Translation Validation for Transformations on Abstract Clocks in Synchronous Languages

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

Formal Verification of Transformations on Abstract Clocks in Synchronous Compilers

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

RESEARCH REPORT
N° 8064
September 2012
Project-Team ESPRESSO

ISSN 0249-6399 ISRN INRIA/RR--8064--FR+ENG

Formal Verification of Transformations on Abstract Clocks in Synchronous Compilers

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

Project-Team ESPRESSO

Research Report n° 8064 — September 2012 — 23 pages

Abstract: Translation validation was introduced in the 90's by Pnueli et al. as a technique to formally verify correctness of code generated from the synchronous data-flow language Signal. Rather than certifying the code generator (by writing it entirely using a theorem prover) or exhaustively qualifying it (by obeying the 27 required documents of DO-178C), translation validation provides a scalable approach to assess the functional correctness of generated code. By revisiting the translation validation approach, which in the 90's suffered from the limitations of theorem proving and model checking techniques, we aim at developing a scalable and flexible approach that can be applied to an existing 500k-lines implementation of the Signal compiler, and handle large-scale, possibly automatically generated Signal programs using efficient SAT/SMT-solving libraries.
We implement translation validation in step-by-step style, by proving each transformation of the compiler from the initial step, until the latest step of actual C-code generation. In this work, we focus on proving the preservation of timing properties during the compilation process. We define a *correct transformation* relation between two formal representations of source and transformed programs, called *clock models*. Then we use an SMT-solver (Satisfiability Modulo Theory) for checking the existence of this relation.

Key-words: Formal Verification, Translation Validation, Certified Compiler, SMT Solver, Multi-clocked Synchronous Programs, Embedded Systems

**RESEARCH CENTRE
RENNES – BRETAGNE ATLANTIQUE**

Campus universitaire de Beaulieu
35042 Rennes Cedex

Vérification Formelle des Transformations sur Les Horloges dans Compilateurs Synchrones des Données de Flux

Résumé : Translation validation a été introduit dans les années 90 par Pnueli et al. Comme une technique pour vérifier formellement exactitude de code généré à partir de la langue synchrone de donnée de flux Signal. Plutôt que de certifier le générateur de code (en l'écrivant entièrement à l'aide de la démonstration de théorèmes) ou de façon exhaustive le qualifiant (en obéissant aux 27 documentations requises selon la norme DO-178C), la translation validation fournit une approche évolutive pour évaluer l'exactitude fonctionnelle du code généré. En re-visitant la translation validation, qui dans les années 90 a souffert des limitations de la démonstration de théorèmes et de model checking techniques disponibles alors, nous visons à développer une approche évolutive et flexible qui peut s'appliquer à un existant 500k lignes de la mise en oeuvre de Signal compilateur, et de traiter à grande échelle, peut-être générés automatiquement Signal programmes efficaces en utilisant les bibliothèques de SAT/SMT-solving.

Nous mettons en oeuvre la translation validation dans l'étape-par-étape de la mode, en prouvant chaque transformation de l'étape initiale, jusqu'à ce que la dernière étape de réelle C-génération de code. Dans ce travail, nous nous concentrons sur la préservation de prouver les propriétés de minutage lors de la compilation. Nous définissons une *transformation correcte* relation entre deux représentations formelles de source et des programmes transformées, appelées *modèles d'horloge*. Ensuite, nous utilisons un SMT-solver(Satisfiability Modulo Theory) permettant de vérifier cette relation.

Mots-clés : Vérification formelle, Validation traduction, Compilateur certifié, SMT solver, Programmes synchrones multi-horloges, Systèmes embarqués

1 Introduction

Adhering to the synchronous paradigm, synchronous data-flow languages such as Esterel, Lustre, Signal [2, 14, 12] have been introduced and successfully used to design and implement embedded and critical real-time systems. Each synchronous data-flow language is generally associated with a compiler which transforms, compiles synchronous programs and usually generates code in some general-purpose programming language. Safety-critical, high-assurance systems specified using such languages are verified by the use of formal methods (e.g. model checking, program proof, and static analysis). The verification is usually applied to the source code of a synchronous program. The designer always expects that all the formally verified properties of the considered system can be carried out to the transformed program and the automatically generated code. However, before code can be generated, the compilation of high-level, synchronous, specification is a complex process that involves many analysis and program transformation stages. Some transformations may introduce additional information or constraints, to refine the meaning of the original specification and/or remove, specialize the behavior of the source specification, such as optimization, static scheduling. Thus, and even if compliant with a "five-nines" (99.999%) reliability, large-scale use of compilers for large specifications may improbably yet not uncertainly yield bugs. Therefore, it is naturally required that the compiler must be formally checked as well to ensure that the source program semantics is preserved.

Means to circumvent compiler bugs are to entirely rewrite the code generator (in our case, e.g., the 500k C-code lines of the Signal compiler) using a theorem proving tool such as Coq [9], or qualify its compliance to DO-178C documents for a particular execution platform, or to formally verify the conformance of its output to its input for each run of the code generator. The first two solutions yield a situation where the code generator can either hardly or impossibly be further optimized and updated, whereas the last one provides ideal separation between the tool under verification and its checker.

In this aim, *translation validation* was introduced in the 90's by Pnueli et al. [22, 23] as a technique to formally verify the correctness of code generated from the language Signal using model checking. Rather than certifying the code generator (by writing it entirely using a theorem prover) or exhaustively qualifying it (by obeying the 27 required documents of DO-178C), translation validation provides a scalable approach to assess the functional correctness of generated code. First, we do not modify or instrument the compiler, and we treat the compiler as a "black box", hence our validator is not suffered from the update or modification of the compiler. Our approach is to apply formal methods to the compiler transformations themselves in order to automatically generate formal evidence that the semantics of the source program is preserved during program transformation and compilation, as per applicable qualification standard (DO-178C). Second, it is important that the validator can be scaled to large programs, in which we represent the desired program semantics with our scalable abstraction and use efficient SMT libraries [5, 18] to achieve the expected goals: traceability and formal evidence. In this paper, we focus on proving the preservation of timing properties during the compilation process. We define a *correct transformation* relation between two formal representations of source and transformed programs, called *clock models*. Then we use an SMT-solver (Satisfiability Modulo Theory) for checking the existence of this relation. A clock model is represented as a first-order logic formula over boolean variables. This boolean formula deterministically characterizes the presence/absence status of all discrete data-flows (input, output and local variables of the program) manipulated by the specification. Based on the features of the compilation, we apply our translation validation to the abstract clock calculation phase and implement it in step-by-step style, by proving each transformation in this phase to provide the explicit proof of the preservation of timing properties. Another significant contribution of our work is that this preservation of timing properties will be used to verify the equivalence between data-flows and the corresponding variables from the program and its generated code. The verification of equivalences will be done by using a *normalizing* value-graph [25], however, because of the preservation of timing properties the resulting value-graph contains only the computations of data-flows and there is no timing information. Therefore we can evaluate it more efficiently and increase in speed.

The remainder of this paper is organized as follows. Section 3 introduces our translation validation approach by example. Section 4 presents the translation of a synchronous program into its clock model. In Section 5, we consider the definition of *correct transformation* on abstract clocks which formally proves conformance between the original specification and that reverse-engineered from its compiled program. It also addresses the application of the verification process to the Signal compiler, and its implementation integrated in the Polychrony toolset [21]. Section 6 presents some related works, concludes our work and outlines future directions.

2 Preliminaries

In this section, we will recall some basic elements of propositional and first-order logics, their semantics and validity, satisfiability checking problems [11, 15].

2.1 Propositional logic

The expressions in propositional logic are called propositional *formulas* which can be any string whose evaluation is either true or false. Propositional formulas are expressed in a propositional language which consists a countable set P whose elements are called *boolean variables* and denoted by p, q, r.

Definition (formula.) Propositional formulas are defined inductively as:

- Every boolean variable is a formula, called *atom*,
- $\top$ and $\bot$ are formulas,
- If A_1, A_2 are formulas, then $A_1 \bowtie A_2$ is a formula.

where $\bowtie \in \{\wedge, \vee, \neg, \rightarrow, \leftrightarrow\}$ used to build formulas are called *connectives.* A formulas of forms $A_1 \vee A_2, A_1 \wedge A_2$ are respectively a *disjunction* and *conjunction.*

The semantics of propositional logic is based on the following assumptions: *(i)* the meaning of atomic propositions depends on their interpretation, *(ii)* the meaning of more complex propositions depends on the meaning of their components as it is shown in Table 1.

Definition (boolean value, interpretation, truth.) A boolean value (or *truth* value) is either 1 or 0. An *interpretation* (or truth assignments) for a set of boolean variables P is a mapping from P to the set of boolean values $\{1, 0\}$.

Interpretations can be extended to arbitrary propositional formulas inductively as following:

- $I(\top) = 1$ and $I(\bot) = 0$.
- $I(A_1 \wedge ... \wedge A_n) = 1$ iff $I(A_i) = 1$ for all i.
- $I(A_1 \vee ... \vee A_n) = 1$ iff $I(A_i) = 1$ for some i.
- $I(\neg A) = 1$ iff $I(A) = 0$.
- $I(A \rightarrow B) = 1$ iff $I(A) = 0$ or $I(B) = 1$.
- $I(A \leftrightarrow B) = 1$ iff $I(A) = I(B)$.

Given an interpretation I, we say that formula A is true (respectively false) in I if $I(A) = 1$ (respectively $I(A) = 0$), denoted by $I \models A(I \not\models A)$.

The following definition defines the satisfiability, validity, and equivalence of formulas.

Definition (model, satisfiability, validity, equivalence) An interpretation I *satisfies* a formula A if A is true in I, and I is called a *model* of A. A formula A is *satisfiable* (*valid*) if it is true in some (every) interpretation. A valid formula is called *tautology.* Two formulas A and B are *equivalent* ($A \equiv B$) if every model of A is a model of B, and vice versa.

The above definition can be generalized to sets of formulas as follows. We say that an interpretation I is a model of a set of formulas S if it satisfies every formula in S, denoted as $I \models S$. A set of formulas is satisfiable if there exists some model. We consider here some main lemmas which are useful for the problems of checking equivalence, validity, and satisfiability.

$\wedge$	1	0
1	1	0
0	0	0

$\vee$	1	0
1	1	1
0	1	0

$\neg$	
1	0
0	1

$\rightarrow$	1	0
1	1	0
0	1	1

$\leftrightarrow$	1	0
1	1	0
0	0	1

Table 1: Operators semantics

Lemma 2.1 *(i) A formula A is valid iff $\neg A$ is unsatisfiable. (ii) A formula A is satisfiable iff $\neg A$ is not valid. (iii) A formula A is valid iff A is equivalent to $\top$. (iv) Formula A and B are equivalent iff the formula $A \leftrightarrow B$ is valid.*

Proof The proofs of properties (i), (ii), (iii), (iv) use the same method. Thus, we will only prove property (iv).
$\Rightarrow$) Assume that $A \leftrightarrow B$ is valid, given any interpretation I, we have $I \models (A \leftrightarrow B)$, following the truth table of $\leftrightarrow$ we can see that $I \models A$ iff $I \models B$, so A and B are equivalent.
$\Leftarrow$) Assume that A and B are equivalent, given any interpretation I. If $I \models A$, then by the equivalence $I \models B$, thus $I \models A \leftrightarrow B$. In the similar way, if $I \not\models A$, then by the equivalence $I \not\models B$, and hence $I \models A \leftrightarrow B$. Therefore, $A \leftrightarrow B$ is valid.

The following lemma can help us reduces satisfiability checking for set of formulas to satisfiability checking for a formula.

Lemma 2.2 *Let $S = \{A_1, ..., A_n\}$ be a finite set of formulas. Then S is satisfiable iff the formula $A_1 \wedge ... \wedge A_n$ is satisfiable.*

Proof We prove in the same manner as the proof of Lemma 2.1 by using the interpretation property of $A_1 \wedge ... \wedge A_n$ and the truth table of $\wedge$. Hence, we omit the detailed proof here.

Evaluating a formula in an interpretation can be formalized as the following decision problem:

Definition (formula evaluation.) Formula evaluation a decision problem whose instance is a pair (A, I), where A is formula and I is an interpretation. The answer is "yes" if $I \models A$.

We can evaluate formulas in interpretations by using straightforward the above definition. In other hand, we can first evaluate its sub-formulas, then using the truth tables to evaluate the formula.

Example We consider the formula $A = (p \rightarrow q) \wedge (p \wedge q \rightarrow r) \rightarrow (p \rightarrow r)$ and the interpretation $I = \{p \mapsto 1, q \mapsto 0, r \mapsto 1\}$. The decision problem using the evaluations of sub-formulas is described as Table 2. Verification engines that can reason on propositional formulas to answer whether $I \models A$ are called SAT solvers.

2.2 First-order logic

As with propositional logic, expressions in first-order logic are made up of sequences of symbols which divided into two categories: logical symbols and non-logical symbols or parameters. Logical symbols consists of *parentheses* $((,))$, *propositional connectives* $(\neg, \vee, \wedge, \rightarrow, \leftrightarrow)$, *variable*, and *quantifiers* $(\forall, \exists)$. Parameters consists of *equality symbol* $(=)$, *predicate symbols* (e.g. $x > y$), *constant symbols* (e.g. $0, \pi$), *function symbols* (e.g. $x + y * z$). Each predicate and function symbol has an associated *arity* which is a natural number indicating how many arguments it takes. Equality and constant symbols can be considered as a special predicate symbol of arity 2,

	subformula	value
1	$(p \rightarrow q) \wedge (p \wedge q \rightarrow r) \rightarrow (p \rightarrow r)$	1
2	$(p \rightarrow r)$	1
3	$(p \rightarrow q) \wedge (p \wedge q \rightarrow r)$	0
4	$(p \wedge q \rightarrow r)$	1
5	$(p \rightarrow q)$	0
6	$p \wedge q$	0
7	p	1
8	q	0
9	r	1

Table 2: Evaluation of formula using truth tables

and a function of arity 0, respectively. We denote $\mathcal{P}$ is the set of predicate symbols, $\mathcal{F}$ is the set of function symbols, $\mathcal{C}$ is the set of constant symbols. And $\Sigma = \mathcal{P} \cup \mathcal{F} \cup \mathcal{C}$ is called a *signature*.

As propositional logic, the expressions in first-order logic are called *formulas* which can be any sequences of symbols (logical and parameter symbols) whose evaluation is either true or false. First-order formulas are expressed in a first-order language which must first specify its parameters. First, we consider the *terms* of a first-order language which made up of logical and parameter symbols as following:

Definition (terms.) Terms are defined as follows.

- Any variable is a term.
- If $c \in \mathcal{C}$, then c is a term
- If $t_1, ..., t_n$ are terms and $f \in \mathcal{F}$ with arity $n > 0$, then $f(t_1, ..., t_n)$ is a term.
- Nothing else is a term.

Or we can write in Backus Naur form: $t ::= x \mid c \mid f(t, ..., t)$ where x is a variable. Given the definition of terms, we can now define the formulas in first-order language.

Definition Given the set of terms, first-order formulas are defined inductively as follows:

- If $P \in \mathcal{P}$ whose arity $n \geq 1$, and $t_1, ..., t_n$ are terms, then $P(t_1, ..., t_n)$ is a formula (*atomic formula*).
- If ϕ is a formula, then $\neg\phi$ is a formula.
- If ϕ and ψ are formulas, then so are $(\phi \vee \psi), (\phi \wedge \psi), (\phi \rightarrow \psi)$ and $(\phi \leftrightarrow \psi)$.
- If ϕ is a formula and x is a variable, then $(\forall x.\phi)$ and $(\exists x.\phi)$ are formulas.
- Nothing else is a formula.

The above definition can be represented in the Backus Naur form as:

$$\phi ::= P(t_1, ..., t_n) \mid (\neg\phi) \mid (\phi \vee \phi) \mid (\phi \wedge \phi) \mid (\phi \rightarrow \phi) \mid (\phi \leftrightarrow \phi) \mid (\forall x.\phi) \mid (\exists x.\phi)$$

The set of *well-formed formulas* is the set of formulas generated inductively from the atomic formulas by using the operations $\mathcal{E}_\neg, \mathcal{E}_\rightarrow$, and $\mathcal{E}_\forall$, where $\mathcal{E}_\neg(\phi) = (\neg\phi), \mathcal{E}_\rightarrow(\phi, \psi) = (\phi \rightarrow \psi)$ and $\mathcal{E}_\forall(\phi) = \forall v_i.\phi, i = 1, 2, ...$ Given a well-formed formula ϕ, a variable x is said free in ϕ:

- If ϕ is an atomic formula, then x is free iff x occurs in ϕ.
- x is free in $(\neg\phi)$ iff x is free in ϕ.
- x is free in $\phi \rightarrow \psi$ iff x is free in ϕ or ψ.
- x is free in $\forall v_i.\phi$ iff x is free in ϕ and $x \neq v_i$.

If $\forall v_i$ appears in ϕ, then v_i is said to be bound in ϕ. A formula without free variable is called a *sentence*.

In first-order logic, we use a *model* (also called a *structure*) to determine the truth of a formula. Given a signature Σ, a model $\mathcal{M}$ of the pair $(\mathcal{F}, \mathcal{P})$ consists of the following set of data:

- A non-empty set A, the universe of concrete values,
- For each constant $c \in \Sigma$, a concrete element $c^{\mathcal{M}}$ of A,
- For each function $f \in \mathcal{F}$ with arity $n > 0$, a concrete function $f^{\mathcal{M}} : A^n \rightarrow A$, and
- For each $P \in \mathcal{P}$ with arity $n > 0$, a subset of $P^{\mathcal{M}} \subseteq A^n$.

Here, it is totally different between f and $f^{\mathcal{M}}$ and between P and $P^{\mathcal{M}}$. The symbols f and P are just that symbols, whereas $f^{\mathcal{M}}$ and $P^{\mathcal{M}}$ denote a concrete function and relation in a model $\mathcal{M}$, respectively.

Example Let $\mathcal{F} = \{i\}$ and $\mathcal{P} = \{T, F\}$ [15] where i is a constant, F and T are predicate symbols with arities one and two, respectively. A model $\mathcal{M}$ a set of concrete values A which may be considered as a states of an automata. The interpretations $i^{\mathcal{M}}$, $T^{\mathcal{M}}$, and $F^{\mathcal{M}}$ would be an initial state, a translation relation, and a set of final states, respectively. For instance, let $A = \{a, b, c\}, i^{\mathcal{M}} = a, T^{\mathcal{M}} = \{(a, a), (a, b), (a, c), (b, c), (c, c)\}$, and $F^{\mathcal{M}} = \{b, c\}$ where (a, b) means that there exists a transition from state a to state b. This model can be used to check a formula of first-order logic $\exists y.T(i, y)$. This formula says that there is a transition from the initial state to some state, and it is true in our model since there exists transitions from the initial state a to states a, b, and c.

It remains the value assignments of variables in our model. Given a model $\mathcal{M}$, a *variable assignment* is a mapping which assigns to each variable x a value of $\mathcal{M}$. Finally, we are able to give a semantics to first-order logic formulas as follows:

Definition Given a model $\mathcal{M}$ for a signature Σ and a variable assignment l, we define the satisfaction relation, denoted by $\mathcal{M} \models_l \phi$ for each formula ϕ over the signature Σ and the variable assignment l by using structural induction on ϕ. If $\mathcal{M} \models_l \phi$ holds, we say that ϕ computes to true in the model $\mathcal{M}$ with respect to the environment l.

The structural induction on formula ϕ is described as the following:

- P : If ϕ of the form $P(t_1, ..., t_n)$, then we interpret the terms $t_1, ..., t_n$ in the set A by replacing all variables with their values according to l. Assume that concrete values $a_1, ..., a_n$ of A for each of these terms, where any function symbol f is interpreted by $f^{\mathcal{M}}$. Then $\mathcal{M} \models_l P(t_1, ..., t_n$ holds iff $(a_1, ..., a_n) \in P^{\mathcal{M}}$.
- $\forall x : \mathcal{M} \models_l \forall x\phi$ holds iff $\mathcal{M} \models_{l[x \mapsto a]} \phi$ holds for all $a \in A$.
- $\exists x : \mathcal{M} \models_l \exists x\phi$ holds iff $\mathcal{M} \models_{l[x \mapsto a]} \phi$ holds for some $a \in A$.

- $\neg$: $\mathcal{M} \models_l \neg\phi$ holds iff $\mathcal{M} \models_l \phi$ does not hold.
- $\vee$: $\mathcal{M} \models_l \phi_1 \vee \phi_2$ holds iff $\mathcal{M} \models_l \phi_1$ or $\mathcal{M} \models_l \phi_2$ holds.
- $\wedge$: $\mathcal{M} \models_l \phi_1 \wedge \phi_2$ holds iff $\mathcal{M} \models_l \phi_1$ and $\mathcal{M} \models_l \phi_2$ hold.
- $\rightarrow$: $\mathcal{M} \models_l \phi_1 \rightarrow \phi_2$ holds iff $\mathcal{M} \models_l \phi_2$ holds whenever $\mathcal{M} \models_l \phi_1$ holds.

We use $\mathcal{M} \not\models \phi$ to denote the fact that $\mathcal{M} \models \phi$ does not hold. Given a model $\mathcal{M}$ for a signature Σ and a variable assignment l, verification engines that can reason on formulas of first-order logic to answer whether $\mathcal{M} \models \phi$ are called *Satisfiability Modulo Theories* (SMT) solvers. A primary goal of a SMT solver is to create a verification engine that can reason natively at a higher level of abstraction, while still retaining the speed and automation of boolean engines.

3 Verification by Example

3.1 Overview of Signal

In Signal language [17, 12], the reactions of a reactive system and its environment's events along time are represented by flows of data, called *signals*. A signal is a sequence of values with the same type along an infinite sequence of instants. The set of instants (or time tags) where a signal is present is the *abstract clock* of the signal. The constructs of the language use an equational style to specify the relations and dependencies of data and clocks between signals. Systems of equations on signals are built using a composition which construct a *process*. A whole program is a process which runs infinitely taking parameters, input signals for computing the output signals to react to the environment.

The language is based on seven different types of equations to construct primitive processes or equations specifying computations over signals. And a composition operation is used to build more elaborate processes in the form of systems of equations. We will present each equation along with its semantic meaning and the implicit relationships between the clocks of the input and output signals.

- *Equation on Data:* The equation $y := f(x_1, ..., x_n)$ where f is a *n-ary* relation over numerical or boolean data types, defines a process whose output $y(t)$ for instant $t \in \hat{y}$ is $y(t) = f(x_1(t), ..., x_n(t))$. The clock constraint of the input and output signals is $\hat{y} = \hat{x}_1 = ... = \hat{x}_n$.
- *Delay:* The equation $y := x\$1 \text{ init } a$ defines a process whose output $y(t_i) = a$ if t_i is the initial instant, and for every other instant, $y(t_i) = x(t_{i-1})$. The clock constraint of the input and output signals is $\hat{y} = \hat{x}$.
- *Merge:* The merge equation $y := x \text{ default } z$ defines a process whose output at instant t is $y(t) = x(t)$ when $t \in \hat{x}$ and $y(t) = z(t)$ if $t \notin \hat{x} \wedge t \in \hat{y}$. The clock constraint of the merge equation is $\hat{y} = \hat{x} \cup \hat{z}$.
- *Sampling:* The sampling equation $y := x \text{ when } b$ defines a process whose output signal $y(t)$ has value $x(t)$ when the signal x is present and the boolean signal b is present with the value *true*. The clock constraint of input and output signals is $\hat{y} = \hat{x} \cap [b]$ where $[b] = \{t \in \hat{b} | b(t) = true\}$.
- *Composition:* $P \triangleq P_1 \mid P_2$ where P_1 and P_2 are processes. P consists of the composition of the systems of equations. The composition operator is commutative and associative.
- *Restriction:* $P \triangleq P_1$ where x, where P_1 and x are a process and a signal, respectively. It enables local declarations in the process P_1, and leads to the same constraints as P_1.
- *Equation on clocks:* The language allows clock constraints to be defined *explicitly* by equations. The signal's clock is represented by a special signal of type *event* which carries only a single value *true*. Thus, equations on clocks over signals are equations over their corresponding event signals. They are: *(i)* the synchronization relation $x \,\hat{=}\, y \triangleq \hat{x} = \hat{y}$, *(ii)* clock union relationship $x \,\hat{+}\, y \triangleq \hat{x} \text{ default } \hat{y}$, *(iii)* clock intersection relationship $x \,\hat{*}\, y \triangleq \hat{x} \text{ when } \hat{y}$, *(iv)* difference relationship $x \,\hat{-}\, y \triangleq \text{when}(\text{not}(\hat{y}) \text{ default } \hat{x})$.

3.2 Illustrative Example

We begin by showing how our verification process works for an illustrative example. Consider the following synchronous program `DEC` written in Signal language:

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

Figure 1: DEC in Signal

In program DEC, there are an input signal FB, an output signal N, and a local signal ZN, all declared as integer signals. When it receives a new positive value FB, the program will compute the output N as the sequence of values FB, FB-1,...,2,1. When the output value is 1, it will accept the next input. The output N is equal to FB if its previous value (referring to ZN with the delay operator $) is less than or equal to 1. Otherwise, it is decremented by 1. The input FB is accepted (or it is present) only when ZN becomes less than or equal to 1. This is defined by the equation `FB ^= when (ZN<=1)` which defines the clock of FB. We will use the symbol $\perp$ to denote the fact that a signal holds no value, or it is absent. And at a particular instant, if a signal is present, then we use a value 1 to represent the value of its clock, otherwise we use a value 0. Then a possible computation and the corresponding clocks of variables of this program are:

$$\begin{pmatrix} FB: & \perp \\ N: & \perp \\ ZN: & 1 \end{pmatrix} \begin{pmatrix} FB: & 2 \\ N: & 2 \\ ZN: & 1 \end{pmatrix} \begin{pmatrix} FB: & \perp \\ N: & 1 \\ ZN: & 2 \end{pmatrix} \begin{pmatrix} FB: & 3 \\ N: & 3 \\ ZN: & 1 \end{pmatrix} ...$$

$$\begin{pmatrix} clk(FB): & 0 \\ clk(N): & 0 \\ clk(ZN): & 1 \end{pmatrix} \begin{pmatrix} clk(FB): & 1 \\ clk(N): & 1 \\ clk(ZN): & 1 \end{pmatrix} \begin{pmatrix} clk(FB): & 0 \\ clk(N): & 1 \\ clk(ZN): & 1 \end{pmatrix} \begin{pmatrix} clk(FB): & 1 \\ clk(N): & 1 \\ clk(ZN): & 1 \end{pmatrix} ...$$

The output program `DEC_BASIC_TRA.SIG` obtained by compiling program `DEC` in the first phase of the Signal compiler (in which the clocks are made explicit) is given by:

```
...
CLK_N := CLK_N ^+ CLK_FB
| CLK_N ^= N ^= ZN
| CLK_FB := when (ZN<=1)
| CLK_FB ^= FB
...
```

The clocks of the variables in the program for the same computation as above are given below. We skip the clocks of local intermediate variables.

$$\begin{pmatrix} clk(FB): & 0 \\ clk(N): & 0 \\ clk(ZN): & 1 \end{pmatrix} \begin{pmatrix} clk(FB): & 1 \\ clk(N): & 1 \\ clk(ZN): & 1 \end{pmatrix} \begin{pmatrix} clk(FB): & 0 \\ clk(N): & 1 \\ clk(ZN): & 1 \end{pmatrix} \begin{pmatrix} clk(FB): & 1 \\ clk(N): & 1 \\ clk(ZN): & 1 \end{pmatrix} ...$$

We can have an observation that the transformed program is a correct transformation of the source program if for all possible clocks of the variables in the transformed program, they are also the clocks of the variables in the source program.

In the next sections, we will show how we formalize this process and propose a method to automate it. First, we compute the formal models, called *clock models*, which represent the clock information of the source program and its compiled program. Once we have the clock models, we check that all possible clocks in the clock model of the transformed program are also the clocks in the clock model of the source program.

4 Signal to Clock Model

In synchronous languages, the logical time is completely determined by the system reactions on the occurrences of observed events. The system is supposed to react fast enough to produce the corresponding output events on the occurrence of input event before the next input event arrives. Each reaction denotes a single *logical instant* in the synchronous model, where the relations between observed events and the data dependencies are expressed [1]. Synchronous data-flow languages represent data as infinite sequences of values called *data-flows*, and each data-flow is combined with an associated abstract clock to define the presence or absence of the data in its data-flow. Thus, the principle of our encoding scheme is that, at a particular instant, the abstract clock can be represented as a variable whose values are `true` (the corresponding data-flow is present) or `false` (the corresponding data-flow is absent).

Consider a program P, we denote by $X_P = \{x_1, x_2, ..., x_n\}$ the set of all data-flow variables. For each data-flow x_i of type numerical, boolean, or event, we encode its clock with a boolean variable $\hat{x}_i$. In consequence of the equational structure of program, we represent the relations between abstract clocks described implicitly or explicitly in terms of first-order logic formulas over boolean variables. And the combination of equations can be represented by the conjunction of the corresponding formulas. We assume that all considered programs are supposed to be written with the primitive operators, meaning that derived operators are replaced by their corresponding primitive ones. And there is no nested operators such as $z := x$ `default` $(y$ `when` $b)$ by using fresh variables to break nested operators. These formulas use the usual logic operators and numerical comparison functions [15]. For the boolean expressions defined by numerical comparison functions and numerical expressions, to ensure the resulting formulas are boolean, we only encode the fact that the clocks of boolean and numerical expressions are synchronized, and we avoid encoding the values of the expressions. For each equation eq_i in program P, we denote by Φ_{eq_i} its abstract clock semantics, then the abstract clock semantics of P can be represented by a first-order logic formula, called its *clock model*, denoted as:

$$\Phi_P = \bigwedge_{i}^{n} \Phi_{eq_i} \tag{1}$$

where n denotes the number of equations composed in P.

We use the method above to compute the clock model of a Signal program. It means that for each signal x, we use a boolean variable $\hat{x}$ to encode its abstract clock. We only need to define the translation of the primitive equations to formulas encoding the abstract clocks, and the implicit or explicit clock relations of the signals involved in the equations. The composition of equations is simply translated as the conjunction of the corresponding first-order logic formulas. For the delay operator \$ (e.g. $x\$1$), it requires memorizing the past value of the signal, that is done by introducing a new variable $m.x$, where $m.x$ stores the previous value of signal x and $m.x'$ stores the current value of signal x. Table 3 shows the translation of the primitive equations of the language, where $\leftrightarrow$ denotes the equivalent relation. For instance, the primitive equation $y := x_1$ `and` x_2 is represented by this first-order logic formula: $\hat{y} \leftrightarrow \widehat{x_1} \leftrightarrow \widehat{x_2} \wedge y \leftrightarrow x_1 \wedge x_2$. Signal allows clock constraints to be defined explicitly by equations; in this context, the signal clock is represented by a special signal of type event and our abstraction encodes the clock by using a boolean variable. By applying the above translation scheme, the following translations are obtained for equations on clocks:

- $x \hat{=}\ y \mapsto \hat{x} \leftrightarrow \hat{y}$ (synchronization)
- $z := x \hat{+}\ y \mapsto \hat{z} \leftrightarrow (\hat{x} \vee \hat{y})$ (union)

Boolean signals		Non-boolean signals	
$y :=$ not x	$\hat{y} \leftrightarrow \hat{x}$ $\wedge\ y \leftrightarrow \neg x$	$y := f(x_1, ..., x_n)$	$\hat{y} \leftrightarrow \widehat{x_1} \leftrightarrow ... \leftrightarrow \widehat{x_n}$
$y := x$ and z	$\hat{y} \leftrightarrow \hat{x} \leftrightarrow \hat{z}$ $\wedge\ y \leftrightarrow x \wedge z$		
$y := x$ or z	$\hat{y} \leftrightarrow \hat{x} \leftrightarrow \hat{z}$ $\wedge\ y \leftrightarrow x \vee z$		
$y := x$ default z	$\hat{y} \leftrightarrow \hat{x} \vee \hat{z}$ $\wedge\ y \leftrightarrow (\hat{x} \wedge x \vee \neg\hat{x} \wedge \hat{z} \wedge z)$	$y := x$ default z	$\hat{y} \leftrightarrow \hat{x} \vee \hat{z}$
$y := x$ when b	$\hat{y} \leftrightarrow (\hat{x} \wedge \hat{b} \wedge b)$ $\wedge\ y \leftrightarrow (\hat{x} \wedge x \wedge \hat{b} \wedge b)$	$y := x$ when b	$\hat{y} \leftrightarrow (\hat{x} \wedge \hat{b} \wedge b)$
$y := x\$1$ init a	$\hat{y} \leftrightarrow \hat{x}$ $\wedge\ y \leftrightarrow (\hat{x} \wedge m.x)$ $\wedge\ m.x_0 \leftrightarrow a$ $\wedge\ m.x' \leftrightarrow (\hat{x} \wedge x \vee \neg\hat{x} \wedge m.x)$	$y := x\$1$ init a	$\hat{y} \leftrightarrow \hat{x}$
$P_1 \mid P_2$	$\Phi_{P_1} \wedge \Phi_{P_2}$		
P where x	$\exists x.\ \Phi_P$		

Table 3: Translation of the primitive equations

- $z := x\hat{}*\ y \mapsto \hat{z} \leftrightarrow (\hat{x} \wedge \hat{y})$ (intersection)
- $z := x\hat{}-\ y \mapsto \hat{z} \leftrightarrow (\hat{x} \wedge \neg\hat{y})$ (difference)

For example, for the Signal program `DEC` shown in Figure 1, following the encoding scheme above, we can obtain the clock model Φ_{DEC} of `DEC` as:

$$\begin{array}{rcl} \mathtt{N} := \mathtt{FB}\ \text{default}\ (\mathtt{ZN} - \mathtt{1}) & \mapsto & \widehat{\mathtt{N}} \leftrightarrow \widehat{\mathtt{FB}} \vee \widehat{\mathtt{ZN}} \\ \mathtt{ZN} := \mathtt{N}\$1\ \text{init}\ \mathtt{1} & \mapsto & \widehat{\mathtt{ZN}} \leftrightarrow \widehat{\mathtt{N}} \\ \mathtt{FB}\ \hat{}=\ \text{when}\ (\mathtt{ZN} <= \mathtt{1}) & \mapsto & \widehat{\mathtt{FB}} \leftrightarrow \widehat{\mathtt{ZN_1}} \wedge \mathtt{ZN_1} \\ \mathtt{ZN_1} := \mathtt{ZN} <= \mathtt{1} & \mapsto & \widehat{\mathtt{ZN}} \leftrightarrow \widehat{\mathtt{ZN_1}} \end{array}$$

$$\Phi_{DEC} = \widehat{\mathtt{N}} \leftrightarrow \widehat{\mathtt{FB}} \vee \widehat{\mathtt{ZN}} \wedge \widehat{\mathtt{ZN}} \leftrightarrow \widehat{\mathtt{N}} \wedge \widehat{\mathtt{FB}} \leftrightarrow \widehat{\mathtt{ZN_1}} \wedge \mathtt{ZN_1} \wedge \widehat{\mathtt{ZN}} \leftrightarrow \widehat{\mathtt{ZN_1}}$$

Process P	Semantics $[[P]]_c$
$y := R(x_1, ..., x_n)$	$\{T_c \in \mathcal{T}c_{\{y,x_1,...,x_n\}} \mid \forall t \in \mathbb{N}, (\forall i, T_c(t)(x_i) = T_c(t)(y))\}$
$y := x$ default z	$\{T_c \in \mathcal{T}c_{\{x,y,z\}} \mid \forall t \in \mathbb{N}, (T_c(t)(y) = T_c(t)(x) = 1) \vee$ $(T_c(x) = 0 \wedge T_c(t)(y) = T_c(t)(z) = 1) \vee$ $(T_c(t)(y) = T_c(t)(x) = T_c(t)(z) = 0)\}$
$y := x$ when b	$\{T_c \in \mathcal{T}c_{\{x,y,b\}} \mid \forall t \in \mathbb{N}, (T_c(t)(x) = 1 \wedge T(t)(b) = 1 \wedge T_c(t)(y) = 1) \vee$ $(T_c(x) = 0 \wedge T_c(t)(y) = 0) \vee$ $(T_c(t)(x) = 1 \wedge T_c(t)(b) = 0 \wedge T_c(t)(y) = 0)\}$
$y := x\$1$ init a	$\{T_c \in \mathcal{T}c_{\{x,y\}} \mid \forall t \in \mathbb{N}, (T_c(t)(x) = T_c(t)(y))\}$
$x \hat{=} y$	$\{T_c \in \mathcal{T}c_{\{x,y\}} \mid \forall t \in \mathbb{N}, (T_c(t)(x) = T_c(t)(y))\}$
$z := x \hat{+} y$	$\{T_c \in \mathcal{T}c_{\{x,y,z\}} \mid \forall t \in \mathbb{N}, (T_c(t)(x) = 1 \wedge T_c(t)(z) = 1) \vee$ $(T_c(t)(x) = 0 \wedge T_c(t)(y) = 1 \wedge T_c(t)(z) = 1) \vee$ $(T_c(t)(x) = 0 \wedge T_c(t)(y) = 0 \wedge T_c(t)(z) = 0)\}$
$z := x \hat{*} y$	$\{T_c \in \mathcal{T}c_{\{x,y,z\}} \mid \forall t \in \mathbb{N}, (T_c(t)(x) = 1 \wedge T_c(t)(y) = 1 \wedge T_c(t)(z) = 1) \vee$ $(T_c(t)(x) = 0 \wedge T_c(t)(z) = 0) \vee$ $(T_c(t)(y) = 0 \wedge T_c(t)(z) = 0)\}$
$z := x \hat{-} y$	$\{T_c \in \mathcal{T}c_{\{x,y,z\}} \mid \forall t \in \mathbb{N}, (T_c(t)(x) = 1 \wedge T_c(t)(y) = 0 \wedge T_c(t)(z) = 1) \vee$ $(T_c(t)(x) = 0 \wedge T_c(t)(z) = 0) \vee$ $(T_c(t)(x) = 1 \wedge T_c(t)(y) = 1 \wedge T_c(t)(z) = 0)\}$

Table 4: Clock semantics of the primitive equations

5 Translation Validation for Clock Transformations

5.1 Soundness of The Clock Model

Let $X = \{x_1, ..., x_n\}$ be a finite set of typed data-flow variables of a program P. We base on the basic elements of *trace semantics* [13, 16] 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 interpretations I for X. An interpretation is a mapping from X to the set of boolean values $\{\mathtt{0}, \mathtt{1}\}$. $I(x) = 1$ if data-flow x holds a value while $I(x) = 0$ if it holds no value.

For example, consider a set of data-flow variables $X = \{x_1, x_2\}$, then the possible clock events are $\mathcal{E}c_X = \{(x_1 \mapsto 0, x_2 \mapsto 0), (x_1 \mapsto 0, x_2 \mapsto 1), (x_1 \mapsto 1, x_2 \mapsto 0), (x_1 \mapsto 1, x_2 \mapsto 1)\}$.

Definition (clock traces). Given a non-empty set of 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 $\mathcal{E}c_X$.

The natural numbers represent the instants $t = 0, 1, 2, ...$, a trace Tc is a chain of clock events along the instants. We denote the interpreted value (0 or 1) of a variable x at instant `t` by $T_c(t)(x)$. Consider the above example, we have $T_c : (0, (x_1 \mapsto 0, x_2 \mapsto 0)), (1, (x_1 \mapsto 1, x_2 \mapsto 0)), ...$ as one of the possible clock traces on X, and $T_c(0)(x_1) = T_c(0)(x_2) = 0$.

Then the clock semantics of a program P is a set of constrained clock traces, denoted by $[[P]]_c$. Table 4 shows the clock semantics of each Signal primitive equation [13].

To show the soundness of our translation, we consider a similar reasoning as in [13]. Let $X = \{x_1, ..., x_n\}$ be a finite set of typed data-flow variables of a synchronous program P and its clock model Φ_P over the corresponding set of clocks $\hat{X} = \{\widehat{x_1}, ..., \widehat{x_n}\}$. Given an interpretation $\hat{I}$

over $\hat{X}$, at a particular instant, it is called a *clock configuration* if and only if $\hat{I} \models \Phi_P$. Given a clock configuration $\hat{I}$, the set of clock events of $\hat{I}$ is computed as: $S_{sat}(\hat{I}) = \{I \in \mathcal{E}c_X \mid \forall i, I(x_i) = \hat{I}(\widehat{x_i})\}$. Then the set of all clock events of clock model Φ_P is $S_{sat}(\Phi_P) = \bigcup_{\hat{I} \models \Phi_P} S_{sat}(\hat{I})$. With a set of clock events $S_{sat}(\Phi_P)$, the concretization of Φ_P is the set of clock traces:

$$\Gamma(\Phi_P) = \{T_c \in \mathcal{T}c_X \mid \forall t, T_c(t) \in S_{sat}(\Phi_P)\} \tag{2}$$

Definition Given the clock model Φ_P, we say that a property φ defined over the set of clocks $\hat{X}$ is satisfied by Φ_P if for any interpretation $\hat{I}$, $\hat{I} \models \Phi_P$ whenever $\hat{I} \models \varphi$, denoted by $\Phi_P \models \varphi$.

Our translation scheme above is sound in term of preserving the clock behaviors of the abstracted program: if a clock model satisfies a property defined over the clocks, then the corresponding program also satisfies this property as stated by the following proposition.

Proposition 5.1 *Let P, Φ_P be a program and its clock model, respectively, φ is a property defined over the clocks. If $\Phi_P \models \varphi$ then $[[P]]_c \subseteq \Gamma(\varphi)$.*

Proof The proof of Proposition 5.1 is done by using Lemma 5.2. Given a clock trace $T_c \in [[P]]_c$, applying Lemma 5.2, $T_c \in \Gamma(\Phi_P)$ means that $\forall t, T_c(t) \in S_{sat}(\Phi_P)$. Since $\Phi_P \models \varphi$, then every interpretation $\hat{I}$ satisfying Φ_P also satisfies φ. Thus, any clock event $I \in S_{sat}(\Phi_P)$ is also in $S_{sat}(\varphi)$, meaning that $\forall t, T_c(t) \in S_{sat}(\varphi)$. Therefore, we have $T_c \in \Gamma(\varphi)$.

Lemma 5.2 *For all program $P, [[P]]_c \subseteq \Gamma(\Phi_P)$.*

Proof We prove it by induction on the structure of program P, meaning that for all primitive operators of the language we show that the clock semantics is a subset of the corresponding concretization.

- Equation on data: $P = y := f(x_1, ..., x_n)$. First, consider y as numerical signal; following the translation scheme, we have the clock model $\Phi_P = \hat{y} \leftrightarrow \widehat{x_1} \leftrightarrow ... \leftrightarrow \widehat{x_n}$. If an interpretation $\hat{I}$ is a model of Φ_P then:

  - either $\forall i, \hat{y} = 0$ and $\widehat{x_i} = 0$;
  - or $\forall i, \hat{y} = 1$ and $\widehat{x_i} = 1$.

  $S_{sat}(\Phi_P)$ is the set of all interpretations of the form above. Let $T_c \in [[P]]_c$ be a clock trace and $t \in \mathbb{N}$ be any instant, then either $\forall i, T_c(t)(y) = T_c(x_i) = 0$ or $T_c(t)(y) = T_c(x_i) = 1$, thus $T_c \in \Gamma(\Phi_P)$. When y is a boolean signal, the proof is similar.

- Delay, sampling, and merging operators: we prove in the same manner.

- Composition: $P = P1|P2$. We have $[[P]]_c \subseteq [[P_1]]_c \subseteq \Gamma(\Phi_{P_1})$ by applying the induction hypothesis. In the same way, we also have $[[P]]_c \subseteq \Gamma(\Phi_{P_2})$. Then, $[[P]]_c \subseteq \Gamma(\Phi_{P_1}) \cap \Gamma(\Phi_{P_2})$. Since $\Gamma(\Phi_{P_1}) \cap \Gamma(\Phi_{P_2}) \subseteq \Gamma(\Phi_{P_1} \wedge \Phi_{P_2})$, we have $[[P]]_c \subseteq \Gamma(\Phi_{P_1} \wedge \Phi_{P_2}) = \Gamma(\Phi_P)$.

- Restriction: $P = P_1$ where x. By definition of clock semantics we have $[[P]]_c \subseteq [[P_1]]_c$ and $\Gamma(\Phi_{P_1}) \subseteq \Gamma(\exists x.\Phi_{P_1})$. Since $[[P_1]] \subseteq \Gamma(\Phi_{P_1})$ by induction then we have the proof.

5.2 Definition of Correct Transformation: Refinement

We adopt the translation validation approach [22, 23] to verify formally that the abstract clock semantics is preserved for every transformation of the compiler. In order to do that, we propose a formal definition of correct transformation on clock models. Consider the two clock models Φ_{P_1}, Φ_{P_2}, to which we refer respectively as a source program and its transformed program produced by the compiler. We assume that they have the same set of variables. We say that P_1 and P_2 have the same clock semantics if they have the same set of clock traces:

$$\forall T_c.(T_c \in \Gamma(\Phi_{P_1}) \leftrightarrow T_c \in \Gamma(\Phi_{P_2})) \tag{3}$$

Requirement (3) is too strong in general to be practice for synchronous data-flow languages. The source language might be non-deterministic, compilers are allowed to select one of the possible behaviors of the source program. Additionally, compilers do transformations, optimizations for removing or eliminating some redundant behaviors of the source program (e.g. eliminating subexpressions, trivial clock constraints). To address these issues, we relax the requirement above as follows:

$$\forall T_c.(T_c \in \Gamma(\Phi_{P_2}) \rightarrow T_c \in \Gamma(\Phi_{P_1})) \tag{4}$$

Requirement (4) says that all clock traces of Φ_{P_2} are clock traces of Φ_{P_1} as well, or $\Gamma(\Phi_{P_2}) \subseteq \Gamma(\Phi_{P_1})$. We say that Φ_{P_2} is a *correct transformation* on abstract clocks of Φ_{P_1} or P_2 *refines* P_1 w.r.t the clock semantics. We write $P_2 \sqsubseteq_{clock} P_1$ to denote the fact that P_2 refines P_1.

With an unverified synchronous data-flow compiler, each compilation task is followed by our refinement verification process to provide formal guarantee as strong as that provided by a formally verified compiler. Indeed, consider the following process:

$$\begin{array}{rcl} Cp'(P_1) & = & \text{if } Cp(P_1) \text{ is} \\ & & \text{Error} \ \rightarrow \ \text{Error} \\ & | & \text{OK}(P_2) \rightarrow \ \text{if } P_2 \sqsubseteq_{clock} P_1 \text{ then OK}(P_2) \text{ else Error} \end{array}$$

where $Cp(P_1)$ is the compilation task from source program P_1 to either compiled code (written as $Cp(P_1) = \text{OK}(P_2)$) or compilation errors (written as $Cp(P_1) = \text{Error}$).

5.3 Proving Refinement

We now discuss an approach to check the existence of refinement between two clock models that is based on the following theorem.

Theorem 5.3 *Given a source program P_1 and its transformed program P_2, P_2 is a correct transformation of P_1 on abstract clocks if it satisfies that for every interpretation $\hat{I}$, if $\hat{I}$ is a clock configuration of Φ_{P_2} then it is a clock configuration of Φ_{P_1}, then $P_2 \sqsubseteq_{clock} P_1$:*

$$\forall \hat{I}.(\hat{I} \models \Phi_{P_2} \rightarrow \hat{I} \models \Phi_{P_1}) \rightarrow P_2 \sqsubseteq_{clock} P_1 \tag{5}$$

Proof To prove Theorem 5.3, we show that if $\forall \hat{I}.(\hat{I} \models \Phi_{P_2} \rightarrow \hat{I} \models \Phi_{P_1})$ then $\Gamma(\Phi_{P_2}) \subseteq \Gamma(\Phi_{P_1})$. Given $T_c \in \Gamma(\Phi_{P_2})$, it means that $\forall t, T_c(t) \in S_{sat}(\Phi_{P_2})$. Since $\forall \hat{I}.(\hat{I} \models \Phi_{P_2} \rightarrow \hat{I} \models \Phi_{P_1})$, thus $S_{sat}(\Phi_{P_2}) \subseteq S_{sat}(\Phi_{P_1})$, meaning that $T_c(t) \in S_{sat}(\Phi_{P_1})$ for every t. Therefore, we have $T_c \in \Gamma(\Phi_{P_1})$.

The checking of the existence of refinement in (5) can be implemented with a SMT-solver such as in [10]. 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. The check formulas belong to decidable theory, this 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 answer that the formula $\neg(\Phi_{P_2} \rightarrow \Phi_{P_1})$ is unsatisfiable. Since our asked formula is over boolean variables, thus the solving is decidable and very efficient [5]. We will show that $\neg(\Phi_{P_2} \rightarrow \Phi_{P_1})$ is unsatisfiable if and only if $\forall \hat{I}.(\hat{I} \models \Phi_{P_2} \rightarrow \hat{I} \models \Phi_{P_1})$ or it is equivalent to show that if $(\Phi_{P_2} \rightarrow \Phi_{P_1})$ is valid then $\forall \hat{I}.((\hat{I} \models \Phi_{P_2}) \rightarrow (\hat{I} \models \Phi_{P_1}))$ and vise-versa. For any interpretation $\hat{I}$ such that $\hat{I} \models \Phi_{P_2}$, it is easy to see that since $(\Phi_{P_2} \rightarrow \Phi_{P_1})$ is valid, for every interpretation $\hat{I}$ if $\hat{I} \models \Phi_{P_2}$ then $\hat{I} \models \Phi_{P_1}$. The inverse direction is based on the definition of validity.

5.4 Implementation

In this section, we describe the main components of the implementation which is integrated in the existing Polychrony toolset [21] to prove the correctness of the Signal compiler on abstract clocks. The compiler [4] consists of a sequence of code transformations. Some transformations are optimizations that rewrite the code to eliminate subexpressions, inefficient expressions. The compilation process may be seen as a sequence of morphisms rewriting Signal programs to Signal programs. And 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 2.

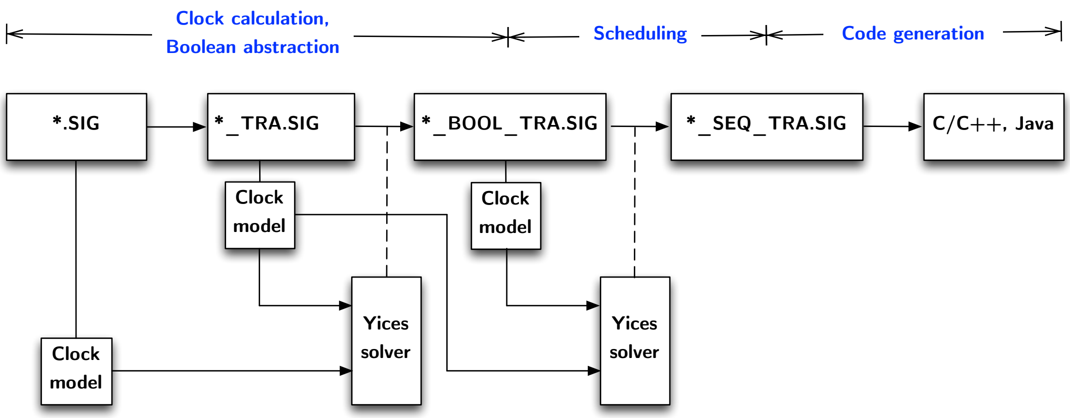

Figure 2: An overview of our integration within Polychrony toolset

The optimized final program `*_SEQ_TRA.SIG` is translated directly to executable code. We are interested in the first stage of the compiler: clock calculation and boolean abstraction. The intermediate forms in the transformations of the compiler may be expressed in the Signal language itself. To prove the correctness of the compiler transformations on abstract clocks our implementation approach takes the input program `P.SIG` and its transformed program `P_TRA.SIG`. It first computes the clock models based on the above translation scheme. The clock models of input and transformed programs are combined as the formula $(\Phi_{P\_TRA.SIG} \rightarrow \Phi_{P.SIG})$. Then it checks that $\models (\Phi_{P\_TRA.SIG} \rightarrow \Phi_{P.SIG})$ or equivalently $M \not\models \neg(\Phi_{P\_TRA.SIG} \rightarrow \Phi_{P.SIG})$. The result of this checking can be exploited for the correctness of the compiler's transformations. If the result says that the checked formula is not valid (or the negation formula is satisfiable) then it emits compilation error. Otherwise, the compiler continues its work. The same procedure is applied for other steps of the compiler. Finally, our verification process asserts that `P_BOOL_TRA.SIG` $\sqsubseteq_{clock}$ `P_TRA.SIG` $\sqsubseteq_{clock}$ `P.SIG` along the transformations of the compiler.

Here, we delegate the checking of the above formula against the clock models to a SMT-solver. Our implementation uses the SMTLIB common format [7] to encode the clock models as input of SMT-solver. For our implementation, we consider the Yices [10] solver, which is one of the best two solvers at the last SMTCOMP competition [24].

6 Related Work and Conclusion

The notion of translation validation was introduced in [22, 23] 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 [6] 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 scheduling semantics, data dependencies of the programs 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 [13]. 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 [8, 19], 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 [9]. With the same purpose, in the work of [20], 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 variable. With the efficiency SMT solver in processing formulas over boolean variables, our approach can deal with large programs whose numbers of variables are huge that make 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 which deterministically characterizes the presence/absence status of all its input/output signals.

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 stages 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 [25] which contains only the computations of data-flows and there is no timing information. We therefore evaluate this graph more efficiently.

References

[1] A. Benveniste, P. Caspi, S.A. Edwards, N. Halbwachs, P. Le Guernic, and R. De Simone: The synchronous languages 12 years later. *Proceedings of the IEEE 91(1).* pp.64-83, 2003

[2] G. Berry: The foundations of Esterel. *In Proof, Language and Interaction: Essay in Honor of Robin Milner, MIT Press*, 2000.

[3] F. Besson, T. Jensen, and J-P. Talpin: Polyhedral analysis for synchronous languages. *In Proceedings of the 6th International Symposium on Static Analysis, volume 1694 LNCS.* pp.51-68, Sep 1999.

[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. Biere, M. Heule, H. van Maaren, and T. Walsh: Handbook of satisfiability: Volume 185 Frontiers in Artificial Intelligence and Applications. *IOS Press, Amsterdam, The Netherlands.* ISBN 978-1-5860-3929-5, 2009.

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

[7] C. Barrett, S. Ranise, A. Stump, and C. Tinelli: The Satisfiability Modulo Theories Library (SMT-LIB). *http://www.SMT-LIB.org*, 2008.

[8] Inria, France: The CompCert Project. *http://compcert.inria.fr.*

[9] Inria, France: The Coq Proof Assitant. *http://coq.inria.fr.*

[10] B. Dutertre, and L. de Moura: Yices sat-solver. *http://yices.csl.ri.com*, 2009.

[11] J.H. Gallier: Logic for computer science. *John Wiley.* 1987.

[12] A. Gamatié: Designing embedded systems with the Signal programming language: Synchronous, Reactive Specification. *Springer, New York.* ISBN 978-1-4419-0940-4, 2009.

[13] 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.

[14] 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.

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

[16] 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.

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

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

[19] 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.

[20] 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.

[21] Inria/Espresso: Polychrony Toolset. *http://www.irisa.fr/espresso/Polychrony.*

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

[23] 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.

[24] A. Stump, and M. Deters: *http://www.smtcomp.org/2009*, 2009.

[25] 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.

RESEARCH CENTRE
RENNES – BRETAGNE ATLANTIQUE

Campus universitaire de Beaulieu
35042 Rennes Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399