Translation Validation for Clock Transformations in a Synchronous Compiler

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

INRIA Rennes - Bretagne Atlantique, 35042 Rennes Cedex, France
{firstname.lastname}@inria.fr

Abstract. Translation validation was introduced as a technique to formally verify the correctness of code generators that attempts to ensure that program transformations preserve the semantics of input program. In this work, we adopt this approach to construct a validator that formally verifies the preservation of clock semantics during the SIGNAL compiler transformations. The clock semantics is represented as a first-order logic formula called *clock model*. We then introduce a *refinement* which expresses the preservation of clock semantics, as a relation on clock models. Our validator does not require any instrumentation or modification of the compiler, nor any rewriting of the source program.

Keywords: Formal Verification, Translation Validation, Certified Compiler, SMT Solver, Synchronous Data-Flow Languages.

1 Introduction

Motivation. Synchronous programming languages such as SIGNAL, LUSTRE and ESTEREL propose a formal semantic framework to give a high-level specification of safety-critical software in automotive and avionics systems [10,13,2]. As other programming languages, synchronous languages are associated with a compiler. The compiler takes a source program, analyses and transforms it, performs optimizations, and finally generates executable code for a particular hardware platform or in some general-purpose programming languages. However, a compiler is a large and very complex program which often consists of hundreds of thousands, if not millions, lines of code, divided into multiple sub-systems and modules. The compilation process involves many analyzes, program transformations and optimizations. Some transformations and optimizations may introduce additional information, or constrain the compiled program. They may refine its meaning and specialize its behavior to meet a specific safety or optimization goal. Consequently, it is not uncommon that compilers silently issue an incorrect result in some unexpected context or inappropriate optimization goal. To circumvent compiler bugs, one can entirely rewrite the compiler with a theorem proving tool such as COQ [8], or check that it is compliant to the DO-178C documents [22]. Nonetheless, 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. [20,21], 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.

Contribution. We consider the SIGNAL compiler, in the first two phases, The *clock information* and *Boolean abstraction* are computed. The next phase is *static scheduling* and the final phase is the *executable code generation*. Obviously, one can prove that the input program and its transformed program at the final phase 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 the case of a synchronous compiler such as SIGNAL, the preservation of the semantics can be decomposed into the preservation of *clock semantics*, *data dependencies*, and *value-equivalence* of variables [18].

This paper focuses on constructing a validator that proves the preservation of clock semantics in the first two phases of the SIGNAL compiler. The clock semantics of the source program and its transformed counterpart are formally represented as *clock models*. A clock model is a first-order logic formula with *uninterpreted functions*. This formula deterministically characterizes the presence/absence status of all discrete data-flows (input, output and local variables of a program) manipulated by the specification at a given logic instant. Given two clock models, a *correct transformation* relation between them is checked by the existence of their *refinement* relation, which expresses the semantic preservation of clock semantics. In the implementation, we apply our translation validation to the first two transformation steps of the compiler.

The remainder of this paper is organized as follows. Section 2 introduces the SIGNAL language. Section 3 presents the abstraction that represents the clock semantics in terms of first-order logic formula. In Section 4, we consider the definition of correct transformation on clock models which formally proves the conformance between the original specification and its transformed counterpart. The application of the verification process to the SIGNAL compiler, and its integration in the Polychrony toolset [19] is addressed in Section 5. Section 6 presents related works and concludes our work.

2 The SIGNAL Language

SIGNAL [5,11] is a polychronous data-flow language that allows the specification of multi-clocked systems, called *polychrony models*. SIGNAL handles unbounded sequences of typed values $x(t)_{t \in \mathbb{N}}$, called *signals*, denoted as x. Each signal is implicitly indexed by a logical *clock* indicating the set of instants at which the signal is present, noted C_x. At a given instant, a signal may be present where it holds a value, or absent where it holds no value, denoted by $\perp$. Given two

signals, they are *synchronous* iff they have the same clock. A process, written `P` or `Q`, consists of the synchronous composition of equations over signals x, y, z, written $x := y \; \mathtt{op} \; z$ or $x := \mathtt{op}(y, z)$, where `op` is an operator. 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 other processes as external parameters. A program is a process and the language is modular.
Data Domains. Data types consist of usual scalar types (Boolean, integer, float, complex, and character), enumerated types, array types, tuple types, and the special type `event`. It is a subtype of the Boolean which has only one value, `true`.
Operators. The core language consists of two kinds of "statement" defined by the following primitive operators: four operators on signals and two operators on processes. The operators on signals define basic processes with implicit clock relations while the operators on processes are used to construct complex processes with the parallel composition operator. In the *delay* operator, inf and sup denote the greatest lower bound and the least upper bound.

- *Stepwise Functions:* $y := f(x_1, ..., x_n)$, where f is a *n-ary* function on values, defines a basic process whose output y is synchronous with $x_1, ..., x_n$ ($C_y = C_{x_1} = ... = C_{x_n}$) and $\forall t \in C_y, y(t) = f(x_1(t), ..., x_n(t))$.
- *Delay:* $y := x\$1 \; \mathtt{init} \; a$ defines a basic process such that y and x are synchronous ($C_y = C_x$), $y(t_0) = a$, and $\forall t \in C_y \wedge t > t_0, y(t) = x(t^-)$ with $t_0 = \inf\{t' | x(t') \neq \bot\}, t^- = \sup\{t' | t' < t \wedge x(t') \neq \bot\}$.
- *Merge:* $y := x \; \mathtt{default} \; z$ defines a basic process which specifies that y is present iff x or z is present ($C_y = C_x \cup C_z$), and that $y(t) = x(t)$ if $t \in C_x$ and $y(t) = z(t)$ if $t \in C_z \setminus C_x$.
- *Sampling:* $y := x \; \mathtt{when} \; b$ where b is a Boolean signal, defines a basic process such that $\forall t \in C_x \cap C_b \wedge b(t) = \mathit{true}, y(t) = x(t)$, and otherwise, y is absent ($C_y = C_x \cap [b]$, where $[b] = \{t \in C_b | b(t) = \mathtt{true}\}$).
- *Composition:* If $\mathtt{P}_1$ and $\mathtt{P}_2$ are processes, then $\mathtt{P}_1 | \mathtt{P}_2$, also denoted as $(|\mathtt{P}_1 | \mathtt{P}_2|)$, is the process resulting of their parallel composition. This process consists of the composition of the systems of equations. The composition operator is commutative, associative, and idempotent.
- *Restriction:* $\mathtt{P} \; \mathtt{where} \; x$, where `P` is a process and x is a signal, specifies a process by considering x as local variable to `P` (i.e., x is not accessible from outside `P`).

Clock Relations. Clock relations can be defined explicitly: $y := \hat{}x$ specifies that y with `event` type is the clock of x, C_x. The synchronization $x \hat{=} y$ means that x and y have the same clock. The clock extraction from a Boolean signal is denoted by a unary *when*: $\mathtt{when} \; b$. The clock union $x \hat{+} y$ defines a clock as the union $C_x \cup C_y$. 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 $C_x \setminus C_y$.
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. We can see that the clock of the output signal is more frequent than that of the

input. The following diagram illustrates one possible execution of the program DEC.

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

t	.	.	.	.	.	.	.	.	.	.
FB	6	⊥	⊥	⊥	⊥	⊥	3	⊥	⊥	2
ZN	1	6	5	4	3	2	1	3	2	1
N	6	5	4	3	2	1	3	2	1	2
C_{FB}	t_0						t_6			t_9
C_{ZN}	t_0	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9
C_N	t_0	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9

3 Clock Model

In SIGNAL, clocks play a much more important role than in other synchronous languages, they are used to express the underlying control (i.e., the synchronization between signals) for any conditional definition. This differs from LUSTRE, where all clocks are built by sampling the fastest clock. We consider the following equation with the primitive operator *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. We use a Boolean variable $\hat{x}$ to capture the status of x: ($\hat{x} = \texttt{true}$) means x is present, and ($\hat{x} = \texttt{false}$) means x is absent. In the same way, the Boolean variable $\hat{y}$ captures the status of y. For b, two Boolean variables $\hat{b}$ and $\widetilde{b}$ are used to represent its status: ($\hat{b} = \texttt{true} \wedge \widetilde{b} = \texttt{true}$) means b is present and holds a value true; ($\hat{b} = \texttt{true} \wedge \widetilde{b} = \texttt{false}$) means b is present and holds a value false; and ($\hat{b} = \texttt{false}$) means b is absent. Hence, at a given instant, the clock relations of the equation above can be encoded by the formula: $\hat{y} \Leftrightarrow (\hat{x} \wedge \hat{b} \wedge \widetilde{b})$

3.1 Abstraction

Let $X = \{x_1, ..., x_n\}$ be the set of all signals in program P consisting of input, output, register (corresponding to *delay* operator), and local signals, denoted by I, O, R and L, respectively. With each signal x_i, based on the encoding scheme proposed in [12], we attach a Boolean variable $\widehat{x_i}$ to encode its clock and a variable $\widetilde{x_i}$ of same type as x_i to encode its value. The composition of processes corresponds to logical conjunctions. Thus the clock model of $\mathtt{P}$ will be a conjunction $\Phi(\mathtt{P}) = \bigwedge_{i=1}^{n} \phi(eq_i)$, whose atoms are $\widehat{x_i}, \widetilde{x_i}$, where $\phi(eq_i)$ is the abstraction of statement eq_i, and n is the number of statements in the program. In the following, we present the abstraction corresponding to each Signal operator.

Stepwise Functions. The functions which apply on signal values in the stepwise functions are usual logic operators (`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 definition of a signal is determined explicitly by the input and register signals, 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 these operations by *uninterpreted functions*. By following the encoding procedure of [1], for every numerical comparison function and numerical operator (denoted by $\Box$) occurring in an equation, we perform the following rewriting: i) Replace each $x \Box y$ by a new variable $v^i_\Box$ of the same type as the return value by $\Box$. Two stepwise functions $x \Box y$ and $x' \Box y'$ are replaced by the same variable $v^i_\Box$ iff x, y are identical to x' and y', respectively; ii) For every pair of newly added variables $v^i_\Box$ and $v^j_\Box$, $i \neq j$, corresponding to the non-identical occurrences $x \Box y$ and $x' \Box y'$, add the implication $(\widetilde{x} = \widetilde{x'} \wedge \widetilde{y} = \widetilde{y'}) \Rightarrow \widetilde{v^i_\Box} = \widetilde{v^j_\Box}$ into the abstraction $\Phi(\mathtt{P})$. The abstraction $\phi(y := f(x_1, ..., x_n))$ of stepwise functions is defined by induction as follows: $\phi(\mathtt{true}) = \mathtt{true}$ and $\phi(\mathtt{false}) = \mathtt{false}$; $\phi(y := x) = (\hat{y} \Leftrightarrow \hat{x}) \wedge (\hat{y} \Rightarrow (\widetilde{y} = \widetilde{x}))$; $\phi(y := x) = (\hat{y} \Leftrightarrow \hat{x}) \wedge (\hat{y} \Rightarrow (\widetilde{y} = \widetilde{x})) \wedge (\hat{x} \Rightarrow \widetilde{x})$ if x is an `event` signal; $\phi(y := \mathtt{not}\ x) = (\hat{y} \Leftrightarrow \hat{x}) \wedge (\hat{y} \Rightarrow (\widetilde{y} \Leftrightarrow \neg\widetilde{x}))$; $\phi(y := x_1\ \mathtt{and}\ x_2) = (\hat{y} \Leftrightarrow \widehat{x_1} \Leftrightarrow \widehat{x_2}) \wedge (\hat{y} \Rightarrow (\widetilde{y} \Leftrightarrow \widetilde{x_1} \wedge \widetilde{x_2}))$; $\phi(y := x_1\ \mathtt{or}\ x_2) = (\hat{y} \Leftrightarrow \widehat{x_1} \Leftrightarrow \widehat{x_2}) \wedge (\hat{y} \Rightarrow (\widetilde{y} \Leftrightarrow \widetilde{x_1} \vee \widetilde{x_2}))$; $\phi(y := x_1 \Box x_2) = (\hat{y} \Leftrightarrow \widehat{v^i_\Box} \Leftrightarrow \widehat{x_1} \Leftrightarrow \widehat{x_2}) \wedge (\hat{y} \Rightarrow (\widetilde{y} = \widetilde{v^i_\Box}))$.

Delay. Considering the *delay* operator, $y := x\$1\ \mathtt{init}\ a$, its encoding $\phi(y := x\$1\ \mathtt{init}\ a)$ contributes to $\Phi(\mathtt{P})$ with the following conjunct: $(\hat{y} \Leftrightarrow \hat{x}) \wedge (\hat{y} \Rightarrow ((\widetilde{y} = m.x) \wedge (m.x' = \widetilde{x}))) \wedge (m.x_0 = a)$. This encoding requires that at any instant, signals x and y have the same status (present or absent). To encode the value of the output signal as well, we introduce a memorization variable $m.x$ that stores the last value of x. The next value of $m.x$ is $m.x'$ and it is initialized to a in $m.x_0$.

Merge. The encoding of the *merge* operator, $y := x\ \mathtt{default}\ z$, contributes to $\Phi(\mathtt{P})$ with the following conjunct: $(\hat{y} \Leftrightarrow (\hat{x} \vee \hat{z})) \wedge \hat{y} \Rightarrow ((\hat{x} \wedge (\widetilde{y} = \widetilde{x})) \vee (\neg\hat{x} \wedge (\widetilde{y} = \widetilde{z})))$

Sampling. The encoding of the *sampling* operator, $y := x\ \mathtt{when}\ b$, contributes to $\Phi(\mathtt{P})$ with the following conjunct: $(\hat{y} \Leftrightarrow (\hat{x} \wedge \hat{b} \wedge \overline{b})) \wedge (\hat{y} \Rightarrow (\widetilde{y} = \widetilde{x}))$

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

Clock Relations. Given the above rules, we can obtain the following abstraction for derived operators on clocks. Here, z is a signal of type `event`: $\phi(z := \hat{}x) = (\hat{z} \Leftrightarrow \hat{x}) \wedge (\hat{z} \Rightarrow \bar{z})$; $\phi(x\ \hat{}{=}\ y) = \hat{x} \Leftrightarrow \hat{y}$; $\phi(z := x\ \hat{}{+}\ y) = (\hat{z} \Leftrightarrow (\hat{x} \vee \hat{y})) \wedge (\hat{z} \Rightarrow \bar{z})$;

$\phi(z := x \,\hat{}{*}\, y) = (\hat{z} \Leftrightarrow (\hat{x} \wedge \hat{y})) \wedge (\hat{z} \Rightarrow \bar{z})$; $\phi(z := x \,\hat{}{-}\, y) = (\hat{z} \Leftrightarrow (\hat{x} \wedge \neg\hat{y})) \wedge (\hat{z} \Rightarrow \bar{z})$; $\phi(z := \texttt{when}\ b) = (\hat{z} \Leftrightarrow (\hat{b} \wedge \bar{b})) \wedge (\hat{z} \Rightarrow \bar{z})$.

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

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

In the following sections, we denote input, output, register, memorization and local variables used in a clock model by $I_{clk}, O_{clk}, R_{clk}, M_{clk}$ and L_{clk}, respectively. Note that the memorization variables are introduced only by the translation into clock models, they are not original in the Signal programs.

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

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

3.2 Concrete Clock Semantics

We rely on the basic elements of *trace semantics* [14] to define the clock semantics of a synchronous program. For each $x_i \in X$, we use $\mathbb{D}_{x_i}$ to denote its domain of values, and $\mathbb{D}^{\perp}_{x_i} = \mathbb{D}_{x_i} \cup \{\perp\}$ to denote its domain of values with absent value, where $\perp \notin \mathbb{D}_{x_i}$ denotes the absent value. Then, the domain of values of X with absent value is defined as $\mathbb{D}^{\perp}_{X} = \bigcup_{i=1}^{n} \mathbb{D}_{x_i} \cup \{\perp\}$

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

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

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

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

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

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

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

4 Clock Model Translation Validation

We adopt the translation validation approach [20,21] to formally verify that the clock semantics is preserved for every transformation of the compiler. In order to apply the translation validation to the transformations, we capture the clock semantics of the original program and its transformed counterpart by means of clock models. Then we introduce a *refinement* relation which expresses the preservation of clock semantics, as relation on clock models.

4.1 Clock Refinement

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

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

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

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

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

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

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

4.2 Adaptation to SIGNAL Compiler

We will adapt the definition of the above general clock refinement to the case of the Signal compiler. We need to consider the following factors [4]. A first consideration is that the programs take the inputs from their environment and the register values. Then, they calculate the outputs to react with the environment. In general, the programs can use some local variables to make the output calculations. However, from the outside, the natural observation of the programs is the snapshot of the values of the input and output signals. In our context, it is the snapshot of the presence of the input and output signals. For example, for the program DEC, the observation is the tuple of the presence of the signals (FB, N) at a considered instant.

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

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

4.3 Proving Clock Refinement by SMT

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

- The identification of a *variable mapping* that maps the non input/output variables from the clock model $\Phi(\mathtt{A})$ to the non input/output variables in the clock model $\Phi(\mathtt{C})$. We denote the mapping by: $\widehat{X_A} \setminus \widehat{X_{IO}} = \alpha(\widehat{X_C} \setminus \widehat{X_{IO}})$
- The premises of a rule such that if the premises hold, then the conclusion, $\Phi(\mathtt{C})$ refines $\Phi(\mathtt{A})$, is `true`. The premise is presented in Fig. 1.

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

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

Fig. 1: Rule CLKREF

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

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

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

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

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

It remains to define the instance of the mapping α for variables $\hat{l}, \widetilde{l} \in R_{clk} \cup L_{clk}$ in the clock model $\Phi(\mathtt{A})$ which correspond to the local or register signal named l in the program. In a Signal program, one signal is defined by an equation $l = eq$, if we follow the definitions of all output and local signals in this equation and apply successively substitutions, then we get that the equation is constructed only by the input and register signals. This property is yielded since the Signal program is determinate, meaning that all definitions of signals are defined determinately by the input and register signals, and the compilers rejects all non-determinate program. Equivalently, in the corresponding clock model $\Phi(\mathtt{A})$, the output, register and local variables are determinately defined by the input I and memorization M variables. The definition is written in the clock model in the form $\hat{l} \Leftrightarrow \hat{f} \wedge (\hat{l} \Rightarrow \widetilde{l} = \widetilde{f})$ or $\hat{l} \Leftrightarrow \hat{f} \wedge (\hat{l} \Rightarrow \widetilde{l} = \widetilde{f}) \wedge \widetilde{f}_0$, where $\hat{f}$, $\widetilde{f}$ and $\widetilde{f}_0$ are the formulas which define the clock relation, the value, and the initial value of the signal l in the clock model $\Phi(\mathtt{A})$. Therefore, we define the following instance of the α mapping in the clock model corresponding to each register or local signal l: $\hat{l} \Leftrightarrow \hat{f} \wedge (\hat{l} \Rightarrow \widetilde{l} = \widetilde{f})$ or $\hat{l} \Leftrightarrow \hat{f} \wedge (\hat{l} \Rightarrow \widetilde{l} = \widetilde{f}) \wedge \widetilde{f}_0$.

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

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

$$\textstyle\bigwedge_{m.s \in M} (\widetilde{s} = \widetilde{s^c} \Rightarrow m.s = m.s^c) \wedge \bigwedge_{\hat{l}, \widetilde{l} \in S \cup L} (\hat{l} \Leftrightarrow \hat{f} \wedge (\hat{l} \Rightarrow \widetilde{l} = \widetilde{f})) \text{ or}$$
$$\textstyle\bigwedge_{m.s \in M} (\widetilde{s} = \widetilde{s^c} \Rightarrow m.s = m.s^c) \wedge \bigwedge_{\hat{l}, \widetilde{l} \in S \cup L} (\hat{l} \Leftrightarrow \hat{f} \wedge (\hat{l} \Rightarrow \widetilde{l} = \widetilde{f}) \wedge \widetilde{f}_0)$$

To solve the validity of the formula $(\Phi(\mathtt{C}) \Rightarrow \Phi(\mathtt{A}))$ in Theorem 1 under the variable mapping, a Smt solver is needed since this formula involves non-Boolean variables and *uninterpreted functions* (using a Sat solver would not be sufficient). A Smt solver decides the satisfiability of arbitrary logic formulas of linear real and integer arithmetic, scalar types, other user-defined data structures, and uninterpreted functions. If the formula belongs to the decidable theory, the solver gives two types of answers: `sat` when the formula has a model (there exists an interpretation that satisfies it); or `unsat`, otherwise. In our case, we will ask the solver to check whether the formula $\neg(\Phi(\mathtt{C}) \wedge \widehat{X_A} \setminus \widehat{X_{IO}} = \alpha(\widehat{X_C} \setminus \widehat{X_{IO}}) \Rightarrow \Phi(\mathtt{A}))$ is unsatisfiable, since this formula is unsatisfiable iff $\models (\Phi(\mathtt{C}) \wedge \widehat{X_A} \setminus \widehat{X_{IO}} = \alpha(\widehat{X_C} \setminus \widehat{X_{IO}}) \Rightarrow \Phi(\mathtt{A}))$. In our translation validation, the clock models which are constructed from Boolean or numerical variables and uninterpreted functions belong to a part of first-order logic which has a *small model* property according to [3]. The numerical variables are involved only in some implications with *uninterpreted functions* such as $(\widetilde{x} = \widetilde{x'} \wedge \widetilde{y} = \widetilde{y'}) \Rightarrow \widetilde{v^i_\square} = \widetilde{v^j_\square}$.

In addition, the formula is quantifier-free. This means that the check of satisfiability can be established by examining a certain finite cardinality of models. Therefore, the formula can be solved efficiently and significantly improves the scalability of the solver.

5 Implementation

This section describes the implementation of our validator and some adaptation when the translation validation is applied to the real Signal compiler. We also show the previously unknown bugs have been detected so far by our validator.

5.1 Towards Certified Compiler

Given a program P, with an unverified compiler, the compilation process can be represented in the following pseudo-code, where $Cp(\texttt{P})$ is the compilation step from the source program P to either compiled code $IR(\texttt{P})$ or compilation errors:

```
if (Cp(P) is Error) then
   output Error;
else output IR(P);
```

Now, the compilation is followed by our refinement verification which checks that the transformed program $IR(\texttt{P})$ refines P w.r.t. the clock semantics:

```
if (Cp(P) is Error) then
   output Error;
else if ($\Phi$(IR(P)) $\sqsubseteq_{clk}$ $\Phi$(P)) then
   output IR(P);
else output Error;
```

This will provide a formal guarantee as strong as that provided by a certified compiler in case the correctness of the validator is proved. We describe the main components of the implementation which is integrated in the existing Polychrony toolset [19] to prove the preservation of clock semantics of the Signal compiler. We are interested here in the first phase: *clock calculation* and *Boolean abstraction* where the intermediate forms of the source program of are expressed in the Signal language itself.

At a high level, our validator, which is depicted in Figure 2, works as follows. First, it takes the input program P and its transformed counterpart P_BASIC_TRA, and constructs the corresponding clock models. These clock models are combined as the formula $(\Phi(\texttt{P\_BASIC\_TRA}) \Rightarrow \Phi(\texttt{P}))$. In the solving phase, it checks the validity of the formula $\Phi(\texttt{P\_BASIC\_TRA}) \Rightarrow \Phi(\texttt{P})$. The result of this check can be exploited for the preservation of clock semantics of the transformations. If the formula is not valid then it emits a compiler bug. Otherwise, the compiler continues its work. The same procedure is applied for the other steps of the compiler. Finally, our verification process asserts that $\Phi(\texttt{P\_BOOL\_TRA}) \sqsubseteq_{clk} \Phi(\texttt{P\_BASIC\_TRA}) \sqsubseteq_{clk} \Phi(\texttt{P})$ along the transformations of the compiler. We delegate the checking of the clock refinement to a Smt solver. For our experiments, we consider the Yices [9] solver, which is one of the best solvers at the Smt-Comp competition [23].

5.2 Detected Bugs

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

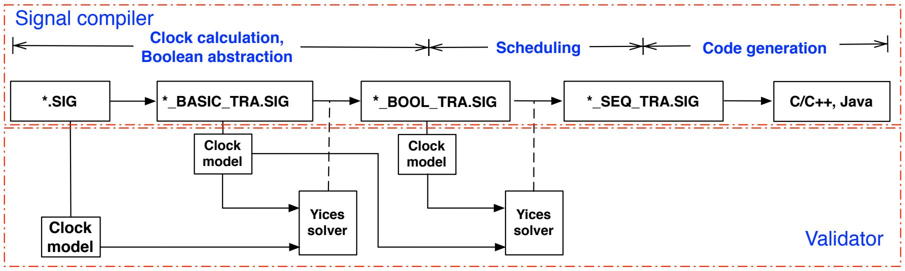

Fig. 2: The Integration within POLYCHRONY Toolset

constraints condition a clock such as in the following segment of SIGNAL program and its clock calculation part in transformed programs:

```
// P_BASIC_TRA.SIG            // P_BOOL_TRA.SIG
| CLK_x := when (y <= 9)      | when Tick ^= C_z ^= C_CLK
| CLK := when (y >= 1)        | when C_z ^= x ^= z
| CLK_x ^= CLK                | C_z := y <= 9
| CLK ^= XZX_24               | C_CLK := y >= 1
// P.SIG
| x ^= when (y <= 9)
| x ^= when (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, the signal x might be absent when `XZX_24` is absent, which is not the case in `P`, nor in `P_BOOL_TRA`). This bug was caught by our validator when it found that $\Phi(\texttt{P\_BOOL\_TRA}) \not\sqsubseteq_{clk} \Phi(\texttt{P\_BASIC\_TRA})$.

The second problem is the wrong implementation of `xor` operator as shown in the followng program. The validator detects this bug with the fact that $\Phi(\texttt{P\_BASIC\_TRA}) \not\sqsubseteq_{clk} \Phi(\texttt{P})$.

```
// P.SIG                       // P_BASIC_TRA.SIG
| b3 := (true xor true) and b1 | CLK_b1 := ^b1
                               | CLK_b1 ^= b1 ^= b3 | b3 := b1
```

The last problem detected was not found by the translation validation but was indirectly discovered when trying to apply it. It occurred in a program in which a *merge* operator with a constant signal was used, such as $y :=$ 1 `default` x. In this case, the code generation phase of the compiler dealt wrongly with the *clock context* of a constant signal by introducing a syntax error in the generated C code. The bug and its fix are given by:

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

6 Related Work and Conclusion

The notion of translation validation was introduced in [20,21] 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 constraints of the whole program and its generated C code. Then they use Bdd [6] representations to implement the symbolic Sts models, and their proof method uses a solver to reason on constraints over signals. The drawback of this approach is that it does not capture explicitly the clock semantics. 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 corresponding generated code is considered as one atomic transition in Sts, thus it does not capture the data dependencies between signals and does not explicitly prove the preservation of abstract clocks in the compiler transformations.

Another related work is the static analysis of Signal programs for efficient code generation [12]. 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 [16], 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. [15,7] 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 [25]. Tristan et al. [24] 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.

With the same purpose, in the work of [17], we encode the source Signal programs and their transformations with *Polynomial Dynamical Systems* (Pds), 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 solver implementation in processing formulas over Boolean variables and uninterpreted functions, our approach can deal with large programs whose number of variables is large. This situation generally makes the state-space explosion problem in model checking techniques.

The present paper provides a proof of correctness of a the synchronous dataflow compiler. We have presented a technique based on Smt solving to prove the preservation of clock semantics during the compilation. Namely, we have shown that implicit clock relations, describing the discrete timing model of a dataflow specification, are preserved in their implementation. The desired behavior of a given source program and the transformed one are represented as clock models. A refinement relation between source and transformed programs is used to express the preservation, which is checked by using a Smt solver. We have constructed and integrated our validator within the Polychrony toolset to prove the correctness of the Signal compiler.

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 libraries [9] 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.

References

1. W. Ackerman, *Solvable Cases of the Decision Problem.* Study in Logic and the Foundations of Mathematics. North-Holland, Amsterdam, 1954.
2. G. Berry, *The Foundations of Esterel.* In Proof, Language and Interaction: Essay in Honor of Robin Milner, MIT Press, 2000.
3. E. Borger, E. Gradel, and Y. Gurevich, *The Classical Decision Problem.* Spinger-Verlag, 1996.
4. L. Besnard, T. Gautier, P. Le Guernic, and J-P. Talpin, *Compilation of Polychronous Data Flow Equations.* In Synthesis of Embedded Software, Springer, 2010.
5. A. Benveniste and P. LeGuernic, *Hybrid Dynamical Systems Theory and the Signal Language.* IEEE Transactions on Automatic Control. 35(5):535-546, May 1990.
6. R. Bryant, *Graph-based Algorithms for Boolean Function Manipulation.* IEEE Transactions on Computers, C-35(8):677-691, Aug 1986.
7. Inria, *The CompCert Project.* http://compcert.inria.fr.

8. Inria, *The Coq Proof Assitant.* http://coq.inria.fr.
9. B. Dutertre, and L. de Moura, *Yices Sat-solver.* http://yices.csl.ri.com, 2009.
10. A. Gamatié, *Designing Embedded Systems with the Signal Programming Language: Synchronous, Reactive Specification.* Springer, New York. ISBN 978-1-4419-0940-4, 2009.
11. T. Gautier and P. Le Guernic and L. Besnard, *Signal, a Declarative Language for Synchronous Programming of Real-time Systems.* Proc. 3rd. Conf. on Functional Programming Languages and Computer Architecture, LNCS 274, May 1990.
12. 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.
13. 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.
14. 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.
15. 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.
16. 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.
17. 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.
18. V. C. Ngo. *Formal Verification of a Synchronous Data-flow Compiler: from Signal to C.* In PhD thesis, 2014.
19. Inria/Espresso, *Polychrony Toolset.* http://www.irisa.fr/espresso/Polychrony.
20. A. Pnueli, M. Siegel, and E. Singerman, *Translation Validation.* In B. Steffen, editor, 4th Intl. Conf. TACAS'98. LNCS 1384. pp.151-166, 1998.
21. 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.
22. RTCA, *DO-178C.* http://rtca.org
23. A. Stump, and M. Deters, *SMT-Comp.* http://www.smtcomp.org/2009, 2009.
24. 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, June 2011.
25. 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.

Appendix A

Constant clock abstraction

We describe the additional features that have to be considered when the translation validation is applied on real Signal programs. In Signal, the occurrence of constants is allowed to designate a constant signal (i.e., a signal with a constant value). However, each occurrence of a constant has a particular clock since the corresponding signal is hidden, and this clock is determined by the context where the constant is used, called *context clock*. This makes our abstraction for Signal operators above invalid in case a constant signal is used. In consequence of that, we have to provide new definitions of the abstraction when the operators use a constant signal (`cst` denotes a constant):

Stepwise functions

- $\phi(y := cst) = \hat{y} \Rightarrow (\tilde{y} \Leftrightarrow cst)$ if y is Boolean signal.
- $\phi(y := cst) = \hat{y} \Rightarrow (\tilde{y} = cst)$ if y is non-Boolean signal, where cst is an interpreted function.
- $\phi(y := x \texttt{ and } cst) = (\hat{y} \Leftrightarrow \hat{x}) \wedge (\hat{y} \Rightarrow (\tilde{y} \Leftrightarrow \tilde{x} \wedge cst))$.
- $\phi(y := x \texttt{ or } cst) = (\hat{y} \Leftrightarrow \hat{x}) \wedge (\hat{y} \Rightarrow (\tilde{y} \Leftrightarrow \tilde{x} \vee cst))$.
- $\phi(y := x \, \Box \, cst) = (\hat{y} \Leftrightarrow \widehat{v^i_\Box} \Leftrightarrow \hat{x}) \wedge (\hat{y} \Rightarrow (\tilde{y} = \widetilde{v^i_\Box}))$.

Merge

$$\phi(y := x \texttt{ default } cst) = (\hat{y} \Leftrightarrow (\hat{x} \vee \hat{y})) \wedge (\hat{y} \Rightarrow (\hat{x} \wedge \tilde{y} = \tilde{x} \vee \neg\hat{x} \wedge \tilde{y} = cst))$$
$$\phi(y := cst \texttt{ default } x) = (\hat{y} \Leftrightarrow (\hat{x} \vee \hat{y})) \wedge (\hat{y} \Rightarrow (\hat{y} \wedge \tilde{y} = cst \vee \neg\hat{y} \wedge \tilde{y} = \tilde{x}))$$

Sampling

$$\phi(y := x \texttt{ when } \texttt{true}) = (\hat{y} \Leftrightarrow (\hat{x} \wedge \hat{y})) \wedge (\hat{y} \Rightarrow (\tilde{y} = \tilde{x}))$$
$$\phi(y := x \texttt{ when } \texttt{false}) = \hat{y} \Leftrightarrow \texttt{ false}$$
$$\phi(y := cst \texttt{ when } b) = (\hat{y} \Leftrightarrow (\hat{b} \wedge \tilde{b})) \wedge (\hat{y} \Rightarrow (\tilde{y} = cst))$$

Clock semantics of the basic processes

Table 1 shows the clock semantics of the Signal processes corresponding to the primitive operators. In the clock semantics of the *delay* operator, inf is the infimum (or the *greatest lower bound*) of the subset of instants such that x is present, *sup* is the supremum (or *least upper bound*) of the subset of instants such that they are smaller than the instant t and x is present.

For instance, the clock semantics of the basic process corresponding to *sampling* operator is the following set of clock traces, where x, y are numerical signals and b is Boolean signal:

$$T_c = \{(0, (x_0, b_0, y)), ..., (i, (x_i, b_i, y_i)), ...\} \text{ such that } \forall i, (x_i, b_i, y_i) \in$$
$$\{(\bot, \bot, \bot), (\bot, \texttt{false}, \bot), (\bot, \texttt{true}, \bot), (v, \bot, \bot), (v, \texttt{false}, \bot), (v, \texttt{true}, v)\}$$

Process P	Clock semantics $[[P]]_c$
$y := f(x_1, ..., x_n)$	$\{T_c \in \mathcal{T}c_{\{y,x_1,...,x_n\}} \mid \forall t \in \mathbb{N}, (\forall i, T_c(t)(x_i) = T_c(t)(y) = \bot)$ or $T_c(t)(y) = v_y$ and $T_c(t)(x_i) = v_{x_i}$ and $v_y = f(v_{x_1}, ..., v_{x_n})\}$
$y := x\$1$ `init` a	$\{T_c \in \mathcal{T}c_{\{x,y\}} \mid \forall t \in \mathbb{N}, (T_c(t)(x) = T_c(t)(y) = \bot)$ or $(T_c(t)(x) \neq \bot$ and $T_c(t)(y) \neq \bot$ and $T_c(t_0)(y) = a$ and $\forall t \geq t_0, T_c(t)(y) = T_c(t_-)(x))$ with $t_0 = inf\{t' \mid T_c(t')(x) \neq \bot\}$, $t_- = sup\{t' \mid t' < t \wedge T_c(t')(x) \neq \bot\}\}$
$y := x$ `when` b	$\{T_c \in \mathcal{T}c_{\{x,y,b\}} \mid \forall t \in \mathbb{N}, (T_c(t)(b) =$ `true` and $T_c(t)(y) = T_c(t)(x) = \bot)$ or $(T_c(t)(b) =$ `true` and $T_c(t)(y) = T_c(t)(x) = v)$ or $(T_c(t)(b) =$ `false` and $T_c(t)(y) = T_c(t)(x) = \bot)$ or $(T_c(t)(b) =$ `false` and $T_c(t)(x) = v$ and $T_c(t)(y) = \bot)$ or $(T_c(t)(b) = \bot$ and $T_c(t)(x) = T_c(t)(y) = \bot)$ or $(T_c(t)(b) = \bot$ and $T_c(t)(x) = v$ and $T_c(t)(y) = \bot)\}$
$y := x$ `default` z	$\{T_c \in \mathcal{T}c_{\{x,y,z\}} \mid \forall t \in \mathbb{N}, (T_c(t)(x) = v$ and $T_c(t)(y) = v)$ or $(T_c(t)(x) = \bot$ and $T_c(t)(z) = v$ and $T_c(t)(y) = v)$ or $(T_c(t)(x) = T_c(t)(z) = T_c(t)(y) = \bot)\}$
$P_1 \mid P_2$	$\{T_c \in \mathcal{T}c_{X_1 \cup X_2} \mid X_1.T_c \in [[P_1]]_c$ and $X_2.T_c \in [[P_2]]_c\}$ where $[[P_1]]_c \subseteq \mathcal{T}c_{X_1}, [[P_2]]_c \subseteq \mathcal{T}c_{X_2}$
P_1/x	$\{T_c \in \mathcal{T}c_{X_1 \setminus \{x\}} \mid \exists T_{c1} \in [[P_1]]_c, (X_1 \setminus \{x\}).T_{c1} = T_c\}$ where $[[P_1]]_c \subseteq \mathcal{T}c_{X_1}$

Table 1: Clock semantics of the Signal primitive operators

Soundness of the abstraction

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

To show the soundness of our abstraction, we consider a similar reasoning as in [12]. 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 $\mathit{\Gamma}(\varphi)$ is given by:

$$\begin{aligned} S_{\mathcal{E}c_X}(\varphi) &= \textstyle\bigcup_{\hat{I} \models \varphi} S_{\mathcal{E}c_X}(\hat{I}) \\ \mathit{\Gamma}(\varphi) &= \{T_c \in \mathcal{T}c_X \,|\, \forall t, T_c(t) \in S_{\mathcal{E}c_X}(\varphi)\} \end{aligned}$$

Proposition 2. *Let P, $\mathit{\Phi}(P)$ be a program and its abstraction, respectively, and φ is a property defined over the clocks. If $\mathit{\Phi}(P) \models \varphi$ then $[[P]]_c \subseteq \mathit{\Gamma}(\varphi)$.*

Proof. The proof of Proposition 2 is done by using Lemma 1. Given a clock trace $T_c \in [[P]]_c$, applying Lemma 1, $T_c \in \mathit{\Gamma}(\mathit{\Phi}(P))$ means that $\forall t, T_c(t) \in S_{\mathcal{E}c_X}(\mathit{\Phi}(P))$. Since $\mathit{\Phi}(P) \models \varphi$, then every interpretation $\hat{I}$ satisfying $\mathit{\Phi}(P)$ also satisfies φ. Thus, any clock event $I \in S_{\mathcal{E}c_X}(\mathit{\Phi}(P))$ is also in $S_{\mathcal{E}c_X}(\varphi)$, meaning that $\forall t, T_c(t) \in S_{\mathcal{E}c_X}(\varphi)$. Therefore, we have $T_c \in \mathit{\Gamma}(\varphi)$.

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

Proof. 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 functions: $P : y := f(x_1, ..., x_n)$. First, consider y as numerical signal; following the encoding scheme, we have $\mathit{\Phi}(P) = (\hat{y} \Leftrightarrow \widehat{v_f^k} \Leftrightarrow \widehat{x_1} \Leftrightarrow ... \Leftrightarrow \widehat{x_n}) \wedge (\hat{y} \Rightarrow \tilde{y} = \widetilde{v_f^k})$. For any interpretation $\hat{I}$ such that $\hat{I} \models \mathit{\Phi}(P)$, we have:
  - either $\forall i, \hat{y} = \texttt{false}, \widehat{v_f^k} = \texttt{false}$ and $\widehat{x_i} = \texttt{false}$;
  - or $\hat{y} = \texttt{true}, \widehat{v_f^k} = \texttt{true}, \forall i. \widehat{x_i} = \texttt{true}$ and $\tilde{y} = \widetilde{v_f^k}$.

  Then the set of clock events according to $\hat{I}$ is given as follows:

  $$\begin{aligned} S_{\mathcal{E}c_X}(\hat{I}) = \{I \in \mathcal{E}c_{\{y,x_1,\ldots,x_n,v_f^k\}} |\; & I(y) = \hat{I}(\tilde{y}) \wedge \forall i, I(x_i) = \hat{I}(\widetilde{x_i}) \text{ if } \hat{I}(\hat{y}) = \\ & \hat{I}(\widehat{v_f^k}) = \hat{I}(\widehat{x_i}) = \texttt{true} \text{ and } I(y) = I(x_i) = \bot \text{ if } \hat{I}(\hat{y}) = \hat{I}(\widehat{v_f^k}) = \\ & \hat{I}(\widehat{x_i}) = \texttt{false}\} \end{aligned}$$

  Let $T_c \in [[P]]_c$ be a clock trace and $t \in \mathbb{N}$, then either $\forall i, T_c(t)(y) = T_c(t)(x_i) = \bot$ or $T_c(t)(y) = f(T_c(t)(x_1), T_c(t)(x_2), ..., T_c(t)(x_n))$. We have $T_c(t) \in S_{\mathcal{E}c_X}(\mathit{\Phi}(P))$ for every instant t. Therefore, $T_c \in \mathit{\Gamma}(\mathit{\Phi}(P))$. When y is a Boolean signal, the proof is similar.

- Delay, sampling, and merging operators: we prove in the same manner.
- 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 $\forall t, T_c(t) \in S_{\mathcal{E}c_X}(\Phi(P_1))$ and $T_c(t) \in S_{\mathcal{E}c_X}(\Phi(P_2))$. That means $\forall t, T_c(t) \in S_{\mathcal{E}c_X}(\Phi(P_1) \wedge \Phi(P_2))$, or $T_c \in \Gamma(\Phi(P))$.

Appendix B

Let us illustrate the above process on the program DEC, and its transformed program DEC_BASIC_TRA in Listing 1.1 at the first phase of the compilation process.

Listing 1.1: DEC_BASIC_TRA in Signal

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

In the first step, we will construct the clock models which are presented by the following first-order logic formulas $\Phi(\mathtt{DEC})$ and $\Phi(\mathtt{DEC\_BASIC\_TRA})$, respectively, where $\mathtt{ZN} <= \mathtt{1}, \mathtt{ZN}^{\mathtt{c}} <= \mathtt{1}, \mathtt{ZN} - \mathtt{1}$ and $\mathtt{ZN}^{\mathtt{c}} - \mathtt{1}$ are replaced by the fresh variables $\mathtt{ZN1}, \mathtt{ZN1}^{\mathtt{c}}, \mathtt{ZN2}$ and $\mathtt{ZN2}^{\mathtt{c}}$. The uninterpreted function symbols $\mathtt{v}^{\mathtt{1}}_{<=}, \mathtt{v}^{\mathtt{1c}}_{<=}, \mathtt{v}^{\mathtt{1}}_{-}$ and $\mathtt{v}^{\mathtt{1c}}_{-}$ are used to encode the stepwise functions $\mathtt{ZN1} := \mathtt{ZN} <= \mathtt{1}, \mathtt{ZN1}^{\mathtt{c}} := \mathtt{ZN}^{\mathtt{c}} <= \mathtt{1}, \mathtt{ZN2} := \mathtt{ZN} - \mathtt{1}$ and $\mathtt{ZN2}^{\mathtt{c}} := \mathtt{ZN}^{\mathtt{c}} - \mathtt{1}$:

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

$$\begin{aligned}
&\Phi(\mathtt{DEC\_BASIC\_TRA}) = \\
&(\widehat{CLK^c} \Leftrightarrow \widehat{CLK\_N^c} \wedge \neg\widehat{CLK\_FB^c}) \wedge (\widehat{CLK^c} \Rightarrow \widetilde{CLK^c}) \\
\wedge\ &(\widehat{CLK\_N^c} \Leftrightarrow \widehat{CLK\_N^c} \vee \widehat{CLK\_FB^c}) \wedge (\widehat{CLK\_N^c} \Rightarrow \widetilde{CLK\_N^c}) \\
\wedge\ &(\widehat{CLK\_N^c} \Leftrightarrow \widehat{N} \Leftrightarrow \widehat{ZN^c}) \\
\wedge\ &(\widehat{N} \Leftrightarrow \widehat{FB} \wedge \widehat{CLK\_FB^c} \wedge \widetilde{CLK\_FB^c} \vee \widehat{ZN2^c} \wedge \widehat{CLK^c} \wedge \widetilde{CLK^c}) \\
\wedge\ &(\widehat{N} \Rightarrow (\widehat{FB} \wedge \widehat{CLK\_FB^c} \wedge \widetilde{CLK\_FB^c} \wedge \tilde{N} = \widetilde{FB} \\
&\vee\neg(\widehat{FB} \wedge \widehat{CLK\_FB^c} \wedge \widetilde{CLK\_FB^c}) \wedge \tilde{N} = \widetilde{ZN2^c})) \\
\wedge\ &(\widehat{ZN2^c} \Leftrightarrow \widehat{v^{1c}_{-}} \Leftrightarrow \widehat{ZN^c}) \wedge (\widehat{ZN2^c} \Rightarrow (\widetilde{ZN2^c} = \widetilde{v^{1c}_{-}})) \\
\wedge\ &(\widehat{ZN^c} \Leftrightarrow \widehat{N}) \wedge (\widehat{ZN^c} \Rightarrow (\widetilde{ZN^c} = m.N \wedge m.N' = \tilde{N})) \wedge (m.N_0 = 1 \\
\wedge\ &(\widehat{CLK\_FB^c} \Leftrightarrow \widehat{ZN1^c} \wedge \widetilde{ZN1^c}) \wedge (\widehat{CLK\_FB^c} \Rightarrow \widetilde{CLK\_FB^c}) \\
\wedge\ &(\widehat{ZN1^c} \Leftrightarrow \widehat{v^{1c}_{<=}} \Leftrightarrow \widehat{ZN^c}) \wedge (\widehat{ZN1^c} \Rightarrow (\widetilde{ZN1^c} = \widetilde{v^{1c}_{<=}})) \\
\wedge\ &(\widehat{CLK\_FB^c} \Leftrightarrow \widehat{FB}) \\
\wedge\ &(\widehat{CLK\_12^c} \Leftrightarrow \widehat{ZN1^c} \wedge \neg\widetilde{ZN1^c}) \wedge (\widehat{CLK\_12^c} \Rightarrow \widetilde{CLK\_12^c})
\end{aligned}$$

In the second step, checking formula construction, our tool will establish the variable mapping $\widehat{X_{\mathtt{DEC}}} \setminus \widehat{X_{IO}} = \alpha(\widehat{X_{\mathtt{DEC\_BASIC\_TRA}}} \setminus \widehat{X_{IO}})$ as follows:

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

In the third step, we delegate the checking validity of the formula

$$(\Phi(\mathtt{DEC\_BASIC\_TRA}) \wedge \widehat{X_{\mathtt{DEC}}} \setminus \widehat{X_{IO}} = \alpha(\widehat{X_{\mathtt{DEC\_BASIC\_TRA}}} \setminus \widehat{X_{IO}}) \Rightarrow \Phi(\mathtt{DEC}))(\text{named } \varphi)$$

to the Smt solver under the logical context defined by the variable mapping and the following assertions:

$$\begin{aligned}
(\widetilde{ZN} = \widetilde{ZN^c} \wedge 1 = 1) \Rightarrow \widetilde{v^{1}_{<=}} = \widetilde{v^{1c}_{<=}} \\
(\widetilde{ZN} = \widetilde{ZN^c} \wedge 1 = 1) \Rightarrow \widetilde{v^{1}_{-}} = \widetilde{v^{1c}_{-}}
\end{aligned}$$

With the Yices solver, we will get `unsat` when checking the satisfiability of $\neg\varphi$, which means that φ is valid. Thus, we can conclude that the transformation is correct.