HAL
open science

Formal Verification of Synchronous Data-flow Compilers

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

To cite this version:

van Chan Ngo, Jean-Pierre Talpin, Thierry Gautier, Paul Le Guernic, Loic Besnard. Formal Verification of Synchronous Data-flow Compilers. [Research Report] RR-7921, 2012. hal-00685633v1

HAL Id: hal-00685633

https://inria.hal.science/hal-00685633v1

Submitted on 5 Apr 2012 (v1), last revised 17 Apr 2012 (v2)

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Formal Verification of Synchronous Data-flow Compilers

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

RESEARCH REPORT
N° 7921
March 2012
Project-Team ESPRESSO

ISSN 0249-6399 ISRN INRIA/RR--7921--FR+ENG

Formal Verification of Synchronous Data-flow Compilers

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

Project-Team ESPRESSO

Research Report n° 7921 — March 2012 — 26 pages

Abstract: Synchronous data-flow languages have been used successfully for design and implementation of embedded and critical real-time systems. Synchronous language compilers compile programs to generate the executable code on particular platforms. To fulfill the high requirements of an efficient and reliable implementation, the correctness of the compilers must be guaranteed. This report aims at constructing a fully automated formal verification process to prove the correctness of a compiler for abstract clocks and clock relations (*temporal constraints*). We represent the source program and its compiled form (e.g. intermediate form, generated executable code) with *polynomial dynamical systems* and prove that the compiled form preserves the abstract clocks, and clock relations of the source program. In order to carry out this correctness proof, an appropriate relation called *refinement* and an automated proof method are provided. In this paper, the highly optimizing, industrial compiler from the widely used synchronous language SIGNAL to C is addressed.

Key-words: Formal Verification, Translation Validation, Validated Compiler, Code Generator, Synchronous Programs

RESEARCH CENTRE
RENNES – BRETAGNE ATLANTIQUE

Campus universitaire de Beaulieu
35042 Rennes Cedex

Vérification Formelle de Compilateurs Synchrones des Données de Flux

Résumé : Synchronous data-flow languages have been used successfully for design and implementation of embedded and critical real-time systems. Synchronous language compilers compile programs to generate the executable code on particular platforms. To fulfill the high requirements of an efficient and reliable implementation, the correctness of the compilers must be guaranteed. This report aims at constructing a fully automated formal verification process to prove the correctness of a compiler for abstract clocks and clock relations (*temporal constraints*). We represent the source program and its compiled form (e.g. intermediate form, generated executable code) with *polynomial dynamical systems* and prove that the compiled form preserves the abstract clocks, and clock relations of the source program. In order to carry out this correctness proof, an appropriate relation called *refinement* and an automated proof method are provided. In this paper, the highly optimizing, industrial compiler from the widely used synchronous language SIGNAL to C is addressed.

Mots-clés : Formal Verification, Translation Validation, Validated Compiler, Code Generator, Synchronous Programs

1 Introduction

In the synchronous approaches, synchronous data-flow languages such as LUSTRE [7] or SIGNAL [6] have been introduced and used successfully for the design and implementation of embedded and critical real-time systems. Programs are compiled to generate target codes for particular platforms. 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. program proof, model checking, etc). We expect that when a compiler does not claim bugs in the formally verified source code that it compiles to executable code, the generated executable code behaves as prescribed by the semantics of the source program. However, compilation is complex and compilers involve many stages where they perform transformations over the data structures of the source program. Some transformations are optimizations based on static analyses to eliminate inefficiencies, subexpressions in the code. Thus, bugs in the compilers can happen, causing 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 manual reviews of the generated code, turning all compiler optimizations off. But these techniques do not fully address the issues, and are expensive in terms of time and performance.

Our approach is to apply formal methods to the compiler itself in order to prove automatically that the semantics of the source program is preserved during the compilation. Multi-clocked synchronous data-flow programming languages describe computation modes without assuming any global clock in a system. This design model is referred as *polychronous* model [12]. The compilers of these languages, such as that we consider here, are composed of a sequence of code transformations and optimizations. The transformations and optimizations rewrite and translate the source code, in particular to eliminate inefficiencies portions. Some of the transformations are non-optimizing translations from the synchronous language or its intermediate language to another, lower-level language (e.g. C/C^{++}, Java code). Abstract clocks and clock relations (or *temporal constraints*) play the most important role in the system design and realization that represent all the control part: activation events of components and interactions between different components in the system. The control flow resulting from the analysis of abstract clocks and clock relations is used to derive an optimized data-flow for generating the target code. Thus it highly requires that the correct handing of these temporal constraints is ensured along the compilation of the compiler.

In this report, adopting the *translation validation* approach [20], we present an automated proof of the correctness of a synchronous data-flow compiler for temporal constraints during its compilation from a synchronous source program to its final sequential executable code. Our aim being to prove the correctness of C code generated from a fully clocked, scheduled, system of data-flow equations resulting from the aforementioned model transformations. Each individual transformation, optimization, or code generation phase of the compiler is followed by our verification process which proves the correctness of this step. The compiler will continue its work if and only if the correctness is proved positively. Our verification framework consists of the following:

- Using polynomial dynamical systems (PDS) over a finite field, as common semantics for both source program and compiled form.
- Formalizing the definition of *correct translation*, called *refinement* relation for the PDS models.
- A syntactic simulation-based proof which automatically reasons on the semantic preservation.

- A *counterexample-guided refinement* methodology which allows to automatically generate the refined model.
- The implementation is done within the existing model checker SIGALI in the Polychrony toolset [**?**].

The remainder of this report is organized as follows. Section 2 introduces the SIGNAL language features and the formal model of synchronous program behaviors. Section 3 presents a method to translate sequential C/C^{++} code to synchronous code, we consider here the translation from C code to SIGNAL program. In Section 4, we present our approach to formally verify the compilation and formalize the notion of "correct translation" by means of a refinement relation between PDSs. Section 5 addresses the application of our verification process to the highly optimizing, industrial compiler from the synchronous language SIGNAL, and the implementation which is integrated in the Polychrony toolset. Section 6 describes some related works, concludes our work and describes future work.

2 Preliminaries

2.1 An equational model of the synchronous program behavior

In this section, we will present an approach to model temporal behaviors, or abstract clocks and clock relations of a synchronous program. We are interested in capturing the data status which can be present or absent and the relationships between the data status.

We introduce some notations that will be used in this paper. 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, P_1 and P_2, we have:

- $Sol(P) \triangleq \{(z_1, ..., z_k) \in (\mathbb{Z}/p\mathbb{Z})^k | P(z_1, .., z_k) = 0\}$
- $Sol(P_1 * P_2) = Sol(P_1) \cup Sol(P_2)$ (union)
- $P_1 \oplus P_2 \triangleq (P_1^{p-1} + P_2^{p-1})^{p-1}$, then $Sol(P_1 \oplus P_2) = Sol(P_1) \cap Sol(P_2)$ (intersection)
- $\overline{P} \triangleq 1 - P^{p-1}$, then $(\mathbb{Z}/p\mathbb{Z})^k \backslash Sol(P) = Sol(\overline{P})$ (complementary)
- $P_1 \Rightarrow P_2 \triangleq \overline{P_1} * P_2$, then $Sol(P_1 \Rightarrow P_2) = \{Z \in (\mathbb{Z}/p\mathbb{Z})^k | P_1(Z) = 0 \Rightarrow P_2(Z) = 0\}$
- $\exists z_i P \triangleq P|_{z_i=1} * P|_{z_i=2} * ... * P|_{z_i=p}$
- $\forall z_i P \triangleq P|_{z_i=1} \oplus P|_{z_i=2} \oplus ... \oplus P|_{z_i=p}$

where $P|_{z_i=v}$ is P obtained by instantiating any occurrence of variable z_i by value v. Manipulations of polynomials over the finite field modulo p can be found in [11].

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 to define the presence or absence of the data in its stream. Streams and abstract clocks are related to functional constraints and temporal constraints, respectively. The structure of synchronous programs is usually described as a series of equational definitions. And 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. As we have mentioned above, we would like to cope with the semantics of abstract clocks and clock constraints of the data streams. 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 timed constraints (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 status of values in a data stream. For the Boolean data stream x, three possible status 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 need to encode 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: $x^2 = y^2$. Thus, it is obvious that the temporal semantics of a synchronous data-flow program can be modeled efficiently with a PDS whose coefficients range over $\mathbb{Z}/3\mathbb{Z}$. We use *state* variables to encode the operators that memorize past values of a data stream (e.g. SIGNAL *delay* operator). 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. Systems of polynomial equations characterize sets of solutions, which are states and *events* of programs. A system of equations

based method consists in manipulating the equation systems instead of the solution sets, avoiding the enumeration of the state space [10]. A PDS has no terminal state because a synchronous data-flow program takes infinite input data streams, thus for every state of its PDS there exists always the events to produce the next state.

Definition 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 $[Q_{0_1}, ..., Q_{0_n}]$.

2.2 Overview of the SIGNAL language features

In SIGNAL language [12], 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 $\mathcal{R}(x_1, ..., x_k)$. 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. 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 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 $\hat{y} = \hat{x} \cup \hat{z}$.

Table 1: Scheduling relation for primitives

$y := f(x_1, ..., x_n)$	$\hat{y} : x_1 \to y, ..., \hat{y} : x_n \to y$
$z := x\ default\ y$	$\hat{x} : x \to z, \hat{y} - \hat{x} : y \to z$
$z := x\ when\ y$	$\hat{z} : x \to z, \hat{z} : y \to \hat{z}$
$y := x\$1\ init\ y_0$	

- *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 $\hat{y}$.

The special event signal $\hat{0}$ denotes the null clock (the clock that is never present). Table 1 shows the scheduling relations associated with primitive equations of the SIGNAL language.

2.3 PDS model of SIGNAL programs

In order to model the temporal behavior of a SIGNAL program, each program individual equation is translated into some polynomial equations. 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 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. The delay operator \$ requires memorizing the past value of the signal, that is done by introducing a *state variable* ξ, where ξ stores the previous value of the signal and ξ' stores the current value of the signal. The following shows the translation of the primitive equations of the SIGNAL language. For instance, the SIGNAL primitive equation $z := x$ and y is represented by a system of two polynomial equations; $z = xy(xy - x - y - 1)$ and $x^2 = y^2$ whose coefficients range over $\mathbb{Z}/3\mathbb{Z}$.

- With Boolean signals:
  $y :=$ not $x : y = -x$
  $z := x$ and $y : z = xy(xy - x - y - 1); x^2 = y^2$
  $z := x$ or $y : z = xy(1 - x - y - xy); x^2 = y^2$

Table 2: Program *Merge* and its PDS model

`process Merge =`
`( ? boolean X;`
`! boolean Z)`
`(| Z := X default (not ZN)`
`| ZN := X$1 init true`
`| X ^= when ZN`
`|)`
`where`
`boolean ZN init true;`
`end;`
	`initialisation:`
	$\xi = 1$
	`evolution:`
	$\xi' =$
	$x + (1 - x^2) * \xi$
	`constraint:`
	$z = x - (1 - x^2)zn,$
	$zn = x^2\xi$,
	$x^2 = -zn - zn^2$

$z := x$ default $y : z = x + (1 - x^2)y$
$z := x$ when $y : z = x(-y - y^2)y$
$y := x\$1$ init $y_0 : \xi' = x + (1 - x^2)\xi; y = x^2\xi; \xi_0 = y_0$

- With Non-Boolean signals:
  $y := f(x_1, ..., x_n) : y^2 = x_1^2 = ... = x_n^2$
  $z := x$ default $y : z^2 = x^2 + y^2 - x^2y^2$
  $z := x$ when $y : z^2 = x^2(-y - y^2)$
  $y := x\$1$ init $y_0 : y^2 = x^2$

For example the simple SIGNAL program shown in Table 2 that specifies the output signal as the merge of the input signal X and its negative past value, is translated in the PDS model with variables x, z and zn corresponding to the boolean signals X, Z, ZN and a state variable ξ for the delay operator. Note that SIGNAL allows one to explicitly manipulate clocks through some derived constructs that can be rewritten in terms of primitive ones. For instance, $y :=$ when b is equivalent to $y := b$ when b.

3 Translating sequential code into synchronous program

3.1 SSA: an intermediate representation

Our methodology of translating sequential code (e.g. C/C^{++}) into synchronous program is via the use of the compiler GNU Compiler Collection (GCC) [5] to transform the sequential code into static single assignment (SSA) as an intermediate form. Then we apply a translation scheme to obtain a synchronous program from SSA as described in [8, 3].

SSA is a form of *Control Data Flow Graph (CDFG)* which is used as an intermediate representation for all compilation phases of GCC. It allows the compiler to do transformations and optimizations easily and efficiently. A CDFG is a directed graph whose vertices and edges represent the control flow nodes and the pass of control flow, respectively. There are three basic block types of control flows :

- *Basic blocks (B_i)*: the set of statements without jumps.
- *Test blocks (T_i)*: represent conditional branching expressions.
- *Join blocks (J_i)*: represent the results of test blocks. Every test node has a corresponding join successor node.

In SSA form, each variable receives exactly one assignment during it lifetime. Translating a program in CDFG form into SSA form is a two steps process.

- Some trivial ϕ-function are inserted at some of the join nodes in CDFG.
- New variables V_i are generated. Each mention of a variable V in the program is replaced by a mention of one of the new variables V_i.

The ϕ function is added to join blocks in order to choose the new variable value depending on the program control-flow. Its form at entrance to a node X is $V \longleftarrow \phi(R, S, ..)$, where $V, R, S, ..$ are variables. The number of operands is the number of control flow predecessors of X or the number of the predecessors of the join block. For example, $x_3 \longleftarrow \phi(x_1, x_2)$ means "x_3 takes the value x_1 when the flow comes from the block where x_1 is defined, and x_2 otherwise". The detailed transformations of C/C^{++} to SSA which are implemented in GCC are discussed in [14, 15].

Consider a typical C program whose CFDG has four basic blocks, one test block and one join block. Applying the two steps above give us the SSA form as follows.

B_0: $y_1 = a_1 * b_1$
$z_1 = a_1 * c_1$
B_1: `if` $(y_1 > z - 1)$ `goto` B_2
`else goto` B_3
B_2: $x_1 = y_1 - z_1$
B_3: $x_2 = z_1 - y_1$
B_4: $x_3 = \phi(x_1, x_2)$
B_5: `result =` $x_3 * a_1$

3.2 SSA to SIGNAL

In this section, we present a scheme to automatically transform SSA form into a synchronous program. To demonstrate it, we present a scheme to automatically transform SSA form into SIGNAL equations.

In general, we can consider SSA forms have the following syntax.

```
(program)
pgm ::= L:blk | pgm
(block)
blk ::= stm;blk | rtn
(instruction)
stm ::= x = f(y*)(function call)
   | x = φ(y*)(φ function)
   | if x goto L(conditional branching)
(return)
rtn ::= goto L (goto) | return (return)
```

The translation scheme is defined by induction on the syntax of a program in the SSA form. For each block of label L, we use an *input clock* x_L, an *immediate clock* x_L^{imm} and an *output clock* x_L^{exit} as boolean signals in the translated synchronous program, we denote the next value of x by x'. The clock x_L is set to be present if L is scheduled in the predecessor block (by emitting x'_L). The clock x_L^{imm} is set to be present to activate the block L immediately. The clock x_L^{exit} is et to true when the execution of the block labeled L terminates. The block is scheduled to execute if and only if the union of clocks x_L and x_L^{imm} is present. For some blocks such as test and join blocks, it is not needed to use all these clocks because their statements can be scheduled to execute when the output clocks of their predecessors are present.

For an instruction *stm*, a block *blk* labelled L and a program *pgm*, the functions $\mathcal{F}[[stm]]_L^{e_1} = \langle P\rangle^{e_2}, \mathcal{F}[[blk]]_L^{e_1} = \langle P\rangle^{e_2}$ and $\mathcal{F}[[pgm]]$ return a SIGNAL process P and the output clock e_2. These function take three arguments the instruction (or block), the label of the block, and an input clock e_1. The following describes a general rules of the translation scheme from SSA form to SIGNAL process. The notation $e \Rightarrow P$ means that if the clock e is present then proposition P holds.

$$
\begin{array}{lll}
(1) & \mathcal{F}[[L : blk; pgm]] & = \mathcal{F}[[blk]]_L^{x_L \vee x_L^{imm}} | \mathcal{F}[[pgm]] \\
(2) & \mathcal{F}[[stm; blk]]_L^e & = \mathcal{F}[[stm]]_L^e = \langle P\rangle^{e_1} | \mathcal{F}[[blk]]_L^{e_1} \\
(3) & \mathcal{F}[if\ x\ goto\ L_1]]_L^e & = \langle \mathcal{G}_L(L_1, e \wedge x)\rangle^{e \wedge \neg x} \\
(4) & \mathcal{F}[x = f(y*)]]_L^e & = \langle \mathcal{E}(f)(xy * e)\rangle^e \\
(5) & \mathcal{F}[[goto\ L_1]]_L^e & = (e \Rightarrow x_L^{exit} | \mathcal{G}_L(L_1, e)) \\
(6) & \mathcal{F}[[return]]_L^e & = (e \Rightarrow (x_L^{exit} | x_f^{exit}))
\end{array}
$$

where $\mathcal{G}_L(L_1, e) =$ if L_1 is after L in the control-flow then
$e \Rightarrow x_{L_1}^{imm}$ else $e \Rightarrow x'_{L_1}$.
$\mathcal{E}(f)(xy * e) = e \Rightarrow (\hat{x} | x = [[f]](y, z)), \forall fxyze.$

Our aim is to translate the C/C^{++} code generated by the compiler that usually consists of some actions of reading and writing data streams. However, in a synchronous data-flow program, it does not needed to represent any information about reading and writing data streams. Thus, it is not need to encode the statements of reading and writing in the C/C^{++} code. In addition, the pointer data type in the generated code is only used in reading and writing statements. As consequence, we will not mention a method to encode pointers, a solution of this problem can be found in [3].

4 Formal verification of synchronous compilers

4.1 Verification framework

We use the notion of translation validation of [**?**] as our verification process. In the old fashion, we have to prove in advance that the compiler always produces a target code which correctly implements the source code. Different from this way, each individual translation (i.e, an execution of the compiler) is followed by a verification phase which verifies that the target code produced in this execution correctly implements the submitted source program. The compiler verification is an extremely complex task. Moreover every change to the compiler requires to redo the proof. It is a main drawback of the proving in advance method. It is avoided with the translation validation method.

The translation validation gets the submitted program and the output of the compiler to analyze. If it finds that the output correctly implements the source program, it generates a detailed proof. Otherwise, it produces counterexamples. A counterexample consists of a scenario in which the output behaves differently than the source code. Thus, the counterexample provides an evidence that the compiler makes errors and needs to be fixed.

We consider here the main components of the formal verification framework which is depicted in Figure 3. These components are necessary to set up the fully automatic translation verification process for the translation steps of a synchronous compiler.

- A common semantic expression for the representation of the input source and the output programs. That makes the polynomial dynamical systems of both synchronous program and compiled form.
- A formalization of the notion of "correct translation" as a refinement relation, based on the common semantic expression PDSs.
- A proof method which allows to prove that the output program's PDS refines the input program's PDS (the output program correctly implements the submitted program).
- An *validation* which makes an automation of the proof method. If successful, it will generate a *proof*. Otherwise, it provides a counterexample in which the model of generated target code is faulty.

4.2 Definition of correct compilation: Refinement

Our aim is to verify that the abstract clocks and clock relations are preserved for every stage of a compiler. In order to say that the compiled form of a source program preserves the abstract clocks and clock relations of the source program, we propose a definition of correct compilation between two PDS models.

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) as defined in Definition 4.2:

Definition 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:

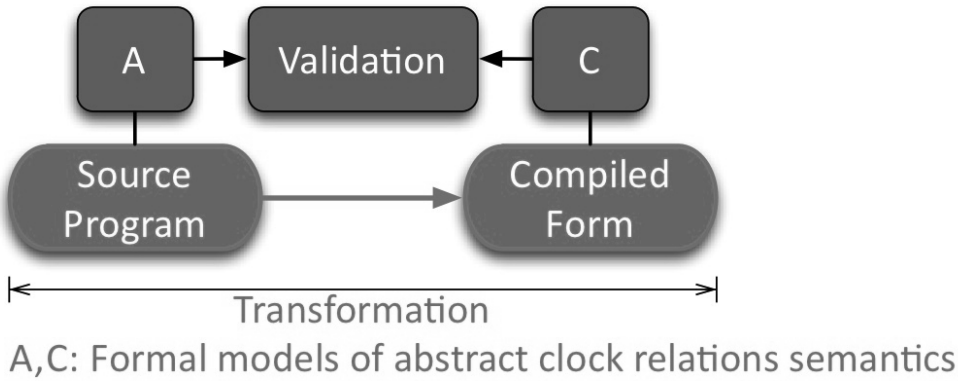

Figure 1: The components of formal verification process

- $Q = \mathcal{D}_X$, 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 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 $i \in \mathbb{N}$.

An action-based execution of σ is denoted by $\sigma_{act} = y_0, y_1, y_2, ...$ We denote by $||L||, ||L||_{act}$ 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* [19]. 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:

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

Requirement (1) is too strong in general to be practice 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 will present a method to establish the refinement between the two given models C and A.

4.3 Proving refinement by simulation

We now discuss an approach to automatically reason on the property that a compiler preserves semantics of the source program during its compilation, in the sense of the 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 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 $Sol((P \Rightarrow \prod_{i \in I} P_i)) = Sol(0) = (\mathbb{Z}/3\mathbb{Z})^m$. Thus $Sol(P) \subseteq Sol(\prod_{i \in I} P_i)$.

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 exists 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 4.3 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 talk about symbolic simulation, we imply talking 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 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 4.1 *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 [9].
(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 [21] 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 4.2 *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 4.2 is trivial with following Lemma 4.3.

Lemma 4.3 *Let C and A be two 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 (Lemma 4.3) Let $\sigma_1 = q_{0,1}, q_{1,1}, q_{2,1}, ...$ is an execution in C starting in $q_1 = q_{0,1}$ and assume $(q_1, q_2) \in \mathcal{R}$. We can define a corresponding execution in A starting in $q_2 = q_{0,2}$ with the same length (in case the execution σ_1 is finite), where the transitions $q_{i,1} \longrightarrow q_{i+1,1}$ are matched by transitions $q_{i,2} \longrightarrow q_{i+1,2}$ such that $(q_{i+1,1}, q_{i+1,2}) \in \mathcal{R}$. We use the induction method on i to prove it.

- Base case: i = 0. It follows directly from $(q_1, q_2) \in \mathcal{R}$ in case q_1 is a terminal state. If there is a transition $q_{0,1} \xrightarrow{P(Y)} q_{1,1}$ such that $y_{0,1} \in Sol(P(Y))$ then there exists a finite set of transitions $(q_{0,2} \xrightarrow{P_j} q^j_{1,2})_{j \in J}$ with $Sol(P) \subseteq Sol(\prod_{j \in J} P_j)$ and $(q_{1,1}, q^j_{1,2}) \in \mathcal{R}, \forall j \in J$. Because $Sol(P) \subseteq Sol(\prod_{j \in J} P_j)$, there exists a polynomial $P_j(Y)$ such that $y_{0,1} \in Sol(P_j)$, and the transition $q_{0,1} \xrightarrow{y_{0,1}} q_{1,1}$ can be matched by the transition $q_{0,2} \xrightarrow{y_{0,1}} q^j_{1,2}$ with $(q_{1,1}, q^j_{1,2}) \in \mathcal{R}$. This yields the execution fragment $q_{0,2}, y_{0,2}, q_{1,2}$ with $y_{0,1} = y_{0,2}$ in A.

- Induction step: Assume $i > 0$ and that the execution $q_2, y_{0,2}, q_{1,2}$, $y_{1,2}, q_{2,2}, y_{2,2}, ..., q_{i,2}$ is already constructed with $(q_{k,1}, q_{k,2}) \in \mathcal{R}$ and $y_{k,1} = y_{k,2}$ for $k = 0, ..., i$. If σ_1 has length i and $q_{i,1}$ is a terminal state, then the execution fragment $\sigma_2 = q_2, y_{0,2}, q_{1,2}, y_{1,2}, q_{2,2}$, $y_{2,2}, ..., q_{i,2}$ is an execution fragment with the same length which is state-wise related to σ_1. Now we assume that $s_{i,1}$ is not terminal. We consider the step $q_{i,1} \xrightarrow{P(Y)} q_{i+1,1}$ with $y_{i,1} \in Sol(P(Y))$ in σ_1. Since $(q_{i,1}, q_{i,2}) \in \mathcal{R}$, there exists a finite set of transitions $(q_{i,2} \xrightarrow{P_j} q^j_{i+1,2})_{j \in J}$ with $Sol(P) \subseteq Sol(\prod_{j \in J} P_j)$ and $(q_{i+1,1}, q^j_{i+1,2}) \in \mathcal{R}, \forall j \in J$. Because $Sol(P) \subseteq Sol(\prod_{j \in J} P_j)$, there exists a polynomial $P_j(Y)$ such that $y_{i,1} \in Sol(P_j)$, and the transition $q_{i,1} \xrightarrow{y_{i,1}} q_{i+1,1}$ can be matched by the transition $q_{i,2} \xrightarrow{y_{i,1}} q^j_{i+1,2}$ with $(q_{i+1,1}, q^j_{i+1,2}) \in \mathcal{R}$. This yields the execution fragment $q_2, y_{0,2}, q_{1,2}, y_{1,2}, ..., q_{i,2}, y_{i,2}, q_{i+1,2}$ with is state-wise related to the execution σ_1 and with $y_{i,1} = y_{i,2}$ in A.

With an unverified compiler of synchronous data-flow language, each compilation task 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:

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

where $Cp(A)$ is the compilation task from source program A to either compiled code (written as $Cp(A) =$ OK(C)) or compilation errors (written as $Cp(A) =$ Error).

4.4 Identification of counterexamples

Given two iLTSs $C = (Q_1, Y, \mathcal{I}_1, \mathcal{T}_1)$ and $A = (Q_2, Y, \mathcal{I}_2, \mathcal{T}_2)$, assume that symbolic simulation for them does not exist. Then there are some executions in C whose corresponding ones do not exist in A, or equivalently, C has some behaviors which do not exist in A. These executions are called *counterexamples*. In the rest of this section, we will focus on finding all counterexamples in C.

First, we will tackle the case when a sequence of states and events σ is a path $<q_1, y_1, ..., y_{n-1}, q_n>$, σ is an execution if it satisfies $q_1 \in \mathcal{I}_1 \wedge \bigwedge_{i=1}^{n-1} (q_{i+1} \in Post(q_i, P_i))$. Let

$$\begin{array}{rcl} t_{q_1} & = & \{q \in \mathcal{I}_2 | \ \exists q' \in Post(q, P(Y)) \text{ with } y_1 \in Sol(P)\} \\ t_{q_i} & = & \{q \in Post(t_{q_{i-1}}, P_{i-1}(Y)) \\ & & \text{with } y_{i-1} \in Sol(P_{i-1}(Y)) | \\ & & \exists q' \in Post(q, P(Y)) \ with \ y_i \in Sol(P)\} \end{array}$$

where:

$$\begin{array}{rcl} Post(q, P(Y)) & = & \{q' \in Q | q \xrightarrow{P(Y)} q'\} \\ Post(C, P(Y)) & = & \bigcup_{q \in C} Post(q, P(Y)) \text{ with } C \subseteq Q \end{array}$$

t_{q_i} represents the set of states such that they are direct successors with the event y_{i-1} whose direct events consist the event y_i. The following lemma establishes the correctness of finding counterexamples procedure.

Lemma 4.4 *The following statements are equivalent:*

- *The execution path σ is not a counterexample.*
- *For all $1 \leq i < n, t_{q_i} \neq \emptyset$.*

For example, consider two iLTSs $C = (Q_1, Y, \mathcal{I}_1, \mathcal{T}_1)$ and $A = (Q_2, Y, \mathcal{I}_2, \mathcal{T}_2)$ with $Q_1 = \{q_0, q_1, q_2, q_3, q_4\}, Q_2 = \{r_0, r_1, r_2, r_3\}$. Assume that the transitions between states in A and C are depicted in Figure 2. We assume that the sets of solutions of the polynomials are pairwise disjoint. It is easy to see that all the execution paths $\{<q_0, y_0, q_1, y_1, q_2, y_3, q_4> \mid \bigwedge_{i=0,1,3} (y_i \in Sol(P_i(Y)))\}$ are counterexamples. Following Lemma 4.4, we have $t_{q_0} = \{r_0\}$, $t_{q_1} = \{r_1\}$, $t_{q_2} = \emptyset$, thus they are counterexamples. Now, we consider the case when the execution σ consists in a loop

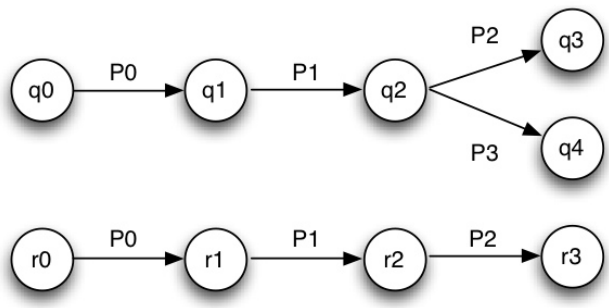

Figure 2: A counterexample

$<q_1, y_1, ..., q_l, y_l><q_{l+1}, ..., q_n, y_n>^{\omega}$. Let

$$\begin{array}{rcl} t_{q_n} & = & \{q \in Post(t_{q_{n-1}}, P_{n-1}(Y)) \\ & & \text{with } y_{n-1} \in Sol(P_{n-1}(Y)) | \\ & & \exists q' \in Post(q, P(Y)) \text{ with } y_n \in Sol(P)\} \\ t_{q_{n+1}} & = & \{q \in Post(t_{q_n}, P(Y)) \text{ with } y_n \in Sol(P)\} \\ t_{q_n loop} & = & t_{q_n} \\ t_{q_i loop} & = & \{q \in t_{q_i} | \, Post(q) \cap t_{q_{i+1} loop} \neq \emptyset\}, \\ & & l+1 \leq i \leq n-1 \end{array}$$

The following lemma holds.

Lemma 4.5 *The following statements are equivalent:*

- *The loop execution σ is not a counterexample.*
- *For all $1 \leq i \leq n, t_{q_i} \neq \emptyset$, and $t_{q_{l+1 loop}} \cap t_{q_{n+1}} \neq \emptyset$.*

4.5 Refining the generated code

Assume that following Theorem 4.1, we compute a greatest binary relation $\mathcal{R} \subseteq Q_1 \times Q_2$ such that.

- $\exists q_1 \in I_1, \forall q_2 \in I_2, (q_1, q_2) \notin \mathcal{R}$.
- $\forall (q_1, q_2) \in \mathcal{R}$ it holds that: if $q_1 \xrightarrow{\alpha} q_1', \exists q_2 \xrightarrow{\alpha} q_2' \wedge (q_1', q_2') \in \mathcal{R}$.

Obviously, $\mathcal{R}$ is not a simulation for (C, A). I_{1_r} denotes the set of states $q_1 \in I_1$ such that there does not exist any state $q_2 \in I_2$ which are similar. By Definition 4.3 if (q_1, P, q_1'), then for every states $q_2 \in I_2$, for all transitions from $q_2, (q_2, P_i, q_2^i)_{i \in I}$ (is finite since the system model is finite) with

- $(P \Rightarrow \prod_{i \in I} P_i) \not\equiv 0$ or
- $\exists i \in I$ such that $(q_1', q_2^i) \notin \mathcal{R}$.

To make the concrete model to be simulated by the abstract model, it is obvious to describe the needed refinement of I_{1_r} such that $\mathcal{R}$ is a simulation for (C, A). Given a state $q_1 \in Q_1$ and a set of states $S \subseteq Q_2$, we say that a state $q_2 \in S$ is the *most-similar* with the state q_1 if:

- for all states $q_i \in S, q_i \neq q_2$,
  $(Events(q_i) \cap Events(q_1)) \subseteq (Events(q_2) \cap Events(q_1))$.

To refine the concrete model, we refine the transitions and/or the successors of the state q_1 as the following:

- refine all direct transitions from q_1: if (q_1, P, q_1') and for all transitions from q_2, $q_2, (q_2, P_i, q_2^i)_{i \in I}$ with $(P \Rightarrow \prod_{i \in I} P_i) \not\equiv 0$, then the transition (q_1, P, q_1') is refined to (q_1, P_r, q_1') with $Sol(P_r) = Sol(P) \cap Events(q_2)$.
- refine all direct-successors states from q_1: if $q_1' \in Post(q_1)$, there exists a

4.6 Composition of compilation phases

Compilation is always decomposed into several stages of transformations, optimizations through intermediate representations. Thus, it is better decomposing 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 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:

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

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

5 Proving the SIGNAL compiler

5.1 Proving the compiler code generation

We will apply the verification approach that we have presented to the widely used compiler from the synchronous language SIGNAL. The compiler of the SIGNAL language [2] 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 phases (see Figure 3:

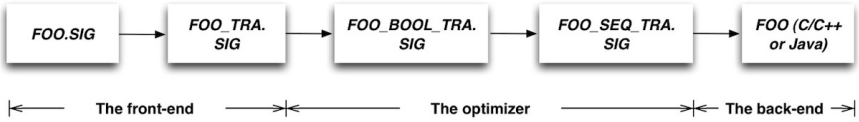

Figure 3: Scheme of the SIGNAL compiler

- *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 phase based on the DCG.
- *The back-end*: translations from the optimized final SIGNAL program to executable code (C/C^{++} or Java).

For instance, we consider a source program called "FOO.SIG", the transformations of the phases front-end, optimizer, back-end are "FOO_TRA.SIG", "FOO_BOOL_TRA.SIG" then"FOO_SEQ_TRA.SIG", and "FOO", respectively. The optimized final program "FOO_SEQ_TRA.SIG" is translated directly to executable code "FOO". 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 full compilation phases. For instance, we consider the compilation of "FOO.SIG" program, the verification asserts that $FOO \preceq FOO_SEQ_TRA.SIG \preceq FOO_BOOL_TRA.SIG \preceq FOO_TRA.SIG \preceq FOO.SIG$ along the transformations of the SIGNAL compiler.

5.2 Implementation of symbolic simulation with SIGALI

In this section, we discuss how to implement the proof method with symbolic simulation, and finding counterexamples for the two iLTSs of a source program and its compiled form. For

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**

that purpose, we use the companion model-checker of the Polychrony toolset, SIGALI. Symbolic simulation and finding counterexamples can be implemented as extended libraries 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 ($Sol(I(X))$),
- $\mathcal{T}(X, Y, X') \equiv Q(X, Y) \oplus (P(X, Y) - X')$ is the polynomial representing the set of transitions.

In SIGALI, polynomials are internally represented as *ternary decision diagrams* (TDD) [4] which are an extension of *binary decision diagrams* (BDD) [1]. They are convenient for an efficient manipulation of the polynomial equation systems. Theorem 4.1 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. The correctness of Algorithm 1 is proved by the following proposition.

Proposition 5.1 *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 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 4.1. Thus we omit it here.

Proposition 5.2 *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 5.1 and Theorem 4.1.

Algorithm 2 Path counterexample checking

Require: $T = <q_1, y_1, ..., y_{n-1}, q_n>$

$j := 1$
$t_q := t_{q_1}$
$T_{prev} := <q_1, y_1>$
while ($t_q \neq \emptyset$ and $j < n - 1$) **do**
  $j := j + 1$
  $t_q := t_{q_j}$
  $T_{prev} := <q_j, y_j>$
end while
if $t_q \neq \emptyset$ **then**
  return GOOD
else
  return (j, T_{prev})
end if

Algorithm 3 Loop counterexample checking

Require: $<q_1, y_1, ..., q_l, y_l><q_{l+1}, ..., q_n, y_n>^{\omega}$

$j := 1$
$t_q := t_{q_1}$
$T_{prev} := <q_1, y_1>$
while ($t_q \neq \emptyset$ and $j < n$) **do**
  $j := j + 1$
  $t_q := t_{q_j}$
  $T_{prev} := <q_j, y_j>$
end while
if $t_q \neq \emptyset$ **then**
  if $t_{qloop} \cap t_{q_{n+1}} \neq \emptyset$ **then**
    return GOOD
  else
    return $(j, <q_n, y_n>)$
  end if
else
  return (j, T_{prev})
end if

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				

Table 3: Experimental results

Following Lemma 4.4 and Lemma 4.5, we present Algorithm 2 and Algorithm 3 to check whether an execution (path or loop) is a counterexample. The algorithms can be efficiently implemented with the existing functions SIGALI.

5.3 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.

6 Related work and conclusions

The notion of translation validation was introduced in [20] 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 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 programs, the logic formula is asked to SAT-solver to solve is very large that makes some inefficiency. In addition, the whole calculation of a synchronous program or the generated code is considered as one atomic transition in STS, thus the STS does not capture the scheduling semantics of the programs. Another related work is the approach of J. C. Peralta et al. [18] 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.

The present paper provides a proof of correctness of the multi-clocked synchronous programming language compiler for temporal constraints and applies this approach to the highly industrial synchronous data-flow language SIGNAL compiler. We have proved that a synchronous data-flow compiler is correct if and only if the abstract clocks and the 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 full compilation phases of the optimizing SIGNAL compiler. Right now, we are working on extending the functionality of SIGALI for finding counterexamples in case there is not a simulation for the source program and its compiled form. Another perspective is to verify that the compiler compilation preserves both the temporal constraints and functional constraints.

References

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

[2] L. Besnard, T. Gautier, P. Le Guernic, and J-P. Talpin, *Compilation of polychronous data flow equations*, In Synthesis of Embedded Software, Springer, 2010.

[3] L. Besnard, T. Gautier, M. Moy, J-P. Talpin, K. Johnson, and F. Maraninchi, *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.

[4] B. Dutertre, M. Le Borgne, and H. Marchand, *SIGALI: un système de calcul formel pour la vérification de programmes SIGNAL*, Manuel d'utilisation. Note technique, non publiée, Dec. 1998.

[5] *Free Software Foundation*. The GNU compiler collection, http://gcc.gnu.org.

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

[7] 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.

[8] H. Kalla, J-P. Talpin, D. Berner, and L. Besnard, *Automated translation of C/C++ models into a synchronous formalism*, 13th IEEE International Symposium and Workshop on Engineering of Computer Based Systems, ECBS'06, 2006.

[9] O. Kouchnarenko, and S. Pinchinat, *Intensional approaches for symbolic methods*, In Electronic Notes in Theoretical Computer Science, Aug 1998.

[10] M. Le Borgne, *Systèmes dynamiques sur des corps finis*, Thèse, Université de Rennes I, Sept. 1993.

[11] M. Le Borgne, A. Benveniste, and P. Le Guernic, *Dynamical systems over Galois fields and control problems*, In Proceedings of 33th IEEE on Decision and Control, volume 3:1505-1509, 1991.

[12] 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.

[13] H. Marchand, H. Rutten, E. Le Borgne, and M. Samaan, *Formal verification of SIGNAL programs: Application to a power transformer station controller*, In Science of Computer Programming. 41(1):85-104, 2001.

[14] D. Novillo, *Tree ssa - a new high-level optimization framework for the gnu compiler collection*, Proceedings of the Nord/USENIX Users Conference, Feb. 2003.

[15] D. Novillo, *Design and implementation of tree-ssa*, In GCC Summit Proceedings, Ottawa, Canada, 2004.

[16] D. Park, *Concurrency and automata on infinite sequences*, In Proceedings of 5th GI Conf. on Th Comp. Sci. LNCS 104:167-183, Mar. 1981.

[17] *Polychrony Toolset*, http://www.irisa.fr/espresso/Polychrony/

[18] J. C. Peralta, T. Gautier, L. Besnard, and P. Le Guernic, *LTSs for translation validation of (multi-clocked) SIGNAL specifications*, In 8th IEEE/ACM International Conference on Formal Method and Models for Codesign. MEMOCODE, 2010.

[19] S. Pinchinat, H. Marchand, and M. Le Borgne, *Symbolic abstractions of automata and their application to the supervisory control problem*, In INRIA Technical Reports No 1279. pp.1-29, Nov. 1999.

[20] 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.

[21] R. Milner, *Operational and algebraic semantics of concurrent processes*, Research Report ECS-LFCS-88-46, Lab. for Foundations of Computer Science, Edinburgh, Feb. 1988.

[22] R. Milner, *A complete axiomatisation for observational congruence of finite-state behaviors*, In SIAM J. Comput. 81(2):227-247, 1989.

[23] R. De Simone, and A. Ressouche, *Compositional semantics of Esterel and verification by compositional reductions*, In Proceedings of CAV'94, LNCS 818, 1994.

[24] R. J. Van Glabbeek, *The linear time-branching time spectrum II: The semantics of sequential systems with silent moves (extended abstract)*, In CONCUR '93: 4th International Conference on Concurrent, volume 75:66-81, Mar 1993.

Table 4: Some useful functions of SIGALI's kernel

$declare(var, var) \rightarrow Lvar$	declaration of variables
$union_lvar(L1, L2) \rightarrow Lvar$	performs the union of variable lists
$diff_lvar(L1, 2) \rightarrow Lvar$	returns the sublist of $L1$ from which the variables of $L2$ have been removed
$implies(P1, P2) \rightarrow Poly$	$P1 \Rightarrow P2$
$complementary(P) \rightarrow Poly$	complementary of polynomial P
$intersection(P1, P2) \rightarrow Poly$	$P1 \oplus P2 \Leftrightarrow Sol(P) = Sol(P1) \cap Sol(P2)$
$union(P1, P2) \rightarrow Poly$	$P1 \otimes P2 \Leftrightarrow Sol(P) = Sol(P1) \cup Sol(P2)$
$exist(L, P) \rightarrow Poly$	existential elimination over the polynomial P w.r.t the variables of L
$forall(L, P) \rightarrow Poly$	universal elimination over the polynomial P w.r.t the variables of L
$rename(P, L1, L2) \rightarrow Poly$	renaming of the variables $L1$ by the variables $L2$ in P
$loop\, x = f(x)\, init\, x_0$	calculating the fixpoint of function x with the initial value x_0

Appendix A

Here we provide an implementation of Algorithm 1 to compute the greatest simulation for two intensional labeled transition systems. The implementation uses some useful functions from the kernel of SIGALI (Table 4). The inputs are the concrete and abstract intensional labeled transition systems *S_I1* and *S_I2*, respectively.

```
%*******************************************%
% This lib computes symbolic simulation     %
% relation between two polynomial dynamical %
% systems based on the implicit transition  %
% system representations of them.           %
% Input: S_I1, S_I2                         %
% Output: ilts_simulation(S_I1,S_I2)        %
%*******************************************%

% internal function P1 ==> P2 %
def implies(P1,P2) :
   union(complementary(P1),
      P2);

def states_simulation(X_1,Y_1,X_1_nexts,Rel_1,
                         X_2,Y_2,X_2_nexts,Rel_2) :
   with
      % define utility variables %
      Y_1_bar = diff_lvar(Y_1,Y_2),
      Y_2_bar = diff_lvar(Y_2,Y_1),
      Y = diff_lvar(Y_1,Y_1_bar),
      X_1_X_2 = union_lvar(X_1,X_2),
      X_1_nexts_Y_1_bar = union_lvar(X_1_nexts,Y_1_bar),
      X_2_nexts_Y_2_bar = union_lvar(X_2_nexts,Y_2_bar),
      X_1_X_2_nexts = union_lvar(X_1_nexts,X_2_nexts)
```

```
   do

   % compute the simulation %
   loop x =
    intersection(x,
         forall(forall(exist(implies(Rel_1,
          exist(intersection(Rel_2,
           rename(x,X_1_X_2,X_1_X_2_nexts)),
            X_2_nexts_Y_2_bar)),Y_1_bar),Y),X_1_nexts))
    init 0;


def ilts_simulation(S_I1,S_I2) :
   with
      % get the components of iLTS1 %
      I_1 = initial_I(S_I1),
      X_1 = state_var_I(S_I1),
      X_1_nexts = state_var_next_I(S_I1),
      Y_1 = event_var_I(S_I1),
      Rel_1 = trans_rel_I(S_I1),

      % get the components of iLTS2 %
      I_2 = initial_I(S_I2),
      X_2_d = state_var_I(S_I2),
      X_2_nexts_d = state_var_next_I(S_I2),
      Y_2 = event_var_I(S_I2),
      Rel_2_d = trans_rel_I(S_I2),
      % rename the states variables %
      X_2 = declare_suff(X_2_d),
      X_2_nexts = declare_suff(X_2_nexts_d),
      Rel_2 = rename(Rel_2_d,union_lvar(X_2_d,X_2_nexts_d),
                         union_lvar(X_2,X_2_nexts)),
      states_sim = states_simulation(X_1,Y_1,X_1_nexts,
                   Rel_1,X_2,Y_2,X_2_nexts,Rel_2)
   do

   % compute the systems simulation %
   intersection(states_sim,
                forall(implies(I_1,exist(intersection(
                       states_sim,I_2),X_2)),X_1));
```

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