Formal Verification of Compiler Transformations on Polychronous Equations

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

Formal Verification of Compiler Transformations on Polychronous Equations

Van Chan Ngo[1], Jean-Pierre Talpin[1], Thierry Gautier[1], Paul Le Guernic[1], and Loïc Besnard[2]

[1] INRIA Rennes-Bretagne Atlantique, 35042 Rennes cedex, France
{Chan.Ngo,Jean-Pierre.Talpin,Thierry.Gautier,Paul.LeGuernic}@inria.fr
[2] IRISA/CNRS, 35042 Rennes cedex, France
Loic.Besnard@irisa.fr

Abstract. In this paper, adopting the translation validation approach, we present a formal verification process to prove the correctness of compiler transformations on systems of polychronous equations. 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. In order to carry out the correctness proof, an appropriate relation called *refinement* and an automated proof method are presented. Each individual transformation or optimization step of the compiler is followed by our validation process which proves the correctness of this running. The compiler will continue its work if and only if the correctness is proved positively. In this paper, the highly optimizing, industrial compiler from the synchronous language SIGNAL to C is addressed.

Keywords: Formal Verification, Translation Validation, Validated Compiler, Multi-clocked Synchronous Programs, Polychronous Model.

1 Introduction

In the synchronous approaches, synchronous data-flow languages such as LUSTRE [9], SIGNAL [7] have been introduced and used successfully for the design and implementation of embedded and critical real-time systems. For the critical, high-assurance systems, the design and realization highly require an efficient and reliable implementation. Thus the systems must be verified using formal methods (e.g. model checking, etc). We want that when the compiler does not claim bugs in the formally verified source code, the generated executable code behaves as abstract clock relations semantics of the source program. However, compilation is complex and compilers involve many phases where they perform transformations over the data structures of the source program. Some transformations might be optimizations based on static analyses to eliminate inefficiencies, subexpressions in the code. Thus, bugs in the compilers can happen, making wrong executable code to be generated from correct source programs. The software industry is aware of these issues and applies many techniques to deal with them, such as

D. Latella and H. Treharne (Eds.): IFM 2012, LNCS 7321, pp. 113–127, 2012.

manual reviews of the generated code, or testing. These techniques are not fully automated, and are expensive in terms of time and performance. An automated formal approach is applied to verify the compiler in order to prove that the semantic of the source program is preserved during the compilation is needed.

In this paper, adopting the *translation validation* approach in [15], we present an automated verification process to prove the correctness of a multi-clocked synchronous language compiler. As a part of the VERISYNC project [18], due to the very important role of abstract clock and clock relations, we are interested in proving that abstract clocks and clock relations semantics of source programs are preserved during the compilation phases of the compiler. Each individual transformation or optimization step of the compiler is followed by our verification process which proves the correctness of this running. The compiler will continue its work if and only if the correctness is proved positively. This approach avoids the disadvantage of proving in advance that the compiler always do correctly since every small change to the compiler requires reproving. Our verification framework uses polynomial dynamical systems (PDS) over a finite field, as common semantics for both source and compiled programs and a syntactic simulation-based proof which automatically proves the semantic preservation. This automated proof is implemented within the existing model checker SIGALI in the Polychronly toolset [12].

The remainder of this paper is organized as follows. Section 2 introduces the formal model of synchronous program behaviors and the automatic translation from a SIGNAL program to its formal model. In Section 3, we present our approaches to formally verify the compilation and formalize the notion of "correct translation" by means of a refinement relation between PDSs. Section 4 addresses the application of our verification approaches to the highly optimizing, industrial compiler from the synchronous language SIGNAL with the implementation which is integrated in the Polychrony toolset. Section 5 describes some related works, concludes our work and describes future work.

2 An Equational Model of Synchronous Programs

2.1 An Equational Model of the Synchronous Program Behavior

We denote by $\mathbb{Z}/p\mathbb{Z}[Z]$ the set of polynomials over variables $Z = \{z_1, ..., z_k\}$ whose coefficients range over $\mathbb{Z}/p\mathbb{Z}$, where $\mathbb{Z}/p\mathbb{Z}$ is the finite field modulo p, with p prime. For a polynomial $P \in \mathbb{Z}/p\mathbb{Z}[Z]$, the solutions of the polynomial equation $P(Z) = 0$ is denoted by $Sol(P)$. We say that $P_1 \equiv P_2$ whenever $Sol(P_1) = Sol(P_2)$. And the representative of $Sol(P)$ of each $\equiv$-equivalence class is called the *canonical generator*. In the following, we shall use some notations:

$$\begin{aligned}
\overline{P} &\triangleq 1 - P^{p-1}. \text{ Thus } (\mathbb{Z}/p\mathbb{Z})^k \backslash Sol(P) = Sol(\overline{P}) \\
P_1 \oplus P_2 &\triangleq (P_1^{p-1} + P_2^{p-1})^{p-1} \\
P_1 \Rightarrow P_2 &\triangleq \{Z \in (\mathbb{Z}/p\mathbb{Z})^k | P_1(Z) = 0 \Rightarrow P_2(Z) = 0\} \equiv \overline{P_1} * P_2 \\
\exists z_i P &\triangleq P|_{z_i=1} * P|_{z_i=2} * ... * P|_{z_i=p} \\
\forall z_i P &\triangleq P|_{z_i=1} \oplus P|_{z_i=2} \oplus ... \oplus P|_{z_i=p}
\end{aligned}$$

where $P|_{z_i=v}$ is P obtained by instantiating any occurrence of variable z_i by value v. The manipulations of polynomials over the finite field modulo p, with p prime can be found in [2].

Synchronous data-flow languages (e.g. LUSTRE, SIGNAL) represent data as an infinite sequence of values called *stream*, and each data stream is combined with an associated *abstract clock* as a means of discrete time. Streams and stream relations, abstract clocks and clock relations are called functional constraints and temporal constraints, respectively. The structure of synchronous programs is usually described as a series of equational definitions, the whole system is represented as systems of equations. This original structure makes that it is natural to represent the program behaviors in terms of systems of equations. The compilers of these languages, such as that we consider here, are composed of a sequence of code transformations. The transformations and optimizations that rewrite or translate source code to eliminate inefficiencies of functional constraints and temporal constraints. Some of the transformations are non-optimizing translations from a synchronous language or its intermediate language to another, lower-level language (e.g. C, Java code). Abstract clocks and clock relations are used to represent all the control parts (e.g. activation events) and interaction between different components in system. The control flow resulting from the analysis of abstract clocks and clock relations is used to derive an optimized data-flow following the transformations of the compiler. Therefore, the correctness of clock analysis in synchronous language compilation strongly impacts the quality of the compiled program. And as we have mentioned above, we would like to cope with the semantics of abstract clocks and clock constraints. In other words, our aim is to build formal models which represent the behaviors of synchronous data-flow programs in terms of the presence, absence of values in a stream (abstract clock) and the clock relations. The principle is to encode the status of a value in a stream with two possible values: *absence* and *presence*. We will use the finite field modulo $p = 3, \mathbb{Z}/3\mathbb{Z}$, i.e. integers modulo $3 : \{-1, 0, 1\}$ to encode the states of values in a data stream. For the Boolean data stream x, three possible states of x at an instant time are encoded as: $present \wedge true \rightarrow 1; present \wedge false \rightarrow -1; absent \rightarrow 0$. For the non-boolean data streams, it only encodes the fact that the value is present or absent (the clock value of the data stream is *true* or *false*): $present \rightarrow \pm 1; absent \rightarrow 0$. And the clock of a data stream is the square $x^2 : 1\ if\ present,\ 0\ if\ absent$. Thus, two synchronous data streams (they have the same clock) x and y satisfy the constraint equation: $x^2 = y^2$. It is obvious that the abstract clocks and clock relations of a synchronous data-flow program can be modeled efficiently with PDSs with coefficients ranging over $\mathbb{Z}/3\mathbb{Z}$.

Definition 1. *A PDS is a system of equations which is organized into three subsystems of polynomial equations of the form:*

$$\begin{cases} Q(X,Y) = & 0 \\ X' & = P(X,Y) \\ Q_0(X) & = 0 \end{cases}$$

where:

- X *is a set of* n *variables, called state variables, represented by a vector in* $(\mathbb{Z}/3\mathbb{Z})^n$*;*
- Y *is a set of* m *variables, called event variables, represented by a vector in* $(\mathbb{Z}/3\mathbb{Z})^m$*;*
- $X' = P(X, Y)$ *is the evolution equation of the system. It can be considered as a vectorial function* $[P_1, ..., P_n]$ *from* $(\mathbb{Z}/3\mathbb{Z})^{n+m}$ *to* $(\mathbb{Z}/3\mathbb{Z})^n$*;*
- $Q(X, Y) = 0$ *is the constraint equation of the system. It is a vectorial equation* $[Q_1, ..., Q_l]$*;*
- $Q_0(X) = 0$ *is the initialization equation of the system. It is a vectorial equation* $[Q_{0_1}, ..., Q_{0_n}]$.

Synchronous data-flow languages use some operators requiring memorization of past value of a data stream, that is done by introducing the state variables. The vector values $(x_1, ..., x_n)$, $(x'_1, ..., x'_n)$ store respectively the past values and the current values of the data streams that are involved in the memorizing operators (e.g. SIGNAL *delay* operator). Systems of polynomial equations characterize sets of solutions, which are *states* and *events* of programs. A system of equation based method consists in manipulating the equation systems instead of the solution sets, avoiding the enumeration of the state space [2]. There is no terminal state since a synchronous data-flow program takes the input data streams that are infinite flows of values, for every state of its PDS there exist always the events to produce the next state.

2.2 Overview of the SIGNAL Language Features

In SIGNAL language [8], a signal noted as x, is a *sequence of values with the same type* $x(t_i)_{i\in\mathbb{N}}$*, which are present at some instants.* The set of instants (or time tags) where a signal is present is the *clock* of the signal, noted $\hat{x}$. A particular type of signal called *event* is characterized only by its presence, and always has the value *true*. The constructs of the language use an equational style to specify the relations between signals in the form $\mathcal{R}(x_1, ..., x_k)$, where the values of signals and the abstract clocks of signals $x_1, ..., x_k$ are the functional constraint and temporal constraint, respectively. Systems of equations on signals are built using a composition construct which defines a *process*. A whole SIGNAL 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. 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 an *n-ary* relation over numerical or boolean data types, defines a process whose output $y(t)$ for tag $t \in \hat{y}$ 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 $y := x\$1$ init a defines a process whose output $y(t_i) = a$ if t_i is the initial time tag, and for every other tag, $y(t_i) = x(t_{i-1})$. The clock constraint of the input and output signals is $\hat{y} = \hat{x}$.
- *Merge:* The merge equation $y := x$ default z defines a process whose output at time tag 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 $y := x$ 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 SIGNAL language allows clock constraints to be defined *explicitly* by equations. The signal's clock is represented in SIGNAL by a special signal of type *event* which carries only a single value *true*. It specifies the presence of the signal, denoted $\hat{}x$. 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 $x \hat{}+ y \triangleq \hat{}x$ default $\hat{}y$, *(iii)* clock intersection relationship $x \hat{}* y \triangleq \hat{}x$ when $\hat{}y$.

Furthermore, the unary form of the sampling operation *when b* returns an event signal representing the clock of $[b]$. The special event signal $\hat{}0$ denotes the null clock (the clock that is never present).

2.3 PDS Model of SIGNAL Programs

In order to model SIGNAL programs behaviors, their processes are translated into systems of polynomial equations over $\mathbb{Z}/3\mathbb{Z}$. Each individual SIGNAL equation is translated into a polynomial equation. The language uses some primitive equations to construct programs. Thus, we only need to define the translation of these primitive equations to polynomial equations over the finite field $(\mathbb{Z}/3\mathbb{Z})^n$. The composition equation type is simply translated as the combination of the polynomial equations in the same equation system. For the equations on clocks they are derived directly from the primitive equations. Table 1 shows the translation of the primitive equations of the SIGNAL language. The delay operator \$ requires memorizing the past value of the signal, that is done by introducing the *state variable* ξ, where ξ stores the previous value of the signal and ξ' stores the current value of the signal. For example the simple SIGNAL program shown in Table 2 that specifies the alternative presence between the input signals A and B is translated in the PDS model with variables a, b, x and zx corresponding to the events A, B and boolean signals X and ZX and a state variable ξ for the delay operator. In particular, SIGNAL allows one to explicitly manipulate clocks

through some derived con- structs that can be rewritten in terms of primitive ones. For instance, $y :=$ when b is equivalent to $y := b$ when b.

Table 1. Translation of the primitive equations

Boolean signals		Non-boolean signals	
$y :=$ not x	$y = -x$	$y := f(x_1, ..., x_n)$	$y^2 = x_1^2 = ... = x_n^2$
$z := x$ and y	$z = xy(xy - x - y - 1)$ $x^2 = y^2$		
$z := x$ or y	$z = xy(1 - x - y - xy)$ $x^2 = y^2$		
$z := x$ default y $z := x$ when y	$z = x + (1 - x^2)y$ $z = x(-y - y^2)y$	$z := x$ default y $z := x$ when y	$z^2 = x^2 + y^2 - x^2y^2$ $z^2 = x^2(-y - y^2)$
$y := x\$1$ init y_0	$\xi' = x + (1 - x^2)\xi$ $y = x^2\xi$ $\xi_0 = y_0$	$y := x\$1$ init y_0	$y^2 = x^2$

Table 2. Program *altern* and its PDS model

```
process altern =
 ( ? event A, B;
   ! )
 (| X := not ZX
  | ZX := X$ 1
  | A ^= when X
  | B ^= when ZX
  |)
 where
 boolean X, ZX init false;
 end;
```

initial equations:
$\xi = -1$
evolution equations:
$\xi' = x + (1 - x^2) * \xi$
constraint equations:
$x = -zx, zx = \xi * x^2$,
$a^2 = -x - x^2, b^2 = -zx - zx^2$

3 Formally Verified Compilation Approaches

3.1 Definition of Correct Translation: Refinement

Given a PDS model L over the finite field $\mathbb{Z}/3\mathbb{Z}$, it can be viewed as an *intensional Labeled Transition System* (iLTS) [10] as defined in Definition 2:

Definition 2. *An intensional Labeled Transition System is a structure* $L = (Q, Y, \mathcal{I}, \mathcal{T})$*, where* Q *is a set of states,* Y *is a set of* m *variables* $Y_1, ..., Y_m$*,* $\mathcal{I}$ *is a set of initial states, and* $\mathcal{T} \subseteq Q \times \mathbb{Z}/3\mathbb{Z}[Y] \times Q$ *is the transition relation. Each transition is labeled by a polynomial over the set* Y*.*

The iLTS representation of a PDS can be obtained directly from the set of state variables, event variables, systems of initial equations, evolution equations, and constraint equations as follows:

- $Q = \mathcal{D}_X$, where $\mathcal{D}_X = \prod_{i\in[1,n]} \mathcal{D}_{x_i} = (\mathbb{Z}/3\mathbb{Z})^n$ as the domain of a set of variables $X = (x_1, ..., x_n)$
- $Y = Y, \mathcal{D}_Y = \prod_{i\in[1,m]} \mathcal{D}_{y_i} = (\mathbb{Z}/3\mathbb{Z})^m$
- $\mathcal{I} = Sol(Q_0(X))$
- $(q, P_q(Y), q') \in \mathcal{T}$ where $P_q(Y) \equiv Q(q, Y) \oplus (P(q, Y) - q')$

We write $q \xrightarrow{P(Y)} q'$ (or for short $q \xrightarrow{P} q'$), instead of $(q, P(Y), q') \in \mathcal{T}$. Then iLTSs can be viewed as an "intensional" representation of classical LTSs, where the labels are tuples in $(\mathbb{Z}/3\mathbb{Z})^m$: each arrow of the iLTS labeled by $P(Y)$ intensionally represents as many arrows labeled by some $y \in Sol(P(Y))$. We will call $Ext(L)$ the corresponding "extensional" LTS.

Definition 3. *Let $L = (Q, Y, \mathcal{I}, \mathcal{T})$ an iLTS. The infinite sequence $\sigma = q_0, y_0, q_1, y_1, q_2, y_2, ...,$ where $q_i \in Q, y_i \in \mathcal{D}_Y$ for each $i \in \mathbb{N}$, is an execution of L if it satisfies the following requirements:*

- $q_0 \in \mathcal{I}$.
- *there exists a polynomial $P(Y)$ such that $(q_i, P(Y), q_{i+1}) \in \mathcal{T} \wedge y_i \in Sol(P(Y))$ for each $i \in \mathbb{N}$.*

We denote by $\sigma_{act} = y_0, y_1, y_2, ...$ is an action-based execution, $||L||, ||L||_{act}$ the sets of executions and action-based executions of the iLTS L, respectively.

Consider the two iLTSs $A = (Q_2, Y, \mathcal{I}_2, \mathcal{T}_2)$ and $C = (Q_1, Y, \mathcal{I}_1, \mathcal{T}_1)$, to which we refer respectively as a source program and a compiled program produced by a synchronous data-flow compiler. We assume that they have the same set of event variables. In case the set of event variables of the compiled model is different from the set of event variables of the source model, we consider only the common event variable and the different event variables are considered as *hiding events* [14]. Our aim is to prove that the desired behaviors of the source program are preserved during the compilation. In our case, the set of action-based executions models the desired behaviors of the program. The behaviors reflect the states of data streams and the data stream clocks constraints of the program. The strongest notion of behavior preservation during compilation is that the source program A and its compiled program C have exactly the same desired behaviors:

$$\forall \sigma_{act}.\ (\sigma_{act} \in ||C||_{act} \Leftrightarrow \sigma_{act} \in ||A||_{act}) \tag{1}$$

Requirement (1) is too strong in general to be in practical for synchronous data-flow languages. The source language is usually non-deterministic, compilers are allowed to select one of the possible behaviors of the source program. In this case, the compiled program C will have fewer behaviors than the source program A. Additionally, compilers do transformations, optimizations for removing or eliminating some wrong behaviors of the source program (e.g. eliminating subexpressions, trivial clock constraints). To address these issues, we relax the requirement (1) as follows:

$$\forall \sigma_{act}.(\sigma_{act} \in ||C||_{act} \Rightarrow \sigma_{act} \in ||A||_{act}) \tag{2}$$

Requirement (2) says that all action-based executions of C are acceptable executions of A. And we say that C *refines* A w.r.t action-based executions. We write $C \sqsubseteq A$ to denote the fact that C refines A. In the next section we present a method to establish the refinement between the two given models C and A.

3.2 Proving Refinement by Simulation

We now discuss an approach to automatically reason that a compiler preserves semantics of the source program during its compilation, in the sense of refinement relation. Given two iLTSs A and C, we propose a *symbolic simulation* for the two iLTSs to establish that $C \sqsubseteq A$. The symbolic simulation satisfies the property that if there exists a symbolic simulation for (C, A) then $C \sqsubseteq A$.

Definition 4. *Let $C = (Q_1, Y, \mathcal{I}_1, \mathcal{T}_1)$ and $A = (Q_2, Y, \mathcal{I}_2, \mathcal{T}_2)$ be two iLTSs. A symbolic simulation for (C, A) is a binary relation $\mathcal{R} \subseteq Q_1 \times Q_2$ which satisfies the following properties:*

- *(A) $\forall q_1 \in \mathcal{I}_1, \exists q_2 \in \mathcal{I}_2, (q_1, q_2) \in \mathcal{R}$.*
- *(B) for any $(q_1, q_2) \in \mathcal{R}$ it holds that: if $q_1 \xrightarrow{P} q_1'$ there exists a finite set of transitions $(q_2 \xrightarrow{P_i} q_2^i)_{i \in I}$ (where I is a set of indexes) with*
  - *$(P \Rightarrow \prod_{i \in I} P_i) \equiv 0$ and*
  - *$(q_1', q_2^i) \in \mathcal{R}, \forall i \in I$.*

$(P \Rightarrow \prod_{i \in I} P_i) \equiv 0$ denotes that the polynomial $(P \Rightarrow \prod_{i \in I} P_i)$ is equivalent to the zero polynomial, which means that $Sol((P \Rightarrow \prod_{i \in I} P_i)) = Sol(0) = (\mathbb{Z}/3\mathbb{Z})^m$ or $Sol(P) \subseteq Sol(\prod_{i \in I} P_i)$. Condition (A) asserts that every initial state of C is related to an initial state of A. According to condition (B), for every transition of the state q_1 which is labeled by the set of events (or actions) represented by $Sol(P(Y))$, there exist some transitions of the state q_2 which are labeled by the same set of events. And it states that every outgoing transition from q_1 must be matched by outgoing transitions from q_2. Thus, Definition 4 captures exactly classic action-based simulation definition of standard LTSs. Since symbolic simulation is closed under arbitrary unions, there is a greatest symbolic simulation. In the following parts, when we talking about symbolic simulation, we imply talk about the greatest symbolic simulation.

C is simulated by A (or, equivalently, A simulates C), denoted $C \preceq A$, if there exists a symbolic simulation for (C, A). Given two states $q_1 \in Q_1$ and $q_2 \in Q_2$, the state q_1 is simulated by q_2, denoted $q_1 \preceq q_2$, if there exists a symbolic simulation $\mathcal{R}$ for (C, A) with $(q_1, q_2) \in \mathcal{R}$. In that case, we say that the two states "q_1 and q_2 are similar".

Definition 5. *Let $C = (Q_1, Y, \mathcal{I}_1, \mathcal{T}_1)$ and $A = (Q_2, Y, \mathcal{I}_2, \mathcal{T}_2)$ be two iLTSs. We define a family of binary relations $\preceq_j \subseteq Q_1 \times Q_2$ by induction over $j \in \mathbb{N}$.*

- *$\preceq_0 \triangleq Q_1 \times Q_2$.*

– $q_1 \preceq_{(j+1)} q_2$ *iff for all* $(q_1, P, q_1') \in \mathcal{T}_1$*, there exists a finite set of transitions* $(q_2, P_i, q_2^i)_{i \in I}$ *with* $(P \Rightarrow \Pi_{i \in I} P_i) \equiv 0 \wedge q_1' \preceq_j q_2^i$ *for all* $i \in I$*, where* I *is a set of indexes.*

Based on the above definition, we can now have the following theorem which gives us a method to compute the greatest symbolic simulation for two iLTSs.

Theorem 1. *Let* $C = (Q_1, Y, \mathcal{I}_1, \mathcal{T}_1)$ *and* $A = (Q_2, Y, \mathcal{I}_2, \mathcal{T}_2)$ *be two iLTSs.*

1. *There exists a symbolic simulation for* (C, A) *if and only if there exists a simulation for* $(Ext(C), Ext(A))$.
2. *Then for all* $q_1 \in Q_1$ *and* $q_2 \in Q_2, q_1 \preceq q_2$ *iff* $q_1(\bigcap_{n \in \mathbb{N}} \preceq_n) q_2$*, where* $(\bigcap_{n \in \mathbb{N}} \preceq_n) = \preceq_0 \cap \preceq_1 \cap ... \cap \preceq_n$.

Proof. (1) The proof can be found in [10].
(2) Since the number of state variables, event variables and the value domain of a PDS are finite then its iLTS is finite. Symbolic simulation over a finite iLTS (therefore finitely branching) is the limit of nested projective equivalences. Thus we can use the same proof method as in [16] for strong simulation. We omit the proof here.

The use of a symbolic simulation as a proof method to establish the refinement between the two given models C and A is stated in the following theorem.

Theorem 2. *Let* $C = (Q_1, Y, \mathcal{I}_1, \mathcal{T}_1)$ *and* $A = (Q_2, Y, \mathcal{I}_2, \mathcal{T}_2)$ *be two iLTSs. If there exists a symbolic simulation for* (C, A)*, then* $C \sqsubseteq A$.

Proof. The proof of Theorem 2 is trivial with following Lemma 1.

Lemma 1. *Let* C *and* A *be iLTSs,* $\mathcal{R}$ *is a symbolic simulation for* (C, A)*, and* $(q_1, q_2) \in \mathcal{R}$*. Then for each infinite (or finite) execution* $\sigma_1 = q_{0,1}, y_{0,1}, q_{1,1}, y_{1,1}, q_{2,1}, y_{2,1}, ...$ *starting in* $q_{0,1} = q_1$ *there exists an execution* $\sigma_2 = q_{0,2}, y_{0,2}, q_{1,2}, y_{1,2}, q_{2,1}, y_{2,2}, ...$ *from state* $q_{0,2} = q_2$ *of the same length such that* $(q_{j,1}, q_{j,2}) \in \mathcal{R}$ *and* $y_{j,1} = y_{j,2}$ *for all* j.

Proof. Due to the lack of space, we omit the proof here.

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

$$\begin{array}{rl} Cp'(A) = & \text{if } Cp(A) \text{ is} \\ & \quad \text{Error} \rightarrow \text{Error} \\ & \mid \quad \text{OK}(C) \rightarrow \text{ if } C \sqsubseteq A \text{ then OK}(C) \text{ else Error} \end{array}$$

where $Cp(A)$ is the compilation of A to either compiled code (written as $Cp(A) = \text{OK}(C)$) or compilation errors (written as $Cp(A) = \text{Error}$).

3.3 Composition of Compilation Phases

Compilation is always decomposed into several phases of transformations, optimizations through intermediate representations. It is better to decompose the verification process too. Fortunately, our verification process can be decomposed well thanks to the transitive property of symbolic simulation. Let A, I and C are three iLTSs, if $I \preceq A$ and $C \preceq I$ then $C \preceq A$ (the proof is trivial based on the definition of symbolic simulation). We assume that there are two compilation stages Cp_1 and Cp_2 from source program A to I and I to C, respectively. Consider the composition compilation as follows:

$$\begin{array}{rl} Cp(A) = & \text{if } Cp_1(A) \text{ is} \\ & \quad \text{Error} \rightarrow \text{Error} \\ \mid & \quad \text{OK}(I) \rightarrow \text{ if } I \sqsubseteq A \text{ then } Cp_2(I) \text{ else Error} \end{array}$$

It is obvious to see that the compilation $Cp(A)$ is formally verified from A to C.

4 Proving the SIGNAL Compiler

4.1 Implementation of Symbolic Simulation with SIGALI

In this section, we discuss how to implement the proof method with symbolic simulation for the two iLTSs of a source program and its compiled form using the companion model-checker of the Polychrony toolset, SIGALI. Symbolic simulation can be implemented as an extended library of SIGALI, we represent a PDS as an iLTS in the more specific form $L = (X, X', Y, \mathcal{I}, \mathcal{T})$, where:

- X, X', Y are the sets of state and event variables as in the PDS,
- $\mathcal{I}(X) = Q_0(X)$ is the polynomial representing the set of initial states, $Sol(I)$,
- $\mathcal{T}(X, Y, X') \equiv Q(X, Y) \oplus (P(X, Y) - X')$ is the polynomial representing the set of transitions.

In SIGALI, polynomials are internally represented as *ternary decision diagrams* (TDD) [5] which are an extension of *binary decision diagrams* (BDD) [1]. They are convenient for an efficient manipulation the polynomial equation systems. Theorem 1 gives us an iterative algorithm to compute the greatest symbolic simulation for (C, A). It can be obtained by computing the convergence of the sequence $(\mathcal{R}_j)_{j\in\mathbb{N}}$ as in Algorithm 1 which can be efficiently implemented with the fixed-point computation of the SIGALI kernel (see Appendix B). The correctness of Algorithm 1 is proved by the following proposition.

Proposition 1. *For all* $j \in \mathbb{N}, \mathcal{R}_j(x_1, x_2) = 0$ *if and only if* $x_1 \preceq_j x_2$.

Proof. $\Rightarrow$) We use an induction proving method over j. It holds obviously with $j = 0$. Assume that we have $\mathcal{R}_{j+1}(x_1, x_2) = 0$ and let $x_1 \xrightarrow{P} x_1'$ be a transition in C. It is clear that $P(Y) \equiv \mathcal{T}_1(x_1, Y, x_1')$. We define the polynomial $Q(Y) \equiv \exists x_2' \mathcal{T}_2(x_2, Y, x_2') \oplus \mathcal{R}_j(x_1', x_2')$, $\mathcal{R}_j$ being computed in Algorithm 1 above. This

Algorithm 1. Compute symbolic simulation $\mathcal{R}(X_1, X_2)$

Require: $C = (X_1, X_1', Y, \mathcal{I}_1, \mathcal{T}_1), A = (X_2, X_2', Y, \mathcal{I}_2, \mathcal{T}_2)$
Ensure: $\mathcal{R}(X_1, X_2)$
1: $\mathcal{R}_0(X_1, X_2) \equiv 0$
2: **while** $\mathcal{R}_j(X_1, X_2)$ is not convergent **do**
3: $\mathcal{R}_{j+1}(X_1, X_2)$ is the canonical generator of the $\equiv$-class of:
4: $\mathcal{R}_j(X_1, X_2) \oplus$
5: $\forall X_1' \forall Y[(\mathcal{T}_1(X_1, Y, X_1') \Rightarrow \exists X_2'(\mathcal{T}_2(X_2, Y, X_2') \oplus \mathcal{R}_j(X_1', X_2'))]$
6: **end while**
7: **if** $\forall X_1[(\mathcal{I}_1(X_1) \Rightarrow \exists X_2(\mathcal{I}_2(X_2) \oplus \mathcal{R}(X_1, X_2))]$ **then**
8:
9: **return** $\mathcal{R}(X_1, X_2)$
10: **else**
11: **return** $\mathcal{R}(X_1, X_2) \equiv 1$
12: **end if**

polynomial captures the set $\{y | \exists x_2 \xrightarrow{P_i} x_2^i, P_i(y) = 0 \wedge x_1' \preceq_j x_2^i\}$. By the definition of $\mathcal{R}_{j+1}$, the y value is in $Sol(\mathcal{T}_1(x_1, Y, x_1'))$, thus $Sol(P(Y)) \subseteq \bigcup_i Sol(P_i)$, which means $x_1 \preceq_{(j+1)} x_2$.
$\Leftarrow$) We can apply again an induction method over j similar to the proof of the Theorem 1. Thus we omit it here.

Proposition 2. *Algorithm 1 terminates and at the end,* $\mathcal{R}(x_1, x_2) = 0$ *if and only if* $x_1 \preceq x_2$.

Proof. Termination is guaranteed by the fact that relations $\mathcal{R}_j$ are finite and nested. The second statement is a corollary of Proposition 1 and Theorem 1.

4.2 Proving the Compiler Transformations

The compiler of the SIGNAL language [3] that we consider is composed of a sequence of code transformations. Some transformations are optimizations that rewrite the code to eliminate subexpressions, inefficiencies. 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 SIGNAL program. For convenience, the transformations of the compiler are classed into three stages:

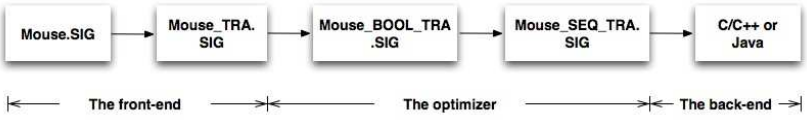

Fig. 1. Scheme of the SIGNAL compiler

- *The front-end*: non-optimizing translations from the source program in SIGNAL language to a program in SIGNAL language. The clock information of all signals in the source program is calculated, which is called *clock calculus*.
- *The optimizer*: the synchronization and precedence relations of all signals and clocks are represented in a directed labeled graph structure called the *Data Control Graph* (DCG); it is composed of a *Clock Hierarchy* (CH) and a *Conditioned Precedence Graph* (CPG). A node of this CPG is a primitive equation or, in a hierarchical organization, a composite SIGNAL process containing its own DCG. Then the optimizations are performed on the output of the front-end stage based on the DCG.
- *The back-end*: translations from the optimized final SIGNAL program to executable code (C/C++ or Java).

For instance, consider a source program called *Mouse.SIG* (example program available in the online examples of the Polychrony toolset), the transformations of the stages front-end, optimizer, back-end are *Mouse_TRA.SIG*, *Mouse_BOOL_TRA.SIG*, and *Mouse_SEQ_TRA.SIG*, respectively.

The optimized final program *Mouse_SEQ_TRA.SIG* is translated directly to executable code. We are interested in the first two stages of the compiler: the non-optimizing translations and the optimizations. The intermediate forms in the transformations of the compiler may be expressed in the SIGNAL language itself. Moreover the Polychrony toolset provides a function to translate a SIGNAL program into a PDS over the finite field $\mathbb{Z}/3\mathbb{Z}$. Then the correctness of the compiler is proved in each transformation of the two first stages. For instance, we consider the compilation of *Mouse.SIG* program, the verification asserts that $Mouse\_SEQ\_TRA.SIG \preceq Mouse\_BOOL\_TRA.SIG \preceq Mouse\_TRA.SIG \preceq Mouse.SIG$ along the transformations of the SIGNAL compiler.

Experimental Results. We here provide some experimental results verifying the transformations of the SIGNAL compiler with a simulation based proof method. The experimental results deal with the complexity of the symbolic simulation computation. All the examples here are available in the online examples of the Polychrony toolset. In the X, Y, 'Correct' columns, we write the numbers of state variables, event variables and the correctness of the compiler transformations, respectively (hence, the transition relation $\mathcal{T}(X, Y, X')$ will have $2X + Y$ variables). We measure description complexity of the symbolic simulation by the size of fix point computation in Algorithm 1 (in terms of the number of TDD nodes that we need to represent the manipulation of polynomial equation systems). The number of TDD nodes is showed in SIGALI model checker only when it is big enough, so for the tests whose numbers of TDD nodes are not showed we write "Small". We denote $\mathcal{R}_1(X_1, X_2), \mathcal{R}_2(X_1, X_2), \mathcal{R}_3(X_1, X_2)$ are symbolic simulations for $(A\_TRA.z3z, A.z3z)$,$(A\_BOOL\_TRA.z3z, A\_TRA.z3z)$, and $(A\_SEQ\_TRA.z3z, A\_BOOL\_TRA.z3z)$, respectively, for the compilation of the SIGNAL program, called A.

Table 3. Experimental results

Name	X	Y	$\mathcal{R}_1(X_1, X_2)$ TDD nodes	$\mathcal{R}_2(X_1, X_2)$ TDD nodes	$\mathcal{R}_3(X_1, X_2)$ TDD nodes	Correct
MOUSE.z3z	2	5	Small	Small	Small	Yes
MOUSE_TRA.z3z	2	5				
MOUSE_BOOL_TRA.z3z	2	6				
MOUSE_SEQ_TRA.z3z	2	6				
RAILROADCROSSING.z3z	2	40	Small	Small	Small	Yes
RRCROSSING_TRA.z3z	2	40				
RRCROSSING_BOOL_TRA.z3z	2	39				
RRCROSSING_SEQ_TRA.z3z	2	39				
CHRONOMETER.z3z	6	33	Small	Small	Small	Yes
CHRONOMETER_TRA.z3z	6	33				
CHRONOMETER_BOOL_TRA.z3z	6	37				
CHRONOMETER_SEQ_TRA.z3z	6	37				
ALARM.z3z	19	45	3775163	3810301	4721454	Yes
ALARM_TRA.z3z	19	45				
ALARM_BOOL_TRA.z3z	19	53				
ALARM_SEQ_TRA.z3z	19	53				

5 Related Work and Conclusions

The notion of translation validation was introduced in [15] by A. Pnueli et al. to verify the code generator of SIGNAL. In this 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 SIGNAL program and its generated C code. Then they use BDD representations to implement the symbolic models STSs, and their proof method uses a SAT-solver to reason on the signals and clock constraints of STSs. It amounts to the mapping for selected states, consisting of the values of input-output-memory variables, for the source and the target code. The drawback of this approach is that 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 SIGNAL program, the logic formula is asked to SAT-solver to solve is very large that makes some inefficiency. Another related work is the approach of J. C. Peralta et al. [13] in a similar approach as the work of A. Pnueli et al. In particular, they translate both the SIGNAL (multi-clocked) specifications given in SIGNAL language and its generated code C/C++ or Java simulator into LTSs. Then, an appropriate pre-order test on both LTSs can be interpreted as a refinement between a generated code implementation and its source SIGNAL specification. The refinement they propose is a bisimulation relation and they use the existing tools to generate the greatest bisimulation relation for the source SIGNAL specification and the target generated code in C/C++. In case there is no bisimulation relation, counterexamples are generated automatically. However, this approach has not been fully automated.

This paper presents the correctness proof of the transformations, optimizations of the multi-clocked synchronous programming language compiler and applies this approach to the highly industrial synchronous data-flow language SIGNAL's compiler. We are interested in proving that abstract clocks and clock relations semantics of source programs are preserved during the compilation phases of the compiler. The desired behaviors of a given source program and its compiled program are represented as PDSs over the finite field of integers modulo $p = 3$. A refinement relation between the source program and its compiled form is used to express the preservation. A proof by simulation is presented to establish the refinement relation. Each compilation stage is followed by our refinement verification process to provide formal guarantees as strong as those provided by a formally verified compiler. If the compilation task from the source program to the compiled form applies without compilation errors, and the compiled form refines the source program, then the compiled form is produced as output else the compiler terminates with an error.

We have implemented and integrated our verification process within the Polychrony toolset by extending the functionality of the existing model checker SIGALI to prove the correctness of the front-end and optimizations phases of the optimizing SIGNAL compiler.

As future work, given a synchronous data-flow program and the corresponding generated C/C++ code, we would like to formally verify that the generated code correctly implements the source program. As we have shown, the verification process can be decomposed into several stages as the decomposition of the compilation task, thanks to the transitive property of symbolic simulation. Thus we only need to prove that there exists a symbolic simulation for the generated C/C++ code and the optimized final program given that the optimized final program refines the source program. In order to do that, we could first translate the asynchronous C/C++ code into the synchronous language SIGNAL. One of the methods is to represent C/C++ code in the Static Single Assignment (SSA) intermediate form and then translate the SSA intermediate form into SIGNAL [4]. The rest of work is the same as the verification process we have presented in this paper.

References

1. Bryant, R.: Graph-based algorithms for boolean function manipulation. IEEE Transactions on Computers C-35(8), 677–691 (1986)
2. Le Borgne, M., Benveniste, A., Le Guernic, P.: Dynamical systems over Galois fields and control problems. In: Proceedings of 33th IEEE on Decision and Control, vol. 3, pp. 1505–1509 (1991)
3. Besnard, L., Gautier, T., Le Guernic, P., Talpin, J.-P.: Compilation of polychronous data flow equations. In: Synthesis of Embedded Software. Springer (2010)
4. Besnard, L., Gautier, T., Moy, M., Talpin, J.-P., Johnson, K., Maraninchi, F.: Automatic translation of C/C++ parallel code into synchronous formalism using an SSA intermediate form. In: Proceedings of the 9th Workshop on Automated Verification of Critical Systems, AVOCS (2009)

5. Dutertre, B., Le Borgne, M., Marchand, H.: SIGALI: un système de calcul formel pour la vérification de programmes SIGNAL. Manuel d'utilisation. Note technique, non publiée (December 1998)
6. Park, D.: Concurrency and Automata on Infinite Sequences. In: Deussen, P. (ed.) GI-TCS 1981. LNCS, vol. 104, pp. 167–183. Springer, Heidelberg (1981)
7. Gamatie, A.: Designing embedded systems with the SIGNAL programming: Synchronous, Reactive Specification. Springer, New York (2009) ISBN 978-1-4419-0940-4
8. Le Guernic, P., Talpin, J.-P., Le Lann, J.-C.: Polychrony for system design. Journal for Circuits, Systems and Computers 12(3), 261–304 (2003)
9. Halbwachs, N.: A synchronous language at work: the story of LUSTRE. In: 3th ACM-IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE 2005) (July 2005)
10. Kouchnarenko, O., Pinchinat, S.: Intensional approaches for symbolic methods. Electronic Notes in Theoretical Computer Science (August 1998)
11. Marchand, H., Rutten, H., Le Borgne, E., Samaan, M.: Formal verification of SIGNAL programs: Application to a power transformer station controller. Science of Computer Programming 41(1), 85–104 (2001)
12. Polychrony Toolset, `http://www.irisa.fr/espresso/Polychrony/`
13. Peralta, J.C., Gautier, T., Besnard, L., Le Guernic, P.: LTSs for translation validation of (multi-clocked) SIGNAL specifications. In: 8th IEEE/ACM International Conference on Formal Method and Models for Codesign, MEMOCODE (2010)
14. Pinchinat, S., Marchand, H., Le Borgne, M.: Symbolic abstractions of automata and their application to the supervisory control problem. In: INRIA Technical Reports No 1279, pp. 1–29 (November 1999)
15. Pnueli, A., Shtrichman, O., Siegel, M.D.: Translation validation: From SIGNAL to C. In: Olderog, E.-R., Steffen, B. (eds.) Correct System Design. LNCS, vol. 1710, pp. 231–255. Springer, Heidelberg (1999)
16. Milner, R.: Operational and algebraic semantics of concurrent processes. Research Report ECS-LFCS-88-46, Lab. for Foundations of Computer Science, Edinburgh (February 1988)
17. Van Glabbeek, R.J.: The Linear Time-Branching Time Spectrum II: The Semantics of Sequential Systems with Silent Moves (Extended Abstract). In: Best, E. (ed.) CONCUR 1993. LNCS, vol. 715, pp. 66–81. Springer, Heidelberg (1993)
18. VeriSync Project, `http://www.irit.fr/Verisync/`