Formal verification of automatically generated C-code from polychronous data-flow equations

Van Chan Ngo, Jean-Pierre Talpin,
Thierry Gautier, and Paul Le Guernic
INRIA, Campus Universitaire de Beaulieu,
35042 Rennes cedex, France
{Chan.Ngo, Jean-Pierre.Talpin, Thierry.Gautier,
Paul.LeGuernic}@inria.fr

Loïc Besnard
IRISA/CNRS, Campus Universitaire de Beaulieu,
35042 Rennes cedex, France
Loic.Besnard@irisa.fr

***Abstract*—Synchronous data-flow languages are used as design approaches in developing embedded and critical real-time systems in which synchronous programs are verified by applying formal verification. In a synchronous design approach, transformation and optimization are used to transform synchronous programs and generate general purpose executable code. The incorrectness of the transformations make the guarantees unable to carry over the transformed programs and the executable code. In this work, adopting the translation validation approach, we present an automated verification process to verify the correctness of the synchronous language compiler SIGNAL transformations and code generation on the clock information.**

***Keywords*—Formal verification, Translation validation, Validated compiler, Code generator, Synchronous programs.**

I. INTRODUCTION

Adhering to the synchronous paradigm, synchronous data-flow languages such as LUSTRE [10] or SIGNAL [8] have been introduced and successfully used to design and implement embedded software architectures and critical real-time systems. For critical application high-assurance systems, reliability in code generated from these tools is tantamount: generated code shall behave as prescribed by the semantics of the source specification (e.g. program proofs, model checking, traceability and qualification, etc). However, and before code can be generated, the compilation of high-level, synchronous, specification is a complex process that involves many analysis and program transformation stages. Some transformations may introduce additional informations or constraints, to refine the meaning of the original specification and/or remove or specialize the behavior of the source specification, like static scheduling. Thus, and even if compliant with a "five-nines" (99.999%) reliability, large-scale use of high-TRL compilers for large specifications may improbably yet not uncertainly yield bugs. One, nonetheless, expects the formally verified behavior of the source specification to be preserved in the code automatically generated from it and naturally requires it to be formally checked as well.

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

In this aim, translation validation was introduced in the 90's by Pnueli et al. as a technique to formally verify the correctness of code generated from the data-flow synchronous language SIGNAL using model checking. Rather than certifying the code generator (by writing it entirely using a theorem prover) or qualifying it (by obeying to the 27 documentations required as per the DO-178C) translation validation provides a scalable approach to assessing the functional correctness of automatically generated code. By revisiting transition validation, which in the 90's suffered from the limitations of theorem proving and model checking technologies available then, we aim at developing a scalable and flexible approach that applies to an existing 500k-lines implementation of SIGNAL, POLYCHRONY, and is capable of handling large-scale, possibly automatically generated, SIGNAL programs, while using of-the-shelf, efficient, model-checkers and SAT/SMT-solving libraries.

Our approach is to apply formal methods to the compiler transformations itself in order to automatically generate formal evidence that the semantics of the source program is preserved during program transformation and compilation, as per applicable qualification standards (DO-178). Moreover, and on the contrary to previous or related approaches, our aim is to provide means for implementing this approach in a scalable way which, unlike specifications of SMT-solving techniques within theorem provers, uses modern model checking tools or efficient SMT libraries to achieve the expected goals: traceability and formal evidence.

In this paper, we adopt *translation validation* [22] to provide an automated correctness proof of a synchronous data-flow source specification with respect its generated sequential executable code. In [14], we proved that controllers synthesized from logical timing constraints, consisting of ternary clock relations, could be checked correct in all states of the system under verification by using model checking techniques.

The present paper continues this prospect and presents

the verification method applied to code generation stages. It addresses the formal verification of the generated C-code from a refined and optimized intermediate specification in which a controller enforces logical timing constraints and in which the execution-order of data-flow equations is completely scheduled. As a result, all individual transformations, optimizations, and code generation phases of the compiler are followed by a verification step which proves the correctness of transformations. The compiler continues if and only if correctness is proved and returns an error and trace otherwise.

The remainder of this paper is organized as follows. Section II introduces the formal model of our verification tool and the automatic translation from SIGNAL specifications into it. Section III presents a method to translate generated sequential C code back into SIGNAL using SSA decomposition. In Section IV, we define the refinement relation which formally proves conformance between the original specification and that reverse-engineered from its generated code. Section V addresses the application of the verification process to the SIGNAL compiler, and its implementation integrated in the POLYCHRONY toolset. Section VI presents some related works, concludes our work and outlines future directions.

II. AN EQUATIONAL MODEL OF SYNCHRONOUS PROGRAMS

A. Overview of the SIGNAL language features

In SIGNAL language [9], 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 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 $\hat{y} = \hat{x} \cup \hat{z}$.
- *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 language allows clock constraints to be defined *explicitly* by equations. The signal's clock is represented by a special signal of type *event* which carries only a single value $true$. 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 $\hat{}y$.

B. An equational model of the synchronous program behavior

In this section, we will present an approach to model the clock semantics of a synchronous program. The clock semantics consists of the clocks of data-flows in which define the status of the data-flows (present or absent) and the explicit and/or implicit clock relations in the program.

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 $P_1(Z), P_2(Z), P(Z) \in \mathbb{Z}/p\mathbb{Z}[Z]$, we denote

- $Sol(P) \triangleq \{(z_1, ..., z_k) \in (\mathbb{Z}/p\mathbb{Z})^k | P(z_1, .., z_k) = 0\}$
- $Sol(P_1 * P_2) = Sol(P_1) \cup Sol(P_2)$ (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. The manipulations of polynomials over the finite field modulo p can be found in [3].

Synchronous data-flow languages represent data as an infinite sequence of values called data-flow, and each data-flow is combined with an associated abstract clock as a means of discrete time to define the presence or absence of the data in its data-flow. 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 clock semantics. 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 data-flow (abstract clock) and the timed constraints (clock relations). The principle is to encode the status of a value in a data-flow 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-flow. For the Boolean data-flow 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-flows, it only need to encode the fact that the value is present or absent (the clock value of the data-flow is *true* or *false*): $present \rightarrow \pm 1; absent \rightarrow 0$. And the clock of a data-flow is the square $x^2 : 1\ if\ present,\ 0\ if\ absent$. Thus, two synchronous data-flows (they have the same clock) x and y satisfy the constraint equation: $x^2 = y^2$. It is obvious that the clock semantics of a synchronous data-flow program can be modeled efficiently with a PDS whose coefficients range over $\mathbb{Z}/3\mathbb{Z}$. We introduce state variables to encode the operators that memorize the past values of a data-flow (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-flows 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 [2]. A PDS has no terminal state because a synchronous data-flow program takes infinite input data streams, thus for every state, there exists always the events to produce the next state.

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

TABLE I
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:
$z = x - (1 - x^2)zn,$
$zn = x^2\xi,$
$x^2 = -zn - zn^2$

C. PDS *model of* SIGNAL *programs*

In order to model the clock semantics 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 language. For instance, the 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 $y : z = xy(1 - x - y - xy); x^2 = y^2$
  $z := x$ 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 I 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.

III. Translating sequential code into synchronous program

A. 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) [7] 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 [5], [11].

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 [15], [16].

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: $y_1 = a_1 * b_1$
$z_1 = a_1 * c_1$
T_1: `if` $(y_1 > z_1 - 1)$ `goto` B_2
`else goto` B_3
B_2: $x_1 = y_1 - z_1$
B_3: $x_2 = z_1 - y_1$
J_4: $x_3 = \phi(x_1, x_2)$
B_5: `result =` $x_3 * a_1$

B. 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 $e \Rightarrow x'_{L_1}$.
$\mathcal{E}(f)(xy * e) = e \Rightarrow (\hat{x} | x = [[f]](y, z)), \forall f x y z e.$

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 [5].

IV. Formal verification of synchronous compilers

A. Definition of correct transformation: Refinement

Our aim is to verify formally that the clock semantics are preserved for every stage of a compiler. In order to do that, we propose a formal definition of correct transformation between two PDS models. Given a PDS model over the finite field $\mathbb{Z}/3\mathbb{Z}$, it can be represented as an *intensional Labeled Transition System* (iLTS) as defined in Definition 2:

Definition 2: An iLTS is a structure $L = (Q, Y, \mathcal{I}, \mathcal{T})$, where Q is a set of states, Y is a set of event variables, $\mathcal{I}$ 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 sets of state variables, event variables, systems of initial equations, evolution equations, and constraint equations with $Q = \mathcal{D}_X = \prod_{i\in[1,n]} \mathcal{D}_{x_i} = (\mathbb{Z}/3\mathbb{Z})^n$ as the domain of a set of variables $X = (x_1, ..., x_n)$; $Y = (y_1, ..., y_m)$; $\mathcal{I} = Sol(Q_0(X))$; $(q, P_q(Y), q') \in \mathcal{T}$ 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: The infinite sequence $\sigma = q_0, y_0, q_1, y_1, ...$ where $q_i \in Q, y_i \in \mathcal{D}_Y$ for each $i \in \mathbb{N}$, is an *execution* if $q_0 \in \mathcal{I}$ and 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 $i \in \mathbb{N}$. $\sigma_{act} = y_0, y_1, ...$ is called action-based execution of the execution σ.

We denote by $||L||, ||L||_{act}$ the sets of all executions and action-based executions of an 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* [20]. The clock semantics are the event values and they are represented by action-based executions of the corresponding iLTS. Therefore, we say that A and C have the same clock semantics if:

$$\begin{aligned} &\forall \sigma_{act}.((\sigma_{act} \in ||C||_{act} \Rightarrow \sigma_{act} \in ||A||_{act}) \\ \wedge\ &(\sigma_{act} \in ||A||_{act} \Rightarrow \sigma_{act} \in ||C||_{act})) \end{aligned} \quad (1)$$

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. 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:

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

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. 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:

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

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).

B. Proving refinement by simulation

We now discuss an approach to check the existing of refinement by using simulation techniques. We will show that if there exists a *symbolic simulation* for C and A as defined in Definition 4 then $C \sqsubseteq A$.

Definition 4: 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 $Sol(P) \subseteq Sol(\prod_{i\in I} P_i)$ and $(q_1', q_2^i) \in \mathcal{R}, \forall i \in I$.

Condition (A) asserts that every initial state of C is related to an initial state of A. According to condition (B), for every transition from the state q_1 which is labeled by the set of events represented by $Sol(P(Y))$, there exists some transitions from the state q_2 which are labeled by the same set of events. Since symbolic simulation is closed under arbitrary unions, so there is a greatest symbolic simulation. And we say that C is simulated by A (or, equivalently, A simulates C), denoted $C \preceq A$. Two states q_1 and q_2 are *similar*, denoted $q_1 \preceq q_2$, if there exists a symbolic simulation $\mathcal{R}$ with $(q_1, q_2) \in \mathcal{R}$. 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 as a greatest fixed point.

Theorem 1: Let C and A 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: Due to the lack of space, we present the proof in Appendix A. ∎

The use of a symbolic simulation as a proof method to check the existing of refinement between C and A is stated in the following theorem.

Theorem 2: If there exists a symbolic simulation for (C, A), then $C \sqsubseteq A$.

The proof of Theorem 2 is trivial with following Lemma 3.

Lemma 3: $\mathcal{R}$ is a symbolic simulation for (C, A), and $(q_1, q_2) \in \mathcal{R}$. Then for each infinite (or finite) execution $\sigma_1 = q_{0,1}, y_{0,1}, q_{1,1}, y_{1,1}, ...$ starting in $q_{0,1} = q_1$ there exists an execution $\sigma_2 = q_{0,2}, y_{0,2}, q_{1,2}, y_{1,2}, ...$ from state $q_{0,2} = q_2$ with 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 present the proof in Appendix A. ∎

Due to the transitive property of symbolic simulation our verification process can be decomposed well. 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).

C. Identification of counterexamples

Assume that symbolic simulation for (C, A) does not exist, we will find the set of states along with their associated events which can be used to construct counterexamples in C, *CounterPositions* $\subseteq Q_1 \times Y$. The *CounterPositions* can be computed along with the the relation $\overline{\mathcal{R}} \subseteq Q_1 \times Q_2$ which satisfies the following properties:

- for any $(q_1, q_2) \in \overline{\mathcal{R}}$ it holds that: if $q_1 \xrightarrow{P} q_1'$ then for any transition $(q_2 \xrightarrow{P_i} q_2^i)$ with $Sol(P \oplus P_i) \neq \emptyset$, it has $(q_1', q_2^i) \in \overline{\mathcal{R}}$.
- and $(q_1, y) \in$ *CounterPositions*, $\forall y \in Sol(P \oplus P_i)$

The correctness of this computation follows from the fact that the symbolic simulation relation between C and A is a greatest fixed point and $\overline{\mathcal{R}}$ is the complement of the symbolic simulation. Given a *CounterPositions*, every counterexamples $\sigma = q_0, y_0, q_1, y_1, ...$ can be constructed from the *CounterPositions* by getting $(q_i, y_i) \in$ *CounterPositions*, $i = 0, 1, 2, ...$ such that $q_0 \in \mathcal{I}_1$ and q_i is successor of q_{i-1}.

V. Proving the Signal compiler

A. Proving the compiler code generation

In this section, we will apply the translation validation approach that we have presented to the widely used compiler from the synchronous language SIGNAL. This compiler [4] consists 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 programs to SIGNAL programs or executable code. For convenience, the transformations of the compiler are classed into three phases as depicted in Figure 1. The validator asserts that *.C $\preceq$ *_SEQ_TRA.SIG $\preceq$ *_BOOL_TRA.SIG $\preceq$ *_TRA.SIG $\preceq$ *.SIG along the compiler transformations.

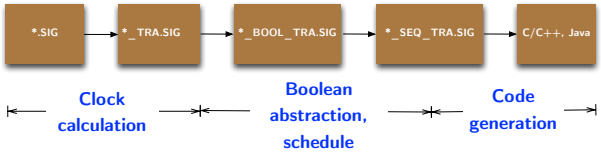

Fig. 1. Scheme of the SIGNAL compiler

B. Implementation of symbolic simulation with SIGALI

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: $\quad \mathcal{R}_{j+1}(X_1, X_2) \equiv \mathcal{R}_j(X_1, X_2) \oplus$
4: $\quad \forall X_1' \forall Y [(\mathcal{T}_1(X_1, Y, X_1') \rightarrow \exists X_2' (\mathcal{T}_2(X_2, Y, X_2') \oplus \mathcal{R}_j(X_1', X_2'))]$
5: **end while**
6: **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**
7: $\quad$ **return** $\mathcal{R}(X_1, X_2)$
8: **else**
9: $\quad$ **return** $\mathcal{R}(X_1, X_2) \equiv 1$
10: **end if**

We will discuss how to implement the validator by using the companion model-checker of the POLYCHRONY toolset, SIGALI in which symbolic simulation computation and identification of counterexamples can be implemented as extended libraries. We use polynomials to represent 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 corresponding PDS; $\mathcal{I}(X) = Q_0(X)$ is polynomial representing the set of initial states $Sol(Q_0)$; $\mathcal{T}(X, Y, X') \equiv Q(X, Y) \oplus (P(X, Y) - X')$ is polynomial representing the transition relation.

Polynomials are internally represented as *Ternary Decision Diagrams* (TDD) (an extension of *Binary Decision diagrams* (BDD) [1], [6]) that are convenient for an efficient manipulation of polynomial equation systems. Theorem 1 gives us an

Name	nX	nY
EquationSolving.z3z	1	8
EquationSolving_SEG_TRA.z3z	1	10
Oscillo.z3z	5	9
Oscillo_SEG_TRA.z3z	5	10
VTAlarm.z3z	19	45
VTAlarm_SEG_TRA.z3z	19	53
VTChronometer.z3z	6	33
VTChronometer_SEG_TRA.z3z	6	37
VTSuper.z3z	5	21
VTSuper_SEG_TRA.z3z	5	28

PDS nodes, time(s)	$iLTS$ nodes, time(s)	$\mathcal{R}(X_1, X_2)$ nodes, time(s)
Small, 0.00 Small, 0.00	Small, 0.00 Small, 0.00	Small, 0.00
Small, 0.00 Small, 0.00	Small, 0.00 Small, 0.00	Small, 0.00
Small, 0.00 Small, 0.00	Small, 0.00 Small, 0.00	Small, 0.00
Small, 0.02 396652, 119.84	140562, 0.67 2191292, 821.20	3810301, 14965.82
Small, 0.00 Small, 0.00	Small, 0.00 Small, 0.01	Small, 0.01

TABLE II
EXPERIMENTAL RESULTS

iterative algorithm to compute the greatest symbolic simulation. 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 SIGALI fixed point computation. The implementation of identification of counterexamples is done in the same manner. The correctness of this algorithm is proved by the following propositions.

Proposition 4: For all $j \in \mathbb{N}, \mathcal{R}_j(x_1, x_2) = 0$ if and only if $x_1 \preceq_j x_2$.

Proposition 5: Algorithm 1 terminates and at the end, $\mathcal{R}(x_1, x_2) = 0$ if and only if $x_1 \preceq x_2$.

C. Experimental results

In Table II, we provide some experimental results computing symbolic simulation. The experimental results deal with the complexity of computation of PDS, iLTS, and symbolic simulation. We obtain a description complexity of the computations by the number of TDD nodes that we need to represent the manipulation of polynomial equations. For instance, we consider a PDS model of program A.SIG in the text form A.Z3Z and its compiled form A_SEG_TRA.SIG in the text form A_SEG_TRA.Z3Z. In the nX, nY columns, we write the numbers of state variables and event variables, respectively. Hence, the polynomial equations $Q_0(X), Q(X, Y)$, and $X' = P(X, Y)$ have $nX, nX + nY$, and $2nX + nY$ variables. The polynomial equations $\mathcal{I}(X)$, and $\mathcal{T}(X, Y, X')$ of the iLTS have nX and $2nX + nY$ variables, respectively. And the symbolic simulation $\mathcal{R}(X_1, X_2)$ has $nX_1 + nX_2$ variables, where $\mathcal{R}(X_1, X_2)$ is the symbolic simulation for (A.Z3Z,A_SEG_TRA.Z3Z). In our implementation, in the PDS column, we write the number of TDD nodes to represent the polynomial equations $Q_0(X), Q(X, Y)$, and $X' = P(X, Y)$ of the PDS. In the iLTS column, the number of TDD nodes is the number of nodes to represent the polynomial equations of the PDS and the polynomial equations of the iLTS. And the number of TDD nodes in the $\mathcal{R}(X_1, X_2)$ column is the number of nodes to represent the polynomial equations of the PDS, the iLTS, and the symbolic simulation. The number of TDD nodes is showed only when it is big enough, thus for the tests whose numbers of TDD nodes are not showed we write "Small" instead. All the examples here are available in the online examples of the POLYCHRONY toolset. The tests were run on a virtual machine with one core processor 2.66 GHz, 1 Gb of physical memory, under Ubuntu-Linux 10.04.

VI. RELATED WORK AND CONCLUSIONS

The notion of translation validation was introduced in [21], [22] by A. Pnueli et al. to verify the code generator of SIGNAL. In that work, the authors define a language of symbolic models to represent both the source and target programs called *Synchronous Transition Systems (*STS*)*. A STS is a set of logic formulas which describe the functional and temporal constraints of the whole program and its generated C code. Then they use BDD representations to implement the symbolic models STSs, and their proof method uses a SAT-solver to reason on the signal constraints. 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 it does not capture explicitly the clock semantics and in some cases, the code generator eliminates the use of a local register variable in the generated code and then, the mapping cannot be established. Additionally, for a large 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 it does not capture the scheduling semantics of the programs. Another related work is the approach of J. C. Peralta et al. [19] in which is based on translation validation approach. In particular, they translate both the SIGNAL (multi-clocked) specifications 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 specification. The refinement they propose is a bisimulation relation and they use the existing tools to generate the greatest bisimulation relation for the 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 clock semantics preservation and applies this approach to the 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 translation validation 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 compiler. As future work, to deal with synchronous programs with large number of variables we intend to represent clock information as formulas over Boolean variables, then use a SMT-solver to check that the clock model of compiled program is a model of the source program. Another perspective is to use *Synchronous Data-flow Dependency Graph (*SDDG*)* to represent the dependency semantics (or schedule semantics) of synchronous programs and verify that the compiler compilation preserves the data dependency semantics.

REFERENCES

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

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

[3] 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.

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

[5] 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.

[6] 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.

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

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

[9] 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.

[10] 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.

[11] 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.

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

[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] V. C. Ngo, J-P. Talpin, T. Gautier, P. Le Guernic, and L. Besnard, *Formal verification of compiler transformations on polychronous equations*, In Processdings of IFM'12, June. 2012.

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

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

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

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

[19] 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.

[20] 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.

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

[22] 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.

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

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

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

[26] 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.

APPENDIX A

Proof Theorem 1: (1) The proof can be found in [12]. (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 [23] for strong simulation. We omit the proof here. ■

Proof Lemma 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.

■

Proof Proposition 4: $\Rightarrow$) 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 $\{y | \exists x_2 \xrightarrow{P_i} x^i_2, P_i(y) = 0 \wedge x'_1 \preceq_j x^i_2\}$. 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. ■

Proof Proposition 5: Termination is guaranteed by the fact that relations $\mathcal{R}_j$ are finite and nested. The second statement is a corollary of Proposition 4 and Theorem 1. ■