Translation Validation for Transformations on Abstract Clocks in Synchronous Languages

Van Chan Ngo, Jean-Pierre Talpin, Thierry Gautier, Paul Le Guernic
Translation Validation for Transformations on Abstract Clocks in Synchronous Languages

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

Project-Team ESPRESSO

Research Report n° 8064 — March 2013 — 26 pages

Abstract: Translation validation was introduced as a technique to formally verify the correctness of code generators that attempts to verify that program transformations preserve the semantics. In this work, we adopt this approach to formally verify that the clock semantics is preserved during the transformations of a synchronous data-flow compiler. We represent the clock semantics of a program and its transformed counterpart as first-order formulas which are called clock models. Then we introduce a refinement relation 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.

Key-words: Formal Verification, Translation Validation, Certified Compiler, SMT solver, Synchronous Data-flow Languages
Validation de Traduction des Transformations sur le Horloges dans les Langues Synchrones

Résumé : Translation validation was introduced as a technique to formally verify the correctness of code generators that attempts to verify that program transformations preserve the semantics. In this work, we adopt this approach to formally verify that the clock semantics is preserved during the transformations of a synchronous data-flow compiler. We represent the clock semantics of a program and its transformed counterpart as first-order formulas which are called clock models. Then we introduce a refinement relation 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.

Mots-clés : Formal Verification, Translation Validation, Certified Compiler, SMT solver, Synchronous Data-flow Languages
1 Introduction

The synchronous languages such as Esterel [3], Lustre [15] and Signal [12] 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 [10], or check that it is compliant to DO-178C documents [22]. 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. [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. Some other works have adopted the translation validation approach in verification of transformations, and optimizations. In [9, 17], 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 and makes Boolean abstraction. The next phase is static scheduling and 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 the case of a synchronous compiler such as that of the Signal language, the preservation of the semantics can be decomposed into the preservation of clock semantics, data dependencies, and value-equivalence of variables. As first contribution to this work, this paper focuses on proving 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 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. 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. 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 [11] to achieve the expected goals: traceability and formal evidence.

Regarding the rest of the work for the overall validation, 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 [24], 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 dependencies.

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 V. Section 6 presents related works, concludes our work and outlines future directions.
2 The Signal Language

2.1 Language Features

Signal \[6, 13\] is a polychronous data-flow language that allows the specification of multi-clocked systems. Signal handles unbounded sequences of typed values \((x(t))_{t\in\mathbb{N}}\), called \textit{signals}, denoted as \(x\). Each signal is implicitly indexed by a logical \textit{clock} indicating the set of instants at which the signal is present, noted \(C_x\). At a given instant, a signal may be present where it holds a value, or absent where it holds no value (denoted by \#). Given two signals, they are \textit{synchronous} if and only if they have the same clock. In Signal, a process (written \(P\) or \(Q\)) consists of the synchronous composition (noted \(|\) ) of equations over signals \(x, y, z\), written \(x := y \text{ op } z\) or \(x := \text{op}(y, z)\), where \textit{op} is an operator. A program is a process.

2.1.1 Data domains

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

2.1.2 Operators

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

- **Stepwise Functions**: \(y := f(x_1, ..., x_n)\), where \(f\) is a \(n\)-ary function on values, defines the extended stream function over synchronous signals as a basic process whose output \(y\) is synchronous with \(x_1, ..., x_n\) and \(\forall t \in C_y, y(t) = f(x_1(t), ..., x_n(t))\). The implicit clock relation is \(C_y = C_{x_1} = ... = C_{x_n}\).

- **Delay**: \(y := x \text{ op } 1 \text{ init } a\) defines a basic process such that \(y\) and \(x\) are synchronous, \(y(0) = a\), and \(\forall t \in C_y \wedge t > 0, y(t) = x(t - 1)\). The implicit clock is \(C_y = C_x\).

- **Merge**: \(y := x \text{ default } z\) defines a basic process which specifies that \(y\) is present if and only if \(x\) or \(z\) is present, and that \(y(t) = x(t)\) if \(t \in C_x\) and \(y(t) = z(t)\) if \(t \in C_z \setminus C_x\). The implicit clock relation is \(C_y = C_x \cup C_z\).

- **Sampling**: \(y := x \text{ when } b\) where \(b\) is a Boolean signal, defines a basic process such that \(\forall t \in C_x \cap C_b \wedge b(t) = \text{true}, y(t) = x(t)\), and otherwise, \(y\) is absent. The implicit clock relation is \(C_y = C_x \cap [b]\), where the sub-clock \([b]\) is defined as \(\{t \in C_b | b(t) = \text{true}\}\).

- **Composition**: If \(P_1\) and \(P_2\) are processes, then \(P_1 \parallel P_2\), also denoted \(\langle P_1 \parallel P_2 \rangle\), is the process resulting of their parallel composition. This process consists of the composition of the systems of equations. The composition operator is commutative, associative, and idempotent.

- **Restriction**: \(P\ where \ x\), where \(P\) is a process and \(x\) is a signal, specifies a process by considering \(x\) as local variable to \(P\) (i.e., \(x\) is not accessible from outside \(P\)).
2.1.3 Clock relations

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

2.1.4 Example

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

```plaintext
process DEC =
( ? integer FB;
 ! integer N )
( | FB \dot{=} \text{when } (ZN <= 1)
 | N := FB \text{ default } (ZN - 1)
 | ZN := N\$1 \text{ init } 1
 )
where integer ZN init 1
end;
```

Let us comment this program: \( ? \text{ integer } FB; ! \text{ integer } N \): \( FB, N \) are respectively input and output signals of type \text{integer}; \( FB \dot{=} \text{when } (ZN <= 1) \): \( FB \) 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); \( \text{where integer } ZN \text{ init } 1 \): 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  |
| N  | 6 5 4 3 2 1 3 2 1  |

| \( C_{FB} \) | \( t_0 \) | \( t_6 \) | \( t_9 \) |
| \( C_{ZN} \) | \( t_0 \) | \( t_1 \) | \( t_2 \) | \( t_3 \) | \( t_4 \) | \( t_5 \) | \( t_6 \) | \( t_7 \) | \( t_8 \) | \( t_9 \) |

| \( C_{R} \) | \( t_0 \) | \( t_1 \) | \( t_2 \) | \( t_3 \) | \( t_4 \) | \( t_5 \) | \( t_6 \) | \( t_7 \) | \( t_8 \) | \( t_9 \) |

2.1.5 Program Structure

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

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

### 2.2 Compilation of Signal Programs

The Signal compiler [5] 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 which are produced in the first phase (clock calculation and Boolean abstraction) 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 [5]. 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 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:

RR n° 8064
\begin{verbatim}
( | 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))
| )
| )
\end{verbatim}
3 Clock Model

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 $\text{true}$; otherwise, it holds no value. Thus, to represent the underlying control conditions, we need to model the statuses present with value $\text{true}$ or $\text{false}$ 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 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 $x$: $(\hat{x} = \text{true})$ means $x$ is present, and $(\hat{x} = \text{false})$ 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: $(\hat{b} = \text{true} \land \bar{b} = \text{true})$ means $b$ is present and holds a value $\text{true}$; $(\hat{b} = \text{true} \land \bar{b} = \text{false})$ means $b$ is present and holds a value $\text{false}$; and $(\hat{b} = \text{false})$ 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:

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

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 $\hat{x}_i$ to encode its clock and a variable $\bar{x}_i$ 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:

- $\hat{\cdot} : X \rightarrow \mathbb{B}$ associates a signal with a Boolean value;
- $\bar{\cdot} : X \rightarrow \mathbb{D}$ 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 $\hat{x}_i, \bar{x}_i$, 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 Function

The functions which apply on signal values in the primitive stepwise functions are usual logic operators ($\text{not}$, $\text{and}$, $\text{or}$), 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 $v_i$ 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 $v_i$ iff $x, y$ are identical to $x'$ and $y'$, respectively.
- For every pair of newly added variables $v_i$ and $v_j$, $i \neq j$, corresponding to the non-identical occurrences $x \square y$ and $x' \square y'$, add the implication $(x = x' \land y = y') \Rightarrow v_i = v_j$ into the abstraction $\Phi(P)$.

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

- $\phi(\text{true}) = \text{true}$ and $\phi(\text{false}) = \text{false}$.
- $\phi(y := x) = (\hat{y} \Leftrightarrow \hat{x}) \land (\hat{y} \Rightarrow ((\bar{y} \Leftrightarrow m.x) \land (m.x' \Leftrightarrow \bar{x})))$ if $x$ and $y$ are Boolean. $\phi(y := x) = (\hat{y} \Leftrightarrow \hat{x}) \land (\hat{x} \Rightarrow \bar{x})$ if $x$ is an event signal.
- $\phi(y := x_1 \land x_2) = (\hat{y} \Leftrightarrow \hat{x}_1 \Leftrightarrow \hat{x}_2) \land (\hat{y} \Rightarrow (\bar{y} \Leftrightarrow \bar{x}_1 \lor \bar{x}_2))$.
- $\phi(y := x_1 \lor x_2) = (\hat{y} \Leftrightarrow \hat{x}_1 \Leftrightarrow \hat{x}_2) \land (\hat{y} \Rightarrow (\bar{y} \Leftrightarrow \bar{x}_1 \lor \bar{x}_2))$.
- $\phi(y := x_1 \square x_2) = (\hat{y} \Leftrightarrow \hat{v}_i \Leftrightarrow \hat{x}_1 \Leftrightarrow \hat{x}_2) \land (\hat{y} \Rightarrow (\bar{y} = \bar{v}_i))$.

### 3.2.2 Delay

Considering the delay operator, $y := x$\_\$1$\ init $a$, its encoding $\phi(y := x$\_\$1$\ init $a$) contributes to $\Phi(P)$ with the following conjunct:

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

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

$$\phi(\text{true}) = \text{true}$$ and $\phi(\text{false}) = \text{false}$. 

- $\phi(y := x) = (\hat{y} \Leftrightarrow \hat{x}) \land (\hat{y} \Rightarrow ((\bar{y} \Leftrightarrow m.x) \land (m.x' \Leftrightarrow \bar{x})))$ 
  $$\land (m.x_0 \Leftrightarrow a)$$
- $\phi(y := x \land y) = (\hat{y} \Leftrightarrow \hat{x}_1 \Leftrightarrow \hat{x}_2) \land (\hat{y} \Rightarrow (\bar{y} \Leftrightarrow \bar{x}_1 \lor \bar{x}_2))$.
- $\phi(y := x_1 \lor x_2) = (\hat{y} \Leftrightarrow \hat{x}_1 \Leftrightarrow \hat{x}_2) \land (\hat{y} \Rightarrow (\bar{y} \Leftrightarrow \bar{x}_1 \lor \bar{x}_2))$.
- $\phi(y := x_1 \square x_2) = (\hat{y} \Leftrightarrow \hat{v}_i \Leftrightarrow \hat{x}_1 \Leftrightarrow \hat{x}_2) \land (\hat{y} \Rightarrow (\bar{y} = \bar{v}_i))$. 

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 $m.x_0$. 

Inria
• if $x, y$ and $z$ are Boolean:
  \[(\hat{y} \leftrightarrow (\hat{x} \lor \hat{z})) \land \hat{y} \Rightarrow ((\hat{x} \land (\hat{y} \leftrightarrow \hat{x})) \lor \neg \hat{x} \land ((\hat{y} \leftrightarrow \hat{z})))\]

• if $x, y$ and $z$ are non-Boolean:
  \[\hat{y} \leftrightarrow (\hat{x} \lor \hat{z})\]

### 3.2.4 Sampling

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

• if $x$ and $y$ are Boolean:
  \[(\hat{y} \leftrightarrow (\hat{x} \land \hat{b} \land \hat{b})) \land (\hat{y} \Rightarrow (\hat{x} \leftrightarrow \hat{x}))\]

• if $x$ and $y$ are non-Boolean:
  \[\hat{y} \leftrightarrow (\hat{x} \land \hat{b} \land \hat{b})\]

### 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:

• $\phi(P_1) \land \phi(P_2)$

### 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:

• $\phi(z := \bar{x}) = (\hat{z} \leftrightarrow \bar{x} \land (\hat{z} \Rightarrow \bar{z})$)

• $\phi(x^- = y) = \hat{x} \leftrightarrow \hat{y}$

• $\phi(z := x^- + y) = (\hat{z} \leftrightarrow (\hat{x} \lor \hat{y})) \land (\hat{z} \Rightarrow \bar{z})$)

• $\phi(z := x^- * y) = (\hat{z} \leftrightarrow (\hat{x} \land \hat{y})) \land (\hat{z} \Rightarrow \bar{z})$)

• $\phi(z := x^- - y) = (\hat{z} \leftrightarrow (\hat{x} \land \neg \hat{y})) \land (\hat{z} \Rightarrow \bar{z})$)

• $\phi(z := \text{when } b) = (\hat{z} \leftrightarrow (\hat{b} \land \hat{b}) \land (\hat{z} \Rightarrow \bar{z})$)

### 3.2.7 Nested processes

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

• $\phi(P) \land \phi(P_1)$

• 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 $P_1$:
  \[\bigwedge_{k=1}^{n}(\bar{y}_k \leftrightarrow \hat{o}_k \land \bar{y}_k \leftrightarrow \sigma_k) \land \bigwedge_{h=1}^{m}(\hat{x}_h \leftrightarrow \hat{i}_h \land \bar{x}_h \leftrightarrow \bar{i}_h)\]
Applying the abstraction rules above, the clock semantics of the Signal program DEC is represented by the following first-order logic formula \( \Phi(DEC) \), where \( ZN \leq 1 \) is replaced by \( v^1_{<e} \) and \( ZN - 1 \) is replaced by \( v^1_{\geq e} \).

\[
\begin{align*}
&\left( FB \iff v^1_{<e} \land v^1_{\geq e}\right) \\
&\land (v^1_{<e} \iff ZN) \\
&\land (ZN \iff \bar{N}) \\
&\land (\bar{N} \iff FB \lor v^1_{\geq e}) \\
&\land (v^1_{\geq e} \iff ZN)
\end{align*}
\]

### 3.3 Concrete Clock Semantics

Let \( X_B \subseteq X \) be the set of all Boolean or event signals. We rely on the basic elements of trace semantics [10] to define the clock semantics of a synchronous program.

**Definition** (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 \( \bar{T} \) for \( X_B \). The interpretations \( I, \bar{T} \) are respectively mappings from \( X^n \) to \( \mathbb{R}^n \) and from \( X_B^m \) to \( \mathbb{R}^m \), where \( I(x) = true \) if \( x \) holds a value while \( I(x) = false \) if it holds no value; and \( \bar{T}(x) = true \) if \( x \) holds the value \( true \), \( \bar{T}(x) = false \), otherwise.

For example, consider a program whose variables are \( X = \{x, b\} \) where \( b \) is Boolean variable, the set of clock events is \( \mathcal{E}_C X = \{(x \rightarrow_I false, b \rightarrow_I false, b \rightarrow_I false), (x \rightarrow_I false, b \rightarrow_I true, b \rightarrow_I true), (x \rightarrow_I true, b \rightarrow_I false, b \rightarrow_I false), (x \rightarrow_I true, b \rightarrow_I true, b \rightarrow_I true, b \rightarrow_I true)\} \). 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** (Clock traces) Given a non-empty set \( X \), the set of clock traces on \( X \), denoted by \( T_C X \), is defined by the set of functions \( T_c \) defined from the set \( N \) of natural numbers to \( \mathcal{E}_C X \), denoted by \( T_c : N \rightarrow \mathcal{E}_C X \).

The natural numbers represent the instants \( t = 0, 1, 2, \ldots \). A trace \( T_c \) 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 \( T_c(t)(x_i) \) if \( x_i \in X_B \). Considering the above example, we have \( T_c : (0, (x \rightarrow_I false, b \rightarrow_I false, b \rightarrow_I false)), (1, (x \rightarrow_I true, b \rightarrow_I true), b \rightarrow_I false)) \), ... as one of the possible clock traces on \( X \), and \( T_c(0)(x) = T_c(0)(b) = T_c(0)(\bar{b}) = false \).

**Definition** (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 \( X_1.T_c : N \rightarrow \mathcal{E}_C X_1 \) such that \( \forall t \in N, \forall x \in X_1, X_1.T_c(t)(x) = T_c(t)(x) \) and \( X_1.T_c(t)(x) = T_c(t)(x) \) if \( x \in X_B \).

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

Let \( \bar{X} = \{\bar{x}_1, \ldots, \bar{x}_n, \bar{x}_1, ..., \bar{x}_m\} \cup \bar{V} \cup \bar{V} \) 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 \( \bar{I} \) over \( \bar{X} \), it is called a clock configuration if it is a model of the first-order logic formula \( \Phi(P) \). For example, \((\bar{FB} \rightarrow false, \bar{N} \rightarrow true, \bar{ZN} \rightarrow true)\) is a clock configuration of \( \Phi(DEC) \), but \((\bar{FB} \rightarrow false, \bar{N} \rightarrow true, \bar{ZN} \rightarrow false)\) 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:

$$S_{sat}(\hat{I}) = \{ I \in \mathcal{E}_{\mathcal{C}X} | \forall i, I(x_i) = \hat{I}(\hat{x}_i) \wedge \hat{I}(x_i) = \hat{I}(\hat{x}_i) \text{ if } x_i \in X_B \}$$  \hfill (1)

$$S_{sat}(\Phi(P)) = \bigcup_{\hat{I} \models \Phi(P)} S_{sat}(\hat{I})$$  \hfill (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:

$$\Gamma(\Phi(P)) = \{ T_c \in \mathcal{T}_{\mathcal{C}X} | \forall t, T_c(t) \in S_{sat}(\Phi(P)) \}$$  \hfill (3)

We present the proof of soundness of our abstraction in Appendix A.
4 Clock Model Translation Validation

We adopt the translation validation approach \cite{20,21} to formally verify that the clock semantics are 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. Given two clock models $\Phi(P_1)$ and $\Phi(P_2)$, we write $\Phi(P_2) \sqsubseteq_{clk} \Phi(P_1)$ to denote that $\Phi(P_2)$ is a refinement of $\Phi(P_1)$.

4.1 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:

$$\forall T_c. (T_c \in \Gamma(\Phi(P_1)) \iff T_c \in \Gamma(\Phi(P_2)))$$  \hspace{1cm} (4)

In general, the compilation makes the transformed program more concrete. For instance, when the Signal compiler do 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 (4) is too strong to be practical. To address this issue, we relax the requirement as follows:

$$\forall T_c. (T_c \in \Gamma(\Phi(P_2)) \Rightarrow T_c \in \Gamma(\Phi(P_1)))$$  \hspace{1cm} (5)

Requirement (5) 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 4.1 The clock refinement is reflexive and transitive, or:

- $\forall \Phi(P), \Phi(P) \sqsubseteq_{clk} \Phi(P)$.
- 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 $\Phi(P_1) \sqsubseteq_{clk} \Phi(P_3)$.

4.2 Proving by SMT Solver

We now discuss an approach to check the existence of refinement between two clock models (Requirement \cite{4}) which is based on the following theorem.
Theorem 4.2 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 $I$, if $I$ is a clock configuration of $\Phi(P_2)$ then it is a clock configuration of $\Phi(P_1)$, or:

$$\left( \models \Phi(P_2) \Rightarrow \Phi(P_1) \right) \Rightarrow \Phi(P_2) \subseteq \text{clk} \ \Phi(P_1)$$

\[ (6) \]

Proof To prove Theorem 4.2, we show that if $\forall I. (I \models \Phi(P_2) \Rightarrow I \models \Phi(P_1))$ 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 $\forall I. (I \models \Phi(P_2) \Rightarrow I \models \Phi(P_1))$, 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 $T_c \in \Gamma(\Phi(P_1))$.

To solve the validity of the formula $(\Phi(P_2) \Rightarrow \Phi(P_1))$ in (6), 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 if $\models \Phi(P_2) \Rightarrow \Phi(P_1))$.

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 [1]. The numerical variables are involved only in some implication with uninterpreted functions such as $(\overline{x} = \overline{y} \land \overline{z} = \overline{y'}) \Rightarrow \overline{v_{<}} = \overline{v_{<}}$. 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.

4.3 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, $\Phi(\text{DEC BASIC TRA})$ as follows:

\[
\begin{align*}
(\overline{\text{CLK}} &\Leftrightarrow \overline{\text{CLK}_N} \land \overline{\text{CLK}_{FB}}) \land (\overline{\text{CLK}} \Rightarrow \overline{\text{CLK}}) \\
& \land (\overline{\text{CLK}_N} \Leftrightarrow \overline{\text{CLK}_N} \lor \overline{\text{CLK}_{FB}}) \\
& \land (\overline{\text{CLK}_N} \Rightarrow \overline{\text{CLK}_N}) \\
& \land \overline{\text{CLK}_N} \Leftarrow \overline{\text{N}} \Rightarrow \overline{\text{ZN}} \\
& \land (\overline{\text{N}} \Leftarrow \overline{\text{FB}} \land \overline{\text{CLK}_{FB}} \lor \overline{\text{v}_{<}} \land \overline{\text{CLK}}) \\
& \land (\overline{\text{v}_{<}} \Leftarrow \overline{\text{Z}} \overline{\text{N}}) \\
& \land (\overline{\text{ZN}} \Leftarrow \overline{\text{N}}) \\
& \land (\overline{\text{CLK}_{FB}} \Leftarrow \overline{\text{v}_{<}} \land \overline{\text{v}_{<}}) \land (\overline{\text{v}_{<}} \Rightarrow \overline{\text{ZN}}) \\
& \land \overline{\text{CLK}_{FB}} \Leftarrow \overline{\text{FB}} \\
& \land \overline{\text{CLK}_{12}} \Leftarrow \overline{\text{v}_{<}} \land \overline{\text{v}_{<}}
\end{align*}
\]

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 $\Phi(\text{DEC BASIC TRA}) \Rightarrow \Phi(\text{DEC})$. 

RR n° 8064
5 Implementation

5.1 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 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) then output Error; 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 semantics. 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) then output Error; else
2. if ($\Phi(IR(P)) \sqsubseteq_{ck} \Phi(P)$) then output IR(P); else output Error;

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 in the first phase: clock calculation and boolean abstraction. The intermediate forms in the transformations of the compiler may be expressed in the Signal language itself. At a high level, our tool 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 $(\Phi(P_{BASIC\_TRA}) \Rightarrow \Phi(P))$. In the solving phase, it checks the validity of the formula $\Phi(P_{BASIC\_TRA}) \Rightarrow \Phi(P)$ (or equivalently $M \models \neg(\Phi(P_{BASIC\_TRA}) \Rightarrow \Phi(P))$). The result of this check can be exploited for the preservation of clock semantics of the transformations. If the formula is not valid then it emits 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 $\Phi(P_{BOOL\_TRA}) \sqsubseteq_{ck} \Phi(P_{BASIC\_TRA}) \sqsubseteq_{ck} \Phi(P)$ along the transformations of the compiler.

Here, we delegate the checking of the clock refinement to a SMT solver. Our implementation uses the SMT-LIB common format [8] to encode the clock models as input of the SMT-solver. For our implementation, we consider the Yices [11] solver, which is one of the best solvers at the SMT-COMP competition [23].

5.2 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.

5.2.1 Stepwise Function

- \( \phi(y := cst) = \tilde{y} \Rightarrow (\tilde{y} \leftrightarrow cst) \) if \( y \) is Boolean.
- \( \phi(y := cst) = \emptyset \) if \( y \) is non-Boolean.
- \( \phi(y := x \text{ and } cst) = (\tilde{y} \leftrightarrow \tilde{x}) \land (\tilde{x} \Rightarrow (\tilde{y} \leftrightarrow \overline{cst})) \).
- \( \phi(y := x \text{ or } cst) = (\tilde{y} \leftrightarrow \tilde{x}) \land (\tilde{y} \Rightarrow (\tilde{y} \leftrightarrow \overline{cst})) \).
- \( \phi(y := x \Box cst) = (\tilde{y} \leftrightarrow \overline{v_i} \leftrightarrow \tilde{x}) \land (\tilde{y} \Rightarrow (\tilde{y} = v_i)) \).

5.2.2 Merge

- \( x \) and \( y \) are Boolean.
  \( \phi(y := x \text{ default } cst) = (\tilde{y} \leftrightarrow (\tilde{x} \lor \tilde{y}) \land \tilde{y} \Rightarrow ((\tilde{x} \land (\tilde{y} \leftrightarrow \tilde{x})) \lor (\sim \tilde{x} \land (\tilde{y} \leftrightarrow cst)))) \)
  \( \phi(y := cst \text{ default } x) = (\tilde{y} \leftrightarrow (\tilde{x} \lor \tilde{y}) \land \tilde{y} \Rightarrow ((\tilde{y} \land (\tilde{y} \leftrightarrow cst)) \lor (\sim \tilde{y} \land (\tilde{y} \leftrightarrow \tilde{x})\)) \)
- \( x, y \) and \( z \) are non-Boolean.
  \( \phi(y := x \text{ default } cst) = \tilde{y} \leftrightarrow (\tilde{x} \lor \tilde{y}) \)
  \( \phi(y := cst \text{ default } x) = \tilde{y} \leftrightarrow (\tilde{x} \lor \tilde{y}) \)

5.2.3 Sampling

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

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:

- \( (\hat{x} \Leftrightarrow \hat{x}_i) \land (\bar{x} \Leftrightarrow \bar{x}_i) \) if \( x \) is Boolean.
- \( (\hat{x} \Leftrightarrow \hat{x}_i) \land (\bar{x} = 0 \bar{x}_i) \) if \( x \) is non-Boolean.

### 5.3 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 ^= 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 \( P_{\text{BASIC\_TRA}} \), the introduction of signal \( XZX_24 \) and the synchronization between \( \text{CLK} \) and \( XZX_24 \) cause the incorrect specification of clocks (e.g. in the source program \( P \) and \( P_{\text{BOOL\_TRA}} \), signal \( x \) is present, but in program \( P_{\text{BASIC\_TRA}} \), it might be absent when \( XZX_24 \) is absent). This bug be caught by our validator when it found that \( \Phi(P_{\text{BOOL\_TRA}}) \not\subseteq \Phi(P_{\text{BASIC\_TRA}}) \). In addition, signal \( XZX_24 \) is introduced without declaration that makes a syntax error in \( P_{\text{BASIC\_TRA}} \).

The second problem was present in the Signal program in which a merge operator with a constant signal is used such as \( y := 1 \text{ default } x \). 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

Inria
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:

```c
// 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);
}
```
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 describe the functional and temporal constraints of the whole program and its generated C code. Then they use BDD \[7\] representations to implement the symbolic STS models, and their proof method uses a SAT-solver to reason on the signal constraints. The drawback of this approach is that it does not capture explicitly the clock semantics and in some cases, the code generator eliminates the use of 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 make some inefficiency. Moreover, the whole calculation of a synchronous program or the generated code is considered as one atomic transition in STS, thus it does not capture the data dependencies between signals 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 \[14\]. In a similar way, they formalize the abstract clocks and clock relations as first-order logic formulas with the help of interval abstraction technique. Then, to make the generated code more efficient by detecting and removing the dead-code segments (e.g., segment of code to compute data-flow which is always absent). The approach is that they determine the existence of empty clocks, mutual exclusion of two or more clocks, or clock inclusion by reasoning on the formal model using a SMT-solver. There have been some other works which adopt the translation validation approach in verification of transformations, and optimizations. In \[9, 17\], the programs before and after the transformations and optimizations of C compiler are represented in a common intermediate form, then the preservation of semantics is checked by using symbolic execution and the proof assistant Coq \[10\]. With the same purpose, in the work of \[18\], we encode the source programs and the transformations with Polynomial Dynamical Systems and prove that the transformations preserve the abstract clocks and clock relations of the source programs. By using the simulation in model checking techniques, this approach 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. With the efficiency SMT-solver 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 correctness of the multi-clocked synchronous programming language compiler for clock semantics preservation and applies this approach to the synchronous data-flow language Signal compiler. We have presented a technique based on SMT-solving to prove the preservation of timing properties during compilation. Namely, we have shown that implicit clock relations, describing the discrete timing model of a data-flow 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. All compilation phases are followed by a similar refinement verification process.

We have implemented and integrated our translation validation process within the Polychrony toolset by using the Yices solver to prove the correctness of the full compilation phases of the compiler. As future work, we would like to use the proof of abstract clock semantic preservation in this work to verify the equivalence between data-flows and the corresponding variables from the program and its generated code. The verification of equivalence will be done by using a normalizing value-graph \[24\] which contains only the computations of data-flows and there is no
timing information. We therefore evaluate this graph more efficiently.
References


Inria


Appendix A

Soundness of the Abstraction

Table 1 and 2 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:

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

**Definition** Given the abstraction \( \Phi(P) \), a property \( \varphi \) 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 [14]. 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 \( \varphi \) which is defined over the set \( X \), its concretization \( \Gamma(\varphi) \) is given by:

\[
S_{sat}(\varphi) = \bigcup_{\hat{I} \models \varphi} S_{sat}(\hat{I}) \tag{7}
\]

\[
\Gamma(\varphi) = \{T_c \in T_{cX} \mid \forall t, T_c(t) \in S_{sat}(\varphi)\} \tag{8}
\]

**Proposition 6.1** Let \( P, \Phi(P) \) be a program and its abstraction, respectively, and \( \varphi \) is a property defined over the clocks. If \( \Phi(P) \models \varphi \) then \( [[P]]_c \subseteq \Gamma(\varphi) \).

**Lemma 6.2** For all programs \( P, [[P]]_c \subseteq \Gamma(\Phi(P)) \).

**Proof** (Proposition 6.1) The proof of Proposition 6.1 is done by using Lemma 6.2. Given a clock trace \( T_c \in [[P]]_c \), applying Lemma 6.2 \( T_c \in \Gamma(\Phi(P)) \) 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 \( \varphi \). 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 \( T_c \in \Gamma(\varphi) \).

**Proof** (Lemma 6.2) 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: \( P : y := f(x_1, \ldots, x_n) \). First, consider \( y \) as numerical signal; following the encoding scheme, we have \( \Phi(P) = (\hat{y} \Leftrightarrow \hat{v}_j \Leftrightarrow \hat{x}_i \Leftrightarrow \ldots \Leftrightarrow \hat{x}_n) \). For any interpretation \( \hat{I} \) such that \( \hat{I} \models \Phi(P) \), we have:

  - either \( \forall i, \hat{y}_i = 0 \) and \( \hat{x}_i = 0 \);
  - or \( \forall i, \hat{y}_i = 1 \) and \( \hat{x}_i = 1 \).

\( S_{sat}(\Phi(P)) \) 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.

Inria
<table>
<thead>
<tr>
<th>Process $P$</th>
<th>Clock Semantics $[[P]]_c$</th>
</tr>
</thead>
</table>
| $y := f(x_1,\ldots,x_n)$ | $\{T_c \in T_{c(y,x_1,\ldots,x_n)} | \forall t \in \mathbb{N},$  
| | $(\forall i, T_c(t)(x_i) = T_c(t)(y))\}.$ |
| $y := x \texttt{\$1 init a}$ | $\{T_c \in T_{c(x,y)} | \forall t \in \mathbb{N},$  
| | $(T_c(t)(x) = T_c(t)(y))\}.$ |
| $y := x \texttt{ when } b$ | $\{T_c \in T_{c(x,y,b)} | \forall t \in \mathbb{N},$  
| | $(T_c(t)(x) = T_c(t)(b) = \text{true}$  
| | and $T_c(t)(y) = \text{true})$ or  
| | $(T_c(t)(x) = T_c(t)(b) = \text{true}$  
| | and $T_c(t)(y) = \text{false})$ or  
| | $(T_c(t)(x) = T_c(t)(b) = \text{false})$ or  
| | $(T_c(t)(y) = \text{false})\}.$ |
| $y := x \texttt{ default z}$ | $\{T_c \in T_{c(x,y,z)} | \forall t \in \mathbb{N},$  
| | $(T_c(t)(x) = T_c(t)(y) = \text{true})$ or  
| | $(T_c(t)(x) = \text{false and}$  
| | $T_c(t)(z) = T_c(t)(y))\}.$ |
| $P_1 \mid P_2$ | $X_1T_c \in [[P_1]]_c$ and $X_2T_c \in [[P_2]]_c$  
| | where $[[P_1]]_c \subseteq T_{cX_1}$ and $[[P_2]]_c \subseteq T_{cX_2}$ |

Table 1: Clock Semantics of the Basic Processes.

- Delay, sampling, and merging operators: we prove in the same manner.
- Composition: $P = P_1|P_2$. Let $T_c \in [[P]]_c$ be a clock trace, since $X_1T_c \in [[P_1]]_c$ and $X_2T_c \in [[P_2]]_c$, $[[P_1]]_c \subseteq \Gamma(\Phi(P_1))$ 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 $\forall t, T_c(t) \in S_{sat}(\Phi(P_1) \land \Phi(P_2))$, or $T_c \in \Gamma(\Phi(P))$. 

RR n° 8064
<table>
<thead>
<tr>
<th>Process $P$</th>
<th>Clock Semantics $[[P]]_c$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$y := f(x_1, ..., x_n)$</td>
<td>${ T_c \in \mathcal{T}<em>{c</em>{{y,x_1,\ldots,x_n}}} \mid \forall t \in \mathbb{N}, \langle \forall i, T_c(t)(x_i) = T_0(t)(y) \rangle } \cup { T_c \in \mathcal{T}<em>{c</em>{{x_1,\ldots,x_n}}} \mid \forall t \in \mathbb{N}, T_c(t)(y) = f(T_c(t)(x_1), \ldots, T_c(t)(x_n)) }$</td>
</tr>
<tr>
<td>$y := x$ 1 init $a$</td>
<td>${ T_c \in \mathcal{T}<em>{c</em>{{x,y}}} \mid \forall t \in \mathbb{N}, \langle T_c(t)(x) = T_c(t)(y) \rangle \cup \langle T_c(t)(x) = T_0(t)(y) \rangle }$</td>
</tr>
<tr>
<td>$y := x$ when $b$</td>
<td>${ T_c \in \mathcal{T}<em>{c</em>{{x,y,b}}} \mid \forall t \in \mathbb{N}, \langle T_c(t)(x) = T_c(t)(y) \rangle \cup \langle T_c(t)(x) = T_0(t)(y) \rangle }$</td>
</tr>
<tr>
<td>$y := x$ default $z$</td>
<td>${ T_c \in \mathcal{T}<em>{c</em>{{x,y,z}}} \mid \forall t \in \mathbb{N}, \langle T_c(t)(x) = T_0(t)(y) \rangle \cup \langle T_c(t)(x) = T_c(t)(y) \rangle }$</td>
</tr>
<tr>
<td>$P_1 \mid P_2$</td>
<td>${ T_c \in \mathcal{T}<em>{c</em>{X_1 \cup X_2}} \mid X_1, T_c \subseteq [[P_1]]_c \text{ and } X_2, T_c \subseteq [[P_2]]_c }$</td>
</tr>
</tbody>
</table>

Table 2: Clock Semantics of the Basic Processes with Boolean Signals