2 and 3 show the clock semantics of the primitive operators with non-Boolean and Boolean signals, respectively. For instance, the clock semantics of the basic process corresponding to *Boolean sampling* is the following set of clock traces:

$T_c = \{(0, (c_{x_0}, c_{b_0}, b_0, c_{y_0})), ..., (i, (c_{x_i}, c_{b_i}, b_i, c_{y_i})), ...\}$ s.t
$\forall i, (c_{x_i}, c_{b_i}, b_i, c_{y_i}) \in \{(\text{false}, \text{false}, \text{false}, \text{false}),$
$(\text{true}, \text{false}, \text{false}, \text{false}), (\text{false}, \text{true}, \text{false}, \text{false}),$
$(\text{false}, \text{true}, \text{true}, \text{false}), (\text{true}, \text{true}, \text{false}, \text{false}),$
$(\text{true}, \text{true}, \text{true}, \text{true})\}$

Definition 4. *Given the abstraction $\Phi(P)$, a property φ defined over the set of clocks $\hat{X}$ is satisfied by $\Phi(P)$ if for any interpretation $\hat{I}$, $\hat{I} \models \Phi(P)$ whenever $\hat{I} \models \varphi$, denoted by $\Phi(P) \models \varphi$.*

To show the soundness of our abstraction, we consider a similar reasoning as in [20]. Our abstraction above is sound in terms of preservation of the clock semantics of the abstracted program P: if the clock semantics of the abstraction satisfies a property defined over the clocks, then the abstracted program also satisfies this property as stated by the following proposition. For any property φ which is defined over the set $\hat{X}$, its concretization $\Gamma(\varphi)$ is given by:

$$S_{sat}(\varphi) = \bigcup_{\hat{I} \models \varphi} S_{sat}(\hat{I}) \qquad (4)$$

$$\Gamma(\varphi) = \{T_c \in \mathcal{T}c_X | \ \forall t, T_c(t) \in S_{sat}(\varphi)\} \qquad (5)$$

Proposition 1. *Let P, $\Phi(P)$ be a program and its abstraction, respectively, and φ is a property defined over the clocks. If $\Phi(P) \models \varphi$ then $[[P]]_c \subseteq \Gamma(\varphi)$.*

Lemma 1. *For all programs P, $[[P]]_c \subseteq \Gamma(\Phi(P))$.*

Proof. (Proposition 1) The proof of Proposition 1 is done by using Lemma 1. Given a clock trace $T_c \in [[P]]_c$, applying Lemma 1, $T_c \in \Gamma(\Phi(P))$ means that $\forall t, T_c(t) \in S_{sat}(\Phi(P))$. Since $\Phi(P) \models \varphi$, then every interpretation $\hat{I}$ satisfying $\Phi(P)$ also satisfies φ. Thus, any clock event $I \in S_{sat}(\Phi(P))$ is also in $S_{sat}(\varphi)$, meaning that $\forall t, T_c(t) \in S_{sat}(\varphi)$. Therefore, we have $T_c \in \Gamma(\varphi)$. $\qquad\square$

Proof. (Lemma 1) We prove it by induction on the structure of program P, meaning that for every primitive operator of the language we show that its clock semantics is a subset of the corresponding concretization.

- Stepwise Extensions: $P : y := f(x_1, ..., x_n)$. First, consider y as numerical signal; following the encoding scheme, we have $\Phi(P) = (\hat{y} \Leftrightarrow \widehat{v_f^i} \Leftrightarrow \widehat{x_1} \Leftrightarrow ... \Leftrightarrow \widehat{x_n})$. For any interpretation $\hat{I}$ such that $\hat{I} \models \Phi(P)$, we have:
  - either $\forall i, \hat{y} = 0$ and $\widehat{x_i} = 0$;
  - or $\forall i, \hat{y} = 1$ and $\widehat{x_i} = 1$.

  $S_{sat}(\Phi(P))$ is the set of all interpretations of the form above. Let $T_c \in [[P]]_c$ be a clock trace and $t \in \mathbb{N}$ be any instant, then either $\forall i, T_c(t)(y) = T_c(x_i) = 0$ or $T_c(t)(y) = T_c(x_i) = 1$, thus $T_c \in \Gamma(\Phi(P))$. When y is a boolean signal, the proof is similar.
- Previous Value, Boolean Sampling, and Deterministic Merging operators: we prove in the same manner.

Process P	Clock Semantics $[[P]]_c$
$y := f(x_1, ..., x_n)$	$\{T_c \in \mathcal{T}c_{\{y,x_1,...,x_n\}} \mid \forall t \in \mathbb{N},$ $(\forall i, T_c(t)(x_i) = T_c(t)(y))\}$
$y := x\$1 \ init \ a$	$\{T_c \in \mathcal{T}c_{\{x,y\}} \mid \forall t \in \mathbb{N},$ $(T_c(t)(x) = T_c(t)(y))\}$
$y := x \ when \ b$	$\{T_c \in \mathcal{T}c_{\{x,y,b\}} \mid \forall t \in \mathbb{N},$ $(T_c(t)(x) = T_c(t)(b) = true$ $and \ \overline{T_c(t)(b)} = true$ $and \ T_c(t)(y) = true) \ or$ $(T_c(t)(x) = T_c(t)(b) = true$ $and \ \overline{T_c(t)(b)} = false$ $and \ T_c(t)(y) = false) \ or$ $(T_c(t)(x) = T_c(t)(y) = false) \ or$ $(T_c(t)(b) = T_c(t)(y) = false)\}$
$y := x \ default \ z$	$\{T_c \in \mathcal{T}c_{\{x,y,z\}} \mid \forall t \in \mathbb{N},$ $(T_c(t)(x) = T_c(t)(y) = true) \ or$ $(T_c(t)(x) = false \ and$ $T_c(t)(z) = T_c(t)(y))\}$
$P_1 \mid P_2$	$\{T_c \in \mathcal{T}c_{X_1 \cup X_2} \mid$ $X_1.T_c \in [[P_1]]_c \ and \ X_2.T_c \in [[P_2]]_c\}$ $where \ [[P_1]]_c \subseteq \mathcal{T}c_{X_1}, [[P_2]]_c \subseteq \mathcal{T}c_{X_2}$

Table 2   Clock Semantics of the Basic Processes

- Composition: $P = P1|P2$. Let $T_c \in [[P]]_c$ be a clock trace, since $X_1.T_c \in [[P_1]]_c, X_2.T_c \in [[P_2]]_c, [[P_1]] \subseteq \Gamma(\Phi(P_1))$ and $[[P_2]]_c \subseteq \Gamma(\Phi(P_2))$, we have $\forall t, T_c(t) \in S_{sat}(\Phi(P_1))$ and $T_c(t) \in S_{sat}(\Phi(P_2))$. That means $\forall t, T_c(t) \in S_{sat}(\Phi(P_1) \wedge \Phi(P_2))$, or $T_c \in \Gamma(\Phi(P))$.

$\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\square$

4   Synchronous Dependence Graphs

The SDG represents a synchronous program as a labelled directed graph in which each node is a signal or a clock and each edge from a node to another node represents the dependence between nodes. Each edge is labeled by a clock constraint.

Thus, a dependence between two signals is conditioned: it means that the dependence is effective whenever the condition holds. For instance, $y := x \ when \ b$ specifies that at any instant at which x is present, b is present and b holds the value `true`, then y cannot be set before x. We can use a Boolean condition $\hat{x} \wedge \hat{b} \wedge \bar{b}$ to encode the fact that x is present, b is present and b holds the value `true`, where $\hat{x}, \hat{b}, \bar{b}$ are Boolean variables. Thus, the value of y depends on the current value of x whenever the condition $\hat{x} \wedge \hat{b} \wedge \bar{b}$ is satisfied.

In presenting the construction of a SDG below, first, we

show that a usual *Data Dependence Graph* (DDG) is not sufficient to represent the dependences in a polychronous program. Then clock constraints are represented as first-order logic formulas as in Section 3.

4.1 Data Dependence Graphs

As in [4], a DDG is a directed graph which contains nodes that represent locations of *definitions* and *uses* of variables in *basic blocks*, and edges that represent data dependences between nodes. Considering the pseudo-code of a program called Sum, Figure 2 partially shows its DDG (the figure shows only the data dependences that are related to variable *i*). Data dependence edges are depicted by dotted lines which are added to the *Control Flow Graph* (CFG), and labelled by the name of the variable that creates the dependence. Node numbers in the CFG correspond to statement numbers in the program (we treat each statement as a basic block). Each node that represents a transfer of control (e.g., node 4) has two edges with labels T(`true`) and F(`false`), all others are unlabeled.


```
Program Sum
1. read(n);
2. i = 1;
3. sum = 0;
4. while (i <= n)
5.     sum = sum + i;
6.     i = i + 1;
   endwhile
7. write(sum);
end Sum
```
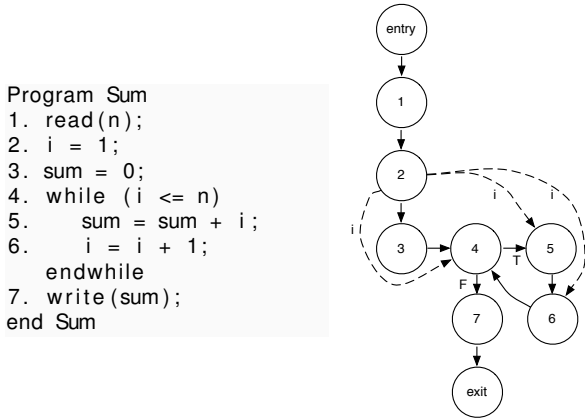
Fig. 2 CFG for Sum, with data dependence edges for i (dotted lines)

4.2 Signal Program as Synchronous Dependence Graph

Such data dependence graphs would not really represent the data dependences of a Signal program. Indeed, the dependences between signals in the program are not static. Since the presence of signals may vary along time (which is expressed by their clock), dependences also vary. To deal with that, the dependences are conditioned, and the conditions are represented by the clocks at which the dependences are effective.

To illustrate the definition of SDGs, we consider a process

which involves the basic process corresponding to the *deterministic merge* operator:

```
   (|
1. | x := expression
2. | z := expression
3. | ...
4. | y := x default z
   |)
```

Here, the numbers are added only for documenting, and the statement number 3 denotes a segment of program. The statements 1, 2 and 4 represent the expressions defining the signal x, z and y, respectively. Roughly speaking, the signal x is defined at statement 1 and is fetched at statement 4. Considering the basic process $y := x \; default \; z$ (and the clock constraints between signals), the "valid" states 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 Boolean abstraction. According to the valid states of the signals, the different data dependences between signals in the basic process $y := x \; default \; z$ are depicted in Figure 3, left, where the labels represent the conditions at which the dependences 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 translated in the Boolean abstraction of clock semantics as: $\hat{x} \Rightarrow \hat{y} \wedge \hat{x}$ and $\neg \hat{x} \wedge \hat{z} \Rightarrow \hat{y} \wedge \hat{z}$.

A SDG for a given program is a labelled directed graph in which each node is a signal or clock variable and each edge represents the dependence between nodes. Each edge is labelled by a first-order logic formula over Boolean variables which represents the clock at which the dependence between the extremity nodes is effective. Formally, a SDG is defined as follows:

Definition 5. *(SDG) A SDG associated with a process P is a tuple $G = \langle N, E, I, O, C, m_N, m_E \rangle$ 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 dependences 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 Boolean formulas over a set of clocks in the Boolean abstraction;*
- *$m_N : N \longrightarrow C$ is a mapping labelling 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.

In contrast with DDG, the clock labelling in SDG provides a dynamic dependence feature. This clock labelling imposes two properties which are implicit for a SDG:

- *An edge exists if its two extremity nodes exist.* This property can be translated in our Boolean abstraction as:

$$\forall (x, y) \in E, m_E(x, y) \Leftarrow (m_N(x) \wedge m_N(y))$$

- *A cycle of dependences stands for a deadlock.* It can be expressed as:

  A SDG G is deadlock-free iff
  $\forall x_1, ..., x_n, x_1 \in G$,
  $m_E(x_1, x_2) \wedge m_E(x_2, x_3) \wedge ... \wedge m_E(x_n, x_1)$ is `false`

We denote the fact that there exists a dependence between two nodes (signals or clocks) x and y at a clock constraint $m_E(x, y) = \hat{c}$ by $x \xrightarrow{\hat{c}} y$. A *dependence path* from x to y is any set of nodes $s = \{x_0, x_1, ..., x_k\}$ such that (an edge is a special case when $k = 1$):

$$x = x_0 \xrightarrow{\widehat{c_0}} x_1 \xrightarrow{\widehat{c_1}} ... \xrightarrow{\widehat{c_{k-1}}} x_k = y$$

In Table 4, we construct the dependences between signals for the core language, where the subclocks $[c]$ and $[\neg c]$ are encoded as $\hat{c} \wedge \bar{c}$ and $\hat{c} \wedge \neg \bar{c}$, respectively, in our abstraction. The edges are labelled by clocks which are represented by a Boolean formula in our abstraction. All the dependences in this table impose the implicit properties for a SDG, for instance, the basic process of the primitive operator *Boolean sampling* satisfies that $\hat{y} \Rightarrow \hat{x} \wedge \hat{y}$ and $\hat{y} \Rightarrow \hat{b} \wedge \hat{y}$. We also assume that all considered programs are written with the primitive operators, meaning that derived operators are replaced by their definition with primitive ones, and there are no nested operators (these nested operators can be broken by using fresh signals). Following the above construction rules, we can obtain the SDG in Figure 3, right, for the simple program DEC (we omit the part of graph that represents the dependences between ZN_2, ZN_1 and ZN).

Process P	Clock Semantics $[[P]]_c$
$y := f(x_1, ..., x_n)$	$\{T_c \in \mathcal{T}c_{\{y, x_1, ..., x_n\}} \mid \forall t \in \mathbb{N},$ $(\forall i, T_c(t)(x_i) = T_c(t)(y)) = false\ or$ $(\forall i, T_c(t)(x_i) = T_c(t)(y) = true\ and$ $\overline{T_c(t)(y)} = f(\overline{T_c(t)(x_1)}, ..., \overline{T_c(t)(x_n)}))\}$
$y := x\$1\ init\ a$	$\{T_c \in \mathcal{T}c_{\{x, y\}} \mid \forall t \in \mathbb{N},$ $(T_c(t)(x) = T_c(t)(y))\ or$ $(T_c(t)(x) = T_c(t)(y) = true\ and$ $\overline{T_c(t_0)(y)} = a\ and$ $\forall t \geq t_0, \overline{T_c(t)(y)} = \overline{T_c(t_-)(x)})$ with $t_0 = inf\{t' \mid T_c(t')(x) = true\}$, $t_- = sup\{t' \mid t' < t \wedge T_c(t')(x) = true\}\}$
$y := x\ when\ b$	$\{T_c \in \mathcal{T}c_{\{x, y, b\}} \mid \forall t \in \mathbb{N},$ $(T_c(t)(x) = T_c(t)(b) = true$ $and\ \overline{T_c(t)(b)} = true$ $and\ T_c(t)(y) = true\ and$ $\overline{T_c(t)(y)} = \overline{T_c(t)(x)})\ or$ $(T_c(t)(x) = T_c(t)(b) = true$ $and\ \overline{T_c(t)(b)} = false$ $and\ T_c(t)(y) = false)\ or$ $(T_c(t)(x) = T_c(t)(y) = false)\ or$ $(T_c(t)(b) = T_c(t)(y) = false)\}$
$y := x\ default\ z$	$\{T_c \in \mathcal{T}c_{\{x, y, z\}} \mid \forall t \in \mathbb{N},$ $(T_c(t)(x) = T_c(t)(y) = true\ and$ $\overline{T_c(t)(y)} = \overline{T_c(t)(x)})\ or$ $(T_c(t)(x) = false\ and\ T_c(t)(y) = T_c(t)(z)$ $and\ \overline{T_c(t)(y)} = \overline{T_c(t)(z)})\}$
$P_1 \mid P_2$	$\{T_c \in \mathcal{T}c_{X_1 \cup X_2} \mid$ $X_1.T_c \in [[P_1]]_c\ and\ X_2.T_c \in [[P_2]]_c\}$ where $[[P_1]]_c \subseteq \mathcal{T}c_{X_1}, [[P_2]]_c \subseteq \mathcal{T}c_{X_2}$

Table 3   Clock Semantics of the Basic Processes with Boolean Signals

5   Translation Validation for Synchronous Program Transformations

We adopt the translation validation approach [35, 36] to formally verify that the clock semantic and the dependence between variables in the program are preserved for every transformation of the compiler. To do that our verification framework uses clock models to represent the clock semantic of

x	$C_x \xrightarrow{\hat{x}} x$ $m_N(C_x) = \hat{x}, m_N(x) = \hat{x}$
c (Boolean signal)	$c \xrightarrow{\hat{c}} [c]$ $m_N(c) = \hat{c}, m_N([c]) = \hat{c}$ $c \xrightarrow{\hat{c}} [\neg c]$ $m_N(c) = \hat{c}, m_N([\neg c]) = \hat{c}$
$x \xrightarrow{c} y$	$[c] \xrightarrow{\hat{c}} y$ $m_N([c]) = \hat{c}, m_N(y) = \hat{y}$
$y := f(x_1, ..., x_n)$	$x_1 \xrightarrow{\hat{y}} y$ ... $x_n \xrightarrow{\hat{y}} y$ $m_N(x_i) = \widehat{x_i}, m_N(y) = \hat{y}, i = 1, ..., n$
$y := x\$1$ init a	$y \xrightarrow{\hat{y}} x$ $m_N(y) = \hat{y}, m_N(x) = \hat{x}$
$y := x$ when b	$x \xrightarrow{\hat{y}} y$ $m_N(x) = \hat{x}, m_N(y) = \hat{y}$ $b \xrightarrow{\hat{y}} C_y$ $m_N(b) = \hat{b}, m_N(C_y) = \hat{y}$
$y := x$ default z	$x \xrightarrow{\hat{x}} y$ $m_N(x) = \hat{x}, m_N(y) = \hat{y}$ $z \xrightarrow{\hat{z} \wedge \neg \hat{x}} y$ $m_N(z) = \hat{z}, m_N(y) = \hat{y}$

Table 4 The Dependences of the Core Language

original program and its transformed counterpart. We then introduce a *refinement* relation which expresses the preservations of clock semantic, as relation on clock models. Let $\Phi(P_1)$ and $\Phi(P_2)$ be two clock models, we write $\Phi(P_2) \sqsubseteq_{clk} \Phi(P_1)$ to denote $\Phi(P_2)$ is a refinement of $\Phi(P_1)$ w.r.t the clock semantics. This relation could be interpreted to mean that if a clock trace is in the set of clock traces of $\Phi(P_2)$, then it belongs to the set of clock traces of $\Phi(P_1)$ as well. For the preservation of dependence, SDG is used to represent the dependence in source program and its transformed counter-

Fig. 3 The SDG Example and SDG of DEC

part. A *refinement* relation which expresses the preservation of data dependence is defined as a relation on SDGs. Given $SDG(P_1)$ and $SDG(P_2)$, $SDG(P_2) \sqsubseteq_{dep} SDG(P_1)$ means that $SDG(P_2)$ is a refinement of $SDG(P_1)$.

5.1 Translation Validation for Clock Transformations

5.1.1 Definition of Correct Transformation: Clock Refinement

Let $\Phi(P_1)$ and $\Phi(P_2)$ be two clock models, to which we refer respectively as a source program and its transformed counterpart produced by the compiler. We assume that they have the same set of input and output variables. We will discuss in detail in the next section in case the compiler renames some local variables. We say that P_1 and P_2 have the same clock semantics if $\Phi(P_1)$ and $\Phi(P_2)$ have the same set of clock traces:

$$\forall T_c.(T_c \in \Gamma(\Phi(P_1)) \Leftrightarrow T_c \in \Gamma(\Phi(P_2))) \tag{6}$$

In general, the compilation makes the transformed program more concrete. For instance, when the Signal compiler does the *Boolean abstraction* which is used to generate the sequential executable code, the signal with the fastest clock is always present in the generated code. Additionally, compilers do transformations, optimizations for removing or eliminating some redundant behaviors of the source program (e.g. eliminating subexpressions, trivial clock relations). Therefore, Requirement (6) is too strong to be practical. To address this issue, the relaxed requirement is given by:

$$\forall T_c.(T_c \in \Gamma(\Phi(P_2)) \Rightarrow T_c \in \Gamma(\Phi(P_1))) \tag{7}$$

Requirement (7) expresses that if every clock trace of $\Phi(P_2)$ is also a clock trace of $\Phi(P_1)$, or $\Gamma(\Phi(P_2)) \subseteq \Gamma(\Phi(P_1))$. We say that $\Phi(P_2)$ is a *correct clock transformation* of $\Phi(P_1)$ or $\Phi(P_2)$ is a clock refinement of $\Phi(P_1)$.

Proposition 2. *The clock refinement is reflexive and transitive:*

- $\forall \Phi(P), \Phi(P) \sqsubseteq_{clk} \Phi(P)$.
- *If* $\Phi(P_1) \sqsubseteq_{clk} \Phi(P_2)$ *and* $\Phi(P_2) \sqsubseteq_{clk} \Phi(P_3)$, *then* $\Phi(P_1) \sqsubseteq_{clk} \Phi(P_3)$.

Proof. The reflexivity is obvious based on the clock refinement definition. For every clock trace $T_c \in \Gamma(\Phi(P_1))$, then $T_c \in \Gamma(\Phi(P_2))$. Since $\Phi(P_2) \sqsubseteq_{clk} \Phi(P_3)$, we have $T_c \in \Gamma(\Phi(P_3))$, or $\Phi(P_1) \sqsubseteq_{clk} \Phi(P_3)$. □
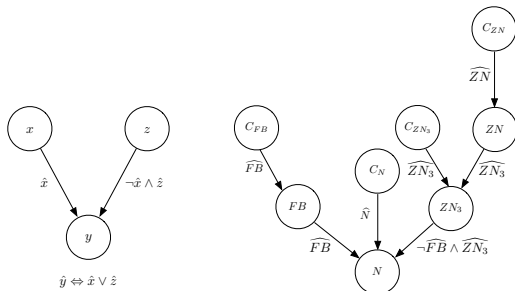
5.1.2 Proving Clock Refinement by SMT solver

We now discuss an approach to check the existence of refinement between two clock models (Requirement (7)) which is based on the following theorem.

Theorem 1. *Given a source program P_1 and its transformed program P_2, P_2 is a correct clock transformation of P_1 if it satisfies that for every interpretation $\hat{I}$, if $\hat{I}$ is a clock configuration of $\Phi(P_2)$, then it is also a clock configuration of $\Phi(P_1)$:*

$$(\models \Phi(P_2) \Rightarrow \Phi(P_1)) \Rightarrow \Phi(P_2) \sqsubseteq_{clk} \Phi(P_1) \qquad (8)$$

Proof. To prove Theorem 1, we show that if $\forall \hat{I}.(\hat{I} \models \Phi(P_2) \Rightarrow \hat{I} \models \Phi(P_1))$ then $\Gamma(\Phi(P_2)) \subseteq \Gamma(\Phi(P_1))$. Given $T_c \in \Gamma(\Phi(P_2))$, it means that $\forall t, T_c(t) \in S_{sat}(\Phi(P_2))$. Since $\forall \hat{I}.(\hat{I} \models \Phi(P_2) \Rightarrow \hat{I} \models \Phi(P_1))$, thus $S_{sat}(\Phi(P_2)) \subseteq S_{sat}(\Phi(P_1))$, meaning that $T_c(t) \in S_{sat}(\Phi(P_1))$ for every t. Therefore, we have $T_c \in \Gamma(\Phi(P_1))$. $\qquad\square$

To solve the validity of the formula $(\Phi(P_2) \Rightarrow \Phi(P_1))$ in (8), a SMT solver is needed since this formula involves non-Boolean variables and *uninterpreted functions*. 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 $\neg(\Phi(P_2) \Rightarrow \Phi(P_1))$ is unsatisfiable. Since $\neg(\Phi(P_2) \Rightarrow \Phi(P_1))$ is unsatisfiable iff $\models \Phi(P_2) \Rightarrow \Phi(P_1))$.

In our translation validation, the clock models which are constructed from Boolean, numerical variables and uninterpreted functions belong to a part of first-order logic which have a *small model* property according to [**?**]. The numerical variables are involved only in some implication with *uninterpreted functions* such as $(\overline{x} = \overline{x'} \wedge \overline{y} = \overline{y'}) \Rightarrow \overline{v_\square^i} = \overline{v_\square^j}$. In addition, the formula is quantifier-free. This means the check of satisfiability can be established by examining a certain finite cardinality of models, and it can be solved efficiently and significantly improves the scalability of the solver.

5.1.3 Illustrate on Example

Consider the program `DEC` and its transformed program of the *clock calculation* phase of the Signal compiler,

`DEC_BASIC_TRA`. For the validation process, the clock semantics of the transformed program is also represented as the clock model, $\Phi(\texttt{DEC\_BASIC\_TRA})$ as follows:

$$
\begin{aligned}
&(\widehat{CLK} \Leftrightarrow \widehat{CLK\_N} \wedge \neg \widehat{CLK\_FB}) \wedge (\widehat{CLK} \Rightarrow \overline{CLK}) \\
&\wedge (\widehat{CLK\_N} \Leftrightarrow \widehat{CLK\_N} \vee \widehat{CLK\_FB}) \\
&\quad \wedge (\widehat{CLK\_N} \Rightarrow \overline{CLK\_N}) \\
&\wedge \widehat{CLK\_N} \Leftrightarrow \widehat{N} \Leftrightarrow \widehat{ZN} \\
&\wedge (\widehat{N} \Leftrightarrow \widehat{FB} \wedge \widehat{CLK\_FB} \vee \widehat{v_-^1} \wedge \widehat{CLK}) \\
&\quad \wedge (\widehat{v_-^1} \Leftrightarrow \widehat{ZN}) \\
&\wedge (\widehat{ZN} \Leftrightarrow \widehat{N}) \\
&\wedge (\widehat{CLK\_FB} \Leftrightarrow \widehat{v_{<=}^1} \wedge \overline{v_{<=}^1}) \wedge (\widehat{v_{<=}^1} \Rightarrow \widehat{ZN}) \\
&\wedge \widehat{CLK\_FB} \Leftrightarrow \widehat{FB} \\
&\wedge \widehat{CLK\_12} \Leftrightarrow \widehat{v_{<=}^1} \wedge \neg \overline{v_{<=}^1}
\end{aligned}
$$

Then to check the transformation from `DEC` to `DEC_BASIC_TRA` is correct w.r.t the clock semantics, the validator will solve the validity of the formula $\Phi(\texttt{DEC\_BASIC\_TRA}) \Rightarrow \Phi(\texttt{DEC})$.

5.2 Translation Validation for SDGs

5.2.1 Definition of Correct Transformation: Dependence Refinement

Considering two SDGs $SDG(P_1)$ and $SDG(P_2)$, to which we refer respectively as a source program and its transformed counterpart produced by a compiler. A dependence path from x to y in $SDG(P_2)$ is *reinforcement* of the dependence path from x to y in $SDG(P_1)$ if at any instant the dependence path in $SDG(P_1)$ is effective implying that the dependence path in $SDG(P_2)$ is effective.

Definition 6. *(Reinforcement) Let $dp_1 = x \xrightarrow{\widehat{c_0}} x_1 \xrightarrow{\widehat{c_1}} ... \xrightarrow{\widehat{c_{n-1}}} y$ and $dp_2 = x \xrightarrow{\widehat{c_0'}} x_1' \xrightarrow{\widehat{c_1'}} ... \xrightarrow{\widehat{c_{m-1}'}} y$ be two dependence paths in $SDG(P_1)$ and $SDG(P_2)$, respectively. It is said that dp_2 is a reinforcement of dp_1 iff $(\bigwedge_{i=0}^{n-1} \widehat{c_i} \Rightarrow \bigwedge_{j=0}^{m-1} \widehat{c_j'})$.*

We write $dp_2 \preceq_{dep} dp_1$ to denote the fact that dp_2 is a reinforcement of dp_1. The condition $(\bigwedge_{i=0}^{n-1} \widehat{c_i} \Rightarrow \bigwedge_{j=0}^{m-1} \widehat{c_j'})$ is used to indicate that if the dependence path in $SDG(P_1)$ is effective then the dependence path in $SDG(P_2)$ is 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 $(c_0 \Rightarrow c_0')$.

Definition 7. *(Deadlock consistency) A dependence path $dp_2 = x \xrightarrow{\widehat{c_0'}} x_1' \xrightarrow{\widehat{c_1'}} ... \xrightarrow{\widehat{c_{m-1}'}} y$ in $SDG(P_2)$ is a deadlock-consistent for $dp_1 = x \xrightarrow{\widehat{c_0}} x_1 \xrightarrow{\widehat{c_1}} ... \xrightarrow{\widehat{c_{n-1}}} y$ in $SDG(P_1)$ iff for every dependence path $y \xrightarrow{\widehat{l_0}} z_1 \xrightarrow{\widehat{l_1}} ... \xrightarrow{\widehat{l_{p-1}}} x$ in $SDG(P_1)$*

such that $(\bigwedge_{i=0}^{n-1} \widehat{c_i} \wedge \bigwedge_{j=0}^{p-1} \widehat{l_j}) \Leftrightarrow$ false, *then for every dependence path* $y \xrightarrow{\widehat{l'_0}} z'_1 \xrightarrow{\widehat{l'_1}} ... \xrightarrow{\widehat{l'_{q-1}}} x$ *in* $SDG(P_2)$, *it satisfies* $(\bigwedge_{u=0}^{m-1} \widehat{c'_u} \wedge \bigwedge_{v=0}^{q-1} \widehat{l'_u}) \Leftrightarrow$ false, *denoted by* $dp_2 \succ_{dep} dp_1$.

When $m = n = p = q = 1$, $x \xrightarrow{\widehat{c_0}} y$ is deadlock-consistent for $x \xrightarrow{\widehat{c_0}} y$ iff $((c_0 \wedge l_0) \Leftrightarrow$ false$) \Rightarrow ((c'_0 \wedge l'_0) \Leftrightarrow$ false$)$. Deadlock consistency expresses the fact that if there are dependence paths from a signal x to a signal y and vise-versa such that there is no cyclic dependence between x and y in the source program, then the transformed program cannot introduce any cyclic dependence between x and y.

Recall that $SDG(P_1)$ and $SDG(P_2)$ are two SDGs, we assume that they have the same set of nodes. We say that the transformed counterpart P_2 of the source program P_1 preserves the dependences between signals if the following conditions are satisfied:

1. For any dependence path between signals from signal x to signal y in $SDG(P_1)$ at a clock constraint $\widehat{c_1}$, then there exists a dependence path from x to y at a clock constraint $\widehat{c_2}$ in $SDG(P_2)$ such that whenever the dependence in $SDG(P_1)$ is effective, then the dependence in $SDG(P_2)$ is also effective.

2. If there is no deadlocks in $SDG(P_1)$, then $SDG(P_2)$ introduces no deadlocks

We say that $SDG(P_2)$ is a *correct transformation* of $SDG(P_1)$ or $SDG(P_2)$ is a *dependence refinement* of $SDG(P_1)$. We write $SDG(P_2) \sqsubseteq_{dep} SDG(P_1)$ to denote the fact that there exists a dependence refinement relation between $SDG(P_2)$ and $SDG(P_1)$. The formal definition of dependence refinement is:

Definition 8. *(Dependence refinement) Let* $SDG(P_1)$ *and* $SDG(P_2)$ *be two synchronous dependence graphs,* $SDG(P_2)$ *is a dependence refinement of* $SDG(P_1)$ *if:*

- $\forall dp_1 = x \xrightarrow{\widehat{c_0}} x_1 \xrightarrow{\widehat{c_1}} ... \xrightarrow{\widehat{c_{n-1}}} y$ *in* $SDG(P_1)$,
  $\exists dp_2 = x \xrightarrow{\widehat{c'_0}} x'_1 \xrightarrow{\widehat{c'_1}} ... \xrightarrow{\widehat{c'_{m-1}}} y$ *in* $SDG(P_2)$
  s.t $dp_2 \preceq_{dep} dp_1$
- $\forall dp_1 = x \xrightarrow{\widehat{c_0}} x_1 \xrightarrow{\widehat{c_1}} ... \xrightarrow{\widehat{c_{n-1}}} y$ *in* $SDG(P_1)$ *and*
  $\forall dp_2 = x \xrightarrow{\widehat{c'_0}} x'_1 \xrightarrow{\widehat{c'_1}} ... \xrightarrow{\widehat{c'_{m-1}}} y$ *in* $SDG(P_2)$,
  $dp_2 \succ_{dep} dp_1$

Proposition 3. *The reinforcement, deadlock consistency, and dependence refinement are reflexive and transitive.*

Proof. **Reinforcement**

- *Reflexivity*: For any dependence path dp, based on the definition, we always have $dp \preceq_{dep} dp$.
- *Transitivity*: Assume that $dp_1 \preceq_{dep} dp_2$ and $dp_2 \preceq_{dep} dp_3$, we have $(\bigwedge_{i=0}^{n-1} \widehat{c_i} \Rightarrow \bigwedge_{j=0}^{m-1} \widehat{c'_j}) \wedge (\bigwedge_{j=0}^{m-1} \widehat{c'_j} \Rightarrow \bigwedge_{k=0}^{p-1} \widehat{c''_k})$, thus $(\bigwedge_{i=0}^{n-1} \widehat{c_i} \Rightarrow \bigwedge_{k=0}^{p-1} \widehat{c''_k})$, or $dp_1 \preceq_{dep} dp_3$.

Deadlock consistency

- *Reflexivity*: Based on the definition, we always have $dp \succ_{dep} dp$.
- *Transitivity*: Assume that $dp_1 \succ_{dep} dp_2$ and $dp_2 \succ_{dep} dp_3$, we have $(((\bigwedge_{i=0}^{n-1} \widehat{c_i} \wedge \bigwedge_{j=0}^{p-1} \widehat{l_j}) \Leftrightarrow$ false$) \Rightarrow ((\bigwedge_{u=0}^{m-1} \widehat{c'_u} \wedge \bigwedge_{v=0}^{q-1} \widehat{l'_u}) \Leftrightarrow$ false$) \wedge ((\bigwedge_{u=0}^{m-1} \widehat{c'_u} \wedge \bigwedge_{v=0}^{q-1} \widehat{l'_u}) \Leftrightarrow$ false$) \Rightarrow ((\bigwedge_{t=0}^{r-1} \widehat{c''_t} \wedge \bigwedge_{z=0}^{s-1} \widehat{l''_z}) \Leftrightarrow$ false$))$, thus $((\bigwedge_{i=0}^{n-1} \widehat{c_i} \wedge \bigwedge_{j=0}^{p-1} \widehat{l_j}) \Leftrightarrow$ false$) \Rightarrow ((\bigwedge_{t=0}^{r-1} \widehat{c''_t} \wedge \bigwedge_{z=0}^{s-1} \widehat{l''_z}) \Leftrightarrow$ false$))$, or $dp_1 \succ_{dep} dp_3$.

Dependence refinement

- *Reflexivity*: For every dependence path dp in $SDG(P)$, we have $dp \preceq_{dep} dp$ and $dp \succ_{dep} dp$, thus $SDG(P) \sqsubseteq_{dep} SDG(P)$.
- *Transitivity*: Assume that $SDG(P_1) \sqsubseteq_{dep} SDG(P_2)$ and $SDG(P_2) \sqsubseteq_{dep} SDG(P_3)$, we will show that $SDG(P_1) \sqsubseteq_{dep} SDG(P_3)$.
  i) For every dependence path dp_3 in $SDG(P_3)$, there exists a dependence path dp_2 in $SDG(P_2)$ such that $dp_2 \preceq_{dep} dp_3$. Since $SDG(P_1) \sqsubseteq_{dep} SDG(P_2)$, there exists a dependence path dp_1 in $SDG(P_1)$ such that $dp_1 \preceq_{dep} dp_2$. Following the transitivity of the reinforcement, we have $dp_1 \preceq_{dep} dp_3$.
  ii) For every dependence path dp_1 and dp_2 from node x to node y in $SDG(P_1)$ and $SDG(P_2)$, respectively, it satisfies $dp_1 \succ_{dep} dp_2$ since $SDG(P_1) \sqsubseteq_{dep} SDG(P_2)$. Because $SDG(P_2) \sqsubseteq_{dep} SDG(P_3)$, for every dependence path dp_3 from x to y in $SDG(P_3)$, we have $dp_2 \succ_{dep} dp_3$. Apply the transitivity property of the deadlock consistency, we have $dp_1 \succ_{dep} dp_3$.

$\square$

5.2.2 Proving Dependence Refinement by SMT solver

Given two SDGs, we introduce an approach to check the existence of dependence refinement between them that is implemented with a SMT-solver. 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, the formulas which label the edges of the graphs are over Boolean variables, thus the solving is decidable and very efficient [11].

Following Definition 8, we will traverse the entire graphs $SDG(P_1)$ and $SDG(P_2)$ to verify that:

- for every path in $SDG(P_1)$, there exists a reinforcement path in $SDG(P_2)$,
- and for any path from x to y in $SDG(P_1)$ and $SDG(P_2)$, they are deadlock-consistent.

It means that the basic element which is verified is that given two dependence paths, how we check the reinforcement and deadlock-consistent properties. Consider two dependence paths $dp_1 = x \xrightarrow{\widehat{c_0}} x_1 \xrightarrow{\widehat{c_1}} ... \xrightarrow{\widehat{c_{n-1}}} y$ and $dp_2 = x \xrightarrow{\widehat{c'_0}} x'_1 \xrightarrow{\widehat{c'_1}} ... \xrightarrow{\widehat{c'_{m-1}}} y$, dp_2 is a reinforcement of dp_1 iff $(\bigwedge_{i=0}^{n-1} \widehat{c_i} \Rightarrow \bigwedge_{j=0}^{m-1} \widehat{c'_j})$. The checking of this condition can be implemented by asking a SMT-solver to check $\models (\bigwedge_{i=0}^{n-1} \widehat{c_i} \Rightarrow \bigwedge_{j=0}^{m-1} \widehat{c'_j})$. In the same way, a SMT-solver can be used to check the deadlock consistency between two dependence paths, that means we will ask the SMT-solver to check the validity of the formula $((\bigwedge_{i=0}^{n-1} \widehat{c_i} \wedge \bigwedge_{j=0}^{p-1} \widehat{l_j}) \Leftrightarrow \texttt{false}) \Rightarrow ((\bigwedge_{u=0}^{m-1} \widehat{c'_u} \wedge \bigwedge_{v=0}^{p-1} \widehat{l'_u}) \Leftrightarrow \texttt{false})$.

We present here the concept of abstraction over SDGs which enable the checking process more efficient. According to the nature of SDGs, the abstraction is computed through the following rules of parallel and series [30] upon the input and output nodes:

$$x \xrightarrow{\widehat{c_0}} y \xrightarrow{\widehat{c_1}} z \qquad \Rightarrow x \xrightarrow{\widehat{c_0} \wedge \widehat{c_1}} z$$
$$x \xrightarrow{\widehat{c_0}} y \text{ and } x \xrightarrow{\widehat{c_1}} y \Rightarrow x \xrightarrow{\widehat{c_0} \vee \widehat{c_1}} y$$

Let $\overline{SDG(P_1)}$ and $\overline{SDG(P_2)}$ be graphs which are applied the rules of parallel and series, then $\overline{SDG(P_2)}$ is a dependence refinement of $\overline{SDG(P_1)}$ if the following conditions are satisfied:

- $\forall e_1 = (x, y)$ in $\overline{SDG(P_1)}, \exists e_2 = (x, y)$ in $\overline{SDG(P_2)}$ s.t $e_2 \preceq_{dep} e_1$
- $\forall e_1 = (x, y)$ in $\overline{SDG(P_1)}, e_2 = (x, y)$ in $\overline{SDG(P_2)}$ $e_2 \succeq_{dep} e_1$

6   Implementation

In this section, we describe the implementation of our validator and some adaptation when the translation validation is applied to the real Signal compiler. And we also show the previously unknown bugs have been detected so far by the validator.

6.1   Toward Certified Compiler

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

1. The compiler takes program P and transforms it.
2. If there is any error (i.e. syntax errors), it outputs an `Error`.
3. Otherwise, it outputs the intermediate representation $IR(P)$ (i.e. the intermediate representation is written in the same language syntaxes as the source program P).

These steps can be represented in the following pseudo-code, where $Cp(P)$ is the compilation step from the source program P to either compiled code $IR(P)$ or compilation errors.

```
1. if (Cp(P) is Error)
2.    then output Error;
3. else output IR(P);
```

Now, it is followed by our refinement verification which checks that the transformed program $IR(P)$ refines P w.r.t the clock semantic and the dependence. This will provide formal guarantee as strong as that provided by a formally certified compiler. Indeed, consider the following process:

```
1. if (Cp(P) is Error)
2.    then output Error;
3. else
4.    if ((Φ(IR(P)) ⊑clk Φ(P)) &&
          (SDG(IR(P)) ⊑dep SDG(P)))
5.       then output IR(P);
6.    else output Error;
```

We describe the main components of the implementation which is integrated in the existing Polychrony toolset [34] to prove the preservation of clock semantics and dependence of the Signal compiler. We are interested in the two first stages: *clock calculation, boolean abstraction* and *static scheduling*. The intermediate forms in the transformations of the compiler may be expressed in the Signal language itself.
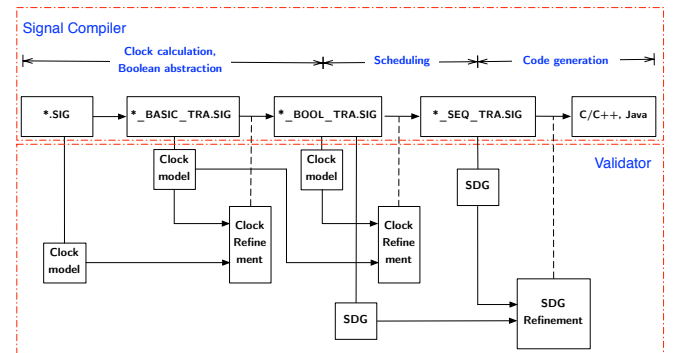

Fig. 4   An overview of our integration within Polychrony toolset

At a high level, our tool which is depicted in Figure 4 works as follows. First, it takes the input program `P.SIG` and its transformed program `P_TRA.SIG`, computes the corresponding clock models. The clock models of input and transformed programs are combined as the formula $(\Phi(P\_TRA.SIG) \Rightarrow \Phi(P.SIG))$. It uses a solver to check $\models (\Phi(\texttt{P\_TRA.SIG}) \Rightarrow \Phi(\texttt{P.SIG}))$ (or equivalently $M \not\models \neg(\Phi(\texttt{P\_TRA.SIG}) \Rightarrow \Phi(\texttt{P.SIG})))$. The result of this checking can be exploited for the preservation of clock semantic of the transformations. If the result says that the checked formula is not valid (or the negation formula is satisfiable) then it emits compilation error. Otherwise, the compiler continues its work. The same procedure is applied for the other steps of the compiler. Finally, our verification process asserts that $\Phi(\texttt{P\_BOOL\_TRA.SIG}) \sqsubseteq_{clk} \Phi(\texttt{P\_TRA.SIG}) \sqsubseteq_{clk} \Phi(\texttt{P.SIG})$ along the transformations of the compiler.

In the similar way, for the scheduling stage of the compilation, our tool takes the program `P_BOOL_TRA.SIG` and its transformed program `P_SEQ_TRA.SIG`, constructs the corresponding SDGs. Then it checks that SDG(`P_SEQ_TRA.SIG`) is a dependence refinement of SDG(`P_BOOL_TRA.SIG`). If the answer is "No", then it emits compilation error. Otherwise, the compiler continues its work.

Here, we delegate the checking of the refinements to a SMT solver. Our implementation uses the SMT-LIB common format [12] to encode the clock models as input of the SMT-solver. For our implementation, we consider the Yices [15] solver, which is one of the best solvers at the SMT-COMP competition [39].

6.2 Constant Clock and Renaming

In Signal, the occurrence of constants is allowed to designate a constant signal (e.g. a signal with a constant value). However, each occurrence of a constant has a particular clock since the corresponding signal is hidden, this clock is determined by the context where the constant is used, called *context clocks*. This makes our abstraction for Signal operator above invalid in case a constant signal is used. In consequence of that, we provide the abstraction for each Signal operator when this operator uses a constant signal, where `cst` denotes a constant.

6.2.1 Stepwise Extensions

- $\phi(y := cst) = \hat{y} \Rightarrow (\bar{y} \Leftrightarrow cst)$ if y is Boolean.
- $\phi(y := cst) = \emptyset$ if y is non-Boolean.
- $\phi(y := x \text{ and } cst) = (\hat{y} \Leftrightarrow \hat{x}) \wedge (\hat{y} \Rightarrow (\bar{y} \Leftrightarrow \bar{x} \wedge cst))$.

- $\phi(y := x \text{ or } cst) = (\hat{y} \Leftrightarrow \hat{x}) \wedge (\hat{y} \Rightarrow (\bar{y} \Leftrightarrow \bar{x} \vee cst))$.
- $\phi(y := x \ \Box \ cst) = (\hat{y} \Leftrightarrow v^i_\Box \Leftrightarrow \hat{x}) \wedge (\hat{y} \Rightarrow (\bar{y} = v^i_\Box))$.

6.2.2 Deterministic Merge

- x and y are Boolean.
  $\phi(y := x \text{ default } cst) = (\hat{y} \Leftrightarrow (\hat{x} \vee \hat{y})) \wedge \hat{y} \Rightarrow ((\hat{x} \wedge (\bar{y} \Leftrightarrow \bar{x})) \vee (\neg\hat{x} \wedge (\bar{y} \Leftrightarrow cst))))$
  $\phi(y := cst \text{ default } x) = (\hat{y} \Leftrightarrow (\hat{x} \vee \hat{y})) \wedge \hat{y} \Rightarrow ((\hat{y} \wedge (\bar{y} \Leftrightarrow cst)) \vee (\neg\hat{y} \wedge (\bar{y} \Leftrightarrow \bar{x}))))$
- x, y and z are non-Boolean.
  $\phi(y := x \text{ default } cst) = \hat{y} \Leftrightarrow (\hat{x} \vee \hat{y})$
  $\phi(y := cst \text{ default } x) = \hat{y} \Leftrightarrow (\hat{x} \vee \hat{y})$

6.2.3 Boolean Sampling

- x and y are Boolean.
  $\phi(y := x \text{ when true}) = (\hat{y} \Leftrightarrow (\hat{x} \wedge \hat{y})) \wedge (\hat{y} \Rightarrow (\bar{y} \Leftrightarrow \bar{x}))$
  $\phi(y := x \text{ when false}) = \hat{y} \Leftrightarrow \text{false}$
  $\phi(y := cst \text{ when } b) = (\hat{y} \Leftrightarrow (\hat{b} \wedge \bar{b})) \wedge (\hat{y} \Rightarrow (\bar{y} \Leftrightarrow cst))$
- x and y are non-Boolean.
  $\phi(y := x \text{ when true}) = \hat{y} \Leftrightarrow (\hat{x} \wedge \hat{y})$
  $\phi(y := x \text{ when false}) = \hat{y} \Leftrightarrow \text{false}$
  $\phi(y := cst \text{ when } b) = \hat{y} \Leftrightarrow (\hat{b} \wedge \bar{b})$

Consider a process P and its sub-process P_1 such that a signal named x is local variable of both P and P_1. When compiling this program, the compiler rename variable x in the sub-process P_1. Our validator requires that the mapping of the original name and the new name for every variable such as x. Based on this mapping, for every variable x and its new name x_i, the following conjunct is added to the clock model:

- $(\hat{x} \Leftrightarrow \widehat{x\_i}) \wedge (\bar{x} \Leftrightarrow \overline{x\_i})$ if x is Boolean.
- $(\hat{x} \Leftrightarrow \widehat{x\_i}) \wedge (\bar{x} = \overline{x\_i})$ if x is non-Boolean.

6.3 Detected Bugs

So far out validator has revealed 2 previously-unknown bugs in the compilation of the Signal compiler, one of them is related to the multiple constraints of clock. Another is a syntax error of generated C code from a Signal program in which a constant signal is presented.

The first problem was introduced when multiple constraints condition a clock such as the following segment of Signal program and its clock calculation parts in transformed programs.

```
// 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
...
```

In the transformed counterpart `P_BASIC_TRA`, the introduction of signal `XZX_24` and the synchronization between `CLK` and `XZX_24` cause the incorrect specification of clocks (e.g. in the source program `P` and `P_BOOL_TRA`, signal `x` is present, but in program `P_BASIC_TRA`, it might be absent when `XZX_24` is absent). This bug be caught by our validator when it found that $\Phi(\text{P\_BOOL\_TRA}) \not\sqsubseteq_{clk} \Phi(\text{P\_BASIC\_TRA})$. In addition, signal `XZX_24` is introduced without declaration that makes a syntax error in `P_BASIC_TRA`.

The second problem was present in the Signal program in which a *merge* operator with a constant signal is used such as `y := 1 default x`. The clock calculation is correct based on the validator result when it check the clock refinements between the input and transformed counterparts. However, it seems that the code generation phase of the compiler deals 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 by:

```
// 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);
}
```

7   Related Work and Conclusion

The notion of translation validation was invented in [35, 36] by A. Pnueli et al. to verify the code generator of Signal. In that work, the authors define a language of symbolic models to represent both the source and target programs, called *Synchronous Transition Systems (STS)*. A STS is a set of logic formulas which describes the functional and temporal con-

straints of the whole program and its generated C code. Then they use BDD [37] representations to implement the symbolic STS models, and their proof method uses a solver to reason on the constraints. The drawback of this approach is that it does not capture explicitly the clock semantic and data dependences. And in some cases, the compiler eliminates the use of a local register variable 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 cause 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 of the programs and does not explicitly prove the preservation of abstract clocks and data dependences.

Another related work is the static analysis of Signal programs for efficient code generation [20]. In a similar way as we do, the authors formalize the abstract clocks and clock relations as first-order logic formulas with the help of interval abstraction technique. The objective is to make the generated code more efficient by detecting and removing the dead-code segments (e.g., segment of code to compute a data-flow which is always absent). They determine the existence of empty clocks, mutual exclusion of two or more clocks, or clock inclusions, by reasoning on the formal model using a SMT solver.

Some other works have adopted the translation validation approach in verification of transformations, and optimizations. In [31], the translation validation is used to verify several common optimizations such as common subexpression elimination, register allocation, and loop inversion. The validator is simulation-based, that means it checks the existence of a simulation relation between two programs. Leroy et al. [29] used this technique to develop the CompCert high-assurance C compiler. The programs before and after the transformations and optimizations of the compiler are represented in a common intermediate form, then the preservation of semantics is checked by using symbolic execution and the proof assistant Coq. It also has shown that translation validation can be used to validate advanced loop optimizations such as software pipelining as in [42]. Tristan et al. [41] recently proposed a framework for translation validation of LLVM optimizer. For a function and its optimized counterpart, they compute a shared value-graph. The graph is *normalized* (roundly speaking, the graph is reduced). After the normalizing, if the outputs of two functions are represented by the same sub-graph, they can safely conclude that two functions are equivalent.

On the other hand, Pouzet et al. [7] introduce a generic machine-based intermediate presentation to describe the transition functions in the modular compilation of Scade/Lustre. The formalization of the intermediate presentation appears as a fundamental need in order to develop a certified compiler in a proof assistant. However, as we mentioned above, this approach yields a situation where any change of the compiler requires redoing the proof. Moreover, the compiler in general is much bigger and more difficult to verify than a validator such as ours. With the same purpose, in the work of [32], we encode the source Signal programs and their transformations with *Polynomial Dynamical Systems*, and we prove that the transformations preserve the abstract clocks and clock relations of the source programs. This approach uses simulation relation in model checking techniques, and it suffers from the increasing of the state-space when it deals with large programs. On the contrary, in our present work, the abstract clocks and clock relations are described as a logic formula over Boolean variables. Thanks to the efficiency of SMT solvers in processing formulas over Boolean variables, our approach can deal with large programs whose number of variables is very big. This situation generally makes the state-space explosion problem in model checking techniques.

The present paper provides a proof of preservations of clock semantic and the data dependence during the transformations of the Signal compiler. We have presented a technique based on SMT solving to prove these preservations of clock semantic and the data dependence. The desired behavior of a given source program and its transformed counterpart are represented as clock models and SDGs. Refinement relations between clock models and synchronous dataflow graphs are used to express the preservations, which are checked by using a SMT solver.

We have implemented and integrated our validator within the Polychrony toolset by using the Yices solver. As future work, we would like to extend our work to the final phase of the Signal compiler, the code generation. Meaning that we will adapt our translation validation to work with the code generation phase. We will use the proof of abstract clock semantics preservation to verify the equivalence between dataflows and the corresponding variables from the program and its generated code. The verification of equivalence will be done by using a value-graph which contains only the computations of data-flows and where there is no timing information. We will therefore evaluate this graph more efficiently.

References

1. W. Ackerman. Solvable cases of the Decision Problem. Study in Logic and the Foundations of Mathematics. North-Holland, Amsterdam, 1954.
2. W.B. Ackerman. Data flow languages. *Computer*. vol. 15, Feb 1982.
3. S. Abramsky and A. Jung: Domain theory: Handbook of logic in computer science. *Clarendon Press*. pp.1-168, 1994.
4. F.E. Allen. Control flow analysis. *In proceedings of a Symposium on Compiler Optimization, SIGPLAN Not. 5, 7*. pp1-19, Jul 1970.
5. A.V. Aho, R. Sethi, and J.D. Ullman. Compilers: Principles, Techniques, and Tools (Second edition). *Addison-Wesley Reading, MA*, 2007
6. A. Benveniste, P. Caspi, S.A. Edwards, N. Halbwachs, P. Le Guernic, and R. De Simone. The synchronous languages 12 years later. *Proceedings of the IEEE 91(1)*. pp.64-83, 2003.
7. D. Biernacki, J-L. Colaco, G. Hamon, and M. Pouzet. Clock-directed modular code generation of synchronous data-flow languages. *In ACM International Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES). Tucson, Arizona*, June 2008.
8. G. Berry: The foundations of Esterel. *In Proof, Language and Interaction: Essay in Honor of Robin Milner, MIT Press*, 2000.
9. L. Besnard, T. Gautier, P. Le Guernic, and J-P. Talpin. Compilation of polychronous data flow equations. *In Synthesis of Embedded Software, Springer*, 2010.
10. A. Benveniste and P. Le Guernic. Hybrid dynamical systems theory and the Signal language. *IEEE Transactions on Automatic Control, 35(5):535-546*, May 1990.
11. A. Biere, M. Heule, H. van Maaren, and T. Walsh. Handbook of satisfiability: Volume 185 Frontiers in Artificial Intelligence and Applications. *IOS Press, Amsterdam, The Netherlands*. ISBN 978-1-5860-3929-5, 2009.
12. C. Barrett, S. Ranise, A. Stump, and C. Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). *http://www.SMT-LIB.org*, 2008.
13. Inria. The CompCert Project. *http://compcert.inria.fr*.
14. Inria. The Coq Proof Assitant. *http://coq.inria.fr*.
15. B. Dutertre, and L. de Moura. Yices sat-solver. *http://yices.csl.ri.com*, 2009.
16. J. Ferrante, K.J. Ottenstein, and J.D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems, 9(3):319-349*, Jul 1987.
17. A. Gamatié. Designing embedded systems with the Signal programming: Synchronous, Reactive specification. *Springer, New York*. ISBN 978-1-4419-0940-4, 2009.
18. J.H. Gallier. Logic for computer science. *John Wiley*. 1987.
19. G.G. De Jong. Generalized data-flow graphs: Theory and applications. Ph.D thesis, Eindhoven University of Technology, 1993.

20. A. Gamatié, and L. Gonnord. 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*. Chicago, IL, USA, April 2011.

21. P. Le Guernic, and T. Gautier. Advanced topics in data-flow computing, chapter data-flow to von Neumann: the Signal approach. *Prentice-Hall*. pp.413-438, 1991.

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

23. P. Le Guernic, J-P. Talpin, and J-C. Le Lann. Polychrony for system design. *Journal for Circuits, Systems and Computers. 12(3):261-304*, Apr 2003.

24. N. Halbwachs. A synchronous language at work: the story of Lustre. *In 3th ACM-IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE'05)*, Jul 2005.

25. M. Huth, and M. Ryan. Logic in computer science: Modelling and Reasoning about systems. *Cambridge University Press*. ISBN 978-0-5215-4310-1, 2004.

26. G. Kahn: The Semantics of simple language for parallel programming. *IFIP Congress*. pp.471-475, 1974.

27. P. Le Guernic, T. Gautier, M. Le Borgne, and C. Le Maire. Programming real-time applications with Signal. *In Proceedings of the IEEE, 79(9):1321-1336*, Sep 1991.

28. L. de Moura, and N. Bjorner. Satisfiability Modulo Theories: An appetizer. *In Brazilian Symposium on Formal Methods (SBMF'2009), Gramado, Brazil*, Aug 2009.

29. X. Leroy. Formal certification of a compiler back-end, or programming a compiler with a proof assistant. *In 33rd Symposium Principles of Programming Languages, pp.42-54. ACM Press*, 2006.

30. O. Maffeis, and P. LeGuernic. Combining ependability with architectural adaptability by means of the signal language. *Static Analysis, LNCS 724*, 1993.

31. G.C. Necula. Translation Validation for an Optimizing Compiler. *In Proceeding PLDI'00 Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*. pp.83-94, May 2000.

32. V.C. Ngo, J-P. Talpin, T. Gautier, P. Le Guernic, and L. Besnard. Formal Verification of Compiler Transformations on Polychronous Equations. *In Proceedings of IFM'12, LNCS 7321*. pp.113-127, 2012.

33. V.C. Ngo, J-P. Talpin, T. Gautier, and P. Le Guernic. Formal Verification of Transformations on Abstract Clocks in Synchronous Compilers. *In HAL-INRIA, Technical Report RR-8064*, Sep 2012.

34. Espresso, Polychrony Toolset. *http://www.irisa.fr/espresso/Polychrony*.

35. A. Pnueli, M. Siegel, and E. Singerman. Translation validation. *In B. Steffen, editor, 4th Intl. Conf. TACAS'98. LNCS 1384*. pp.151-166, 1998.

36. A. Pnueli, O. Shtrichman, and M. Siegel. Translation validation: From Signal to C. *In Correct Sytem Design Recent Insights and Advances. LNCS 1710*. pp.231-255, 2000.

37. R. Bryant: Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677-691, Aug 1986.

38. DO-178C. *http://rtca.org*

39. A. Stump, and M. Deters. *http://www.smtcomp.org/2009*, 2009.

40. S.P. Rajan. Transformations on Dependency Graphs: Formal Specification and Efficient Mechanical Verification. Ph.D thesis, University of British Columbia, Vancouver, Canada.

41. J-B. Tristan, P. Govereau, and G. Morrisett. Evaluating value-graph translation validation for LLVM. *In ACM SIGPLAN Conference on Programming and Language Design Implementation*. California, 2011.

42. J-B. Tristan, and X. Leroy. A simple, verified validator for software pipelining. *In 37th Principles of Programming Languages, pp.83-92. ACM Press*, 2010.

Van Chan Ngo is PhD student at INRIA. His main research interests are design, development, and formal verification of embedded real-time systems, and compiler theory. He finished the engineer degree with excellent ranking in Computer Engineering at the Talent Training Center, Hanoi University of Technology in 2005, and the graduate degree in Applied Mathematics and Computer Science within the French government scholarship program at Verimag Laboratory, Université de Grenoble 1, Grenoble, France in 2008.

Jean-Pierre Talpin is Senior Researcher with INRIA and leads the project team who develops the open-source Polychrony environment. He received his PhD from Université Paris VI Pierre et Marie Curie in 1993. He then was a research associate with the European Computer-Industry Research Centre in Munich before to join INRIA in 1995. Jean-Pierre edited two books with Elsevier and Springer, guest-edited more than ten special issues of ACM and IEEE scientific journals, and authored more than 20 journal articles and book chapters and 60 conference papers. He received the 2004 ACM Award for the most influential POPL paper, for his 2nd conference paper with Mads Tofte, and the 2012 LICS Test of Time Award, for his 1st conference paper with Pierre Jouvelot.

Thierry Gautier is researcher with IN-RIA. He received the graduate degree from the Institut National des Sciences Appliquées, Rennes, France, in 1980, and the PhD degree in computer science from Université de Rennes 1 in 1984. He is one of the designers of the Signal language, the polychronous model of computation and the Polychrony toolset. His main research interests lie in the safe design of complex embedded systems, including formal modeling, formal validation, and transformations of models to target architectures.

Paul Le Guernic graduated from Institut National des Sciences Appliquées de Rennes in 1974. He performed his Thèse de troisième cycle in Computer Science in 1976. From 1978 to 1984 he had a research position at INRIA. He is Directeur de Recherche in this institute since 1985. He has been head of the "Programming Environment for Real-time Applications" group, which has defined and developed the Signal language. His main current interests include the development of theories, tools, and methods, for the design of real-time embedded heterogeneous systems. He is one the architects of the Polychrony toool set.

Loïc Besnard is currently a senior engineer at CNRS, France. He received his Ph.D. degree in computer science from Université de Rennes, France (1992). His research interests include the software reliability for the design of embedded systems: modeling, temporal analysis, formal verification, simulation, and synthesis of embedded systems. He is involved in the development of the Polychony toolset based on the synchronous language Signal.