Translation Validation of Code Generation from the SIGNAL Data-Flow Language to Verilog

Hafiz Muhammad Amjad[1], Kai Hu[1], Jianwei Niu[1], Noor Khan[1], Loïc Besnard[2], Jean-Pierre Talpin[3]

[1]*School of Computer Science & Engineering, Beihang University, Beijing, China*

[1]{amjadphool, hukai, niujianwei, noorkhan_92}@buaa.edu.cn

[2]*IRISA/CNRS, Campus de Beaulieu, Rennes, France*

[2]Loic.Besnard@irisa.fr

[3]*INRIA, Campus de Beaulieu, Rennes, France*

[3]jean-pierre.talpin@inria.fr

Abstract— **The SIGNAL is a high-level synchronous data-flow language for the design and implementation of safety-critical embedded systems. It provides a unified framework for specification, modeling, formal analysis, and automatic code generation for different general-purpose languages like Java, C, and C++. However, fully implemented and verified open source tool for code generation from SIGNAL to Hardware Description Language (HDL) is not available. This paper describes the formal verification of the generated Verilog code from the SIGNAL language. Proving the correctness of generated code is very important when it is for safety-critical embedded systems. We use the translation validation technique for verifying the correctness of the generated code. In this approach, the Polychrony Toolset builds the models of source SIGNAL programs with its associated model checker SIGALI. The open source tool Yosys generates models for target Verilog programs in the SMT-LIB standard format. We transform the model generated by Yosys to the model accepted by the SIGALI model checker. Finally, we use the SIGALI model checker to validate the translation by symbolic simulation between both source and target program models. The target program may have fewer behaviors than the source program therefore if the model of the target program implies the model of the source program, it means the target program preserves the semantics of the source program, and the translation is correct.**

Keywords—translation validation, embedded systems, Verilog, SIGNAL, SIGALI, Yosys, semantics

I. INTRODUCTION

The functionally correct and quick development of safety-critical embedded systems emphasizes the need for a common framework of specification, formal analysis, and implementation. Model-Driven Engineering (MDE) technologies offer a model-based approach and support automated development by raising the level of abstraction. MDE tools utilize model-checking techniques to indicate various subtle bugs early in the design phase [1]. Synchronous languages like LUSTRE [2], ESTEREL [3], and SIGNAL [4] provide a promising option for the design and development of reactive safety-critical embedded systems by offering formal analysis. These synchronous languages use synchrony hypothesis [5] (input, computation, communication, and output take no time at a discrete logical instant) and manage all the synchronizations during compilation [6].

The SIGNAL language has a multi-clocked and synchronous (polychronous) Model of Computation (MoC), thus permitting to describe globally asynchronous, locally synchronous (GALS) systems [7]. A SIGNAL program consists of equations which describe the relations among typed (e.g., Boolean, Integer, Real) signal at logical instants. The open source Polychrony Toolset [8] contains the SIGNAL language compiler together with other tools and interfaces. Before generating target code, the source SIGNAL programs go through several transformations in intermediate stages where the compiler ensures synchronizations and preservation of semantics. Currently, the available version of Polychrony provides services to generate code in general-purpose languages like C, C++, and Java. However, the verified and fully implemented Hardware Description Languages (HDL) code generation facility is not available in the Polychrony Toolset. Sometimes, the designer desires to implement some part or an entire design in hardware for energy-efficient and high-performance applications. We aim at developing such a tool for HDL Verilog code generation from SIGNAL [9] and at validating the correctness of generated code by translation validation. During the compilation from high-level SIGNAL to HDL Verilog code, the SIGNAL compiler performs some optimizations (e.g., dead-code removal and elimination of common sub-expressions) while ensuring synchronizations consistency and freedom from deadlock. Therefore, it enables an efficient and reliable automatic Verilog code generation as compared to writing directly in Verilog.

[10], [11], [12], [13], and [14] describe VHDL code generation from SIGNAL. In [10], the author presents that it is possible to generate behavioral, structural, or RTL level VHDL code from SIGNAL. In [11], the author mentions some limitations that not all the transformations from SIGNAL to VHDL are possible due to different data types. In [12], the author proposes a Hierarchical Conditional Dependency Graph (HCDG) as a unifying approach for high-level synthesis. In [13], the author presents the method for synthesis of GALS architectures. In [14], the author illustrates the possibility of generating RTL level HDL code from SIGNAL and verification of the functional equivalence of program transformations. We are working on a formally verified code generation tool from SIGNAL to Verilog [9].

We are using translation validation technique proposed by A. Pnueli et al. in [15]. The other formal approach for verification of translation is proving in advance that the compiler always produces a correct target code for the submitted source code. The main advantage of translation validation approach is to avoid the requirement for re-implementing and then re-proving the entire compiler after every revision or modification. In translation validation, after each translation phase, the validation phase ensures that the target code correctly implements the source program. For this purpose, three requirements given in [15] are;

- A common semantics framework for the source and the target code representation
- Formalizing the refinement relation which shows preservation of source program semantics in the target code
- An automatic state simulation mechanism for verification that the target program model correctly implement the source code

With probably more than a 6 person-year effort, the European project ASSUME (Affordable Safe & Secure Mobility Evolution) has implemented translation validation in the theorem prover Coq [16] for the proto-code generator of a four-instruction data-flow language: Vélus [17] and by relying existing techniques from the verified CompCert C compiler infrastructure [18], resulting itself of remarkable efforts. Upgrading this proto-language to a full-fledged code generator and its evolution will undoubtedly require tremendously more effort. By contrast, our approach to isolate the verifier (and possibly its evolution to a verified SAT solver) from the implementation of the code generation pipeline makes it easy to adapt to the code generator evolution while providing strong correctness guarantees about the generated code, with minimum engineering efforts.

The source SIGNAL and the target Verilog programs are in different languages, which require more efforts to prove the semantics preservation than the common semantics codes. To reduce the efforts, we explore the tool Yosys [19] that can generate the formal model of target Verilog programs and then developed a Java application that can transform the target model to the format acceptable to SIGALI model checker. The Polychrony Toolset generates the model of SIGNAL programs for its associated model checker SIGALI [8]. Yosys generates the model of the target Verilog program in the SMT-LIB language format. We transform this SMT-LIB format model for the SIGALI model checker. Then we adopted a similar approach as given in [20].

The organization of the rest of the paper is as follows. Section II describes the basic statements and informal semantics of the SIGNAL language. Section III illustrates the formal model and automatic generation of the formal model from a SIGNAL language program. Section IV presents the method for generating a formal model for a Verilog program and its translation for SIGALI model checker. Section V highlights the translation validation approach and its application for our work to validate the translation. Finally, Section VI concludes this paper.

II. An Overview of SIGNAL

The SIGNAL language raises the abstraction level of data-flow specifications of reactive system design by considering zero computation and communication time. Therefore, abstract specifications in SIGNAL become independent of hardware or hardware/software embedded architectures to facilitate the early design stages as well as formal analysis. The compiler of the SIGNAL language stepwise concretizes and optimizes source specifications. Execution or computation time considered at the later stage when deploying the design on a chosen architecture.

Input/output or local variables represent the signals in the SIGNAL language, and they are considered the infinite sequence of typed (e.g., real, integer, Boolean) values. The clock of a signal is the occurring frequency or the time instants at which a signal is present. If "x" represents a signal, then "$\hat{x}$" denotes its clock. The clock of a signal is an event type signal and it has only value "true" when the signal is present otherwise absent denoted by "$\perp$". Equations or statements of the SIGNAL language represent the relations among signals and have the information of clocks and data dependency of the signals involved. Process or program in the SIGNAL language is a parallel composition of equations. Synchronous composition of processes "P" and "Q" denoted by "$P|Q$" takes into account all the solutions of the equations simultaneously at any instant [21]. The primitive processes or operators of SIGNAL are described below:

Instantaneous Relation: The instantaneous relation or n-array function has syntax

$$x := f(x_1, x_2, \dots, x_n) \tag{1}$$

It describes the relationship between the typed sequences of "n" operands signals with the sequence of typed resultant signal "x" using function "f". The function "f" may have logical or arithmetic operators (e.g., not, and, or, +, -, *, /). All the signals involved in (1) have the same clock as

$$\hat{x} = \hat{x}_1 = \hat{x}_2 = \cdots = \hat{x}_n \tag{2}$$

The data dependency graph in (3) shows that at the clock instance "$\hat{x}$", the value of "x" depends on the value of "x_i"

$$\forall\, i \in \mathbb{N}_{>0}\,,\ x_i \xrightarrow{\hat{x}} x \tag{3}$$

Delay: The following equation relates the value of "x" with the previous value of "x_1" where "v_0" is the initial value of "x"

$$x := x_1\$1\ \ v_0 \tag{4}$$

The "x" and "x_1" are synchronous with the same clock as shown below

$$\hat{x} = \hat{x}_1 \tag{5}$$

There is no implicit data dependency in the delay process.

Deterministic Merge: Two signal "x_1" and "x_2" are merged with the operator "default" to signal "x" deterministically as

$$x := x_1 \ \ default \ \ x_2 \tag{6}$$

The merging function enables adding control in data-flow. When "x_1" is present, the value of "x_1" is assigned to "x" otherwise when "x_1" is absent and "x_2" is present the "x" carries the value of "x_2". The clock of "x" is the union of clocks of "x_1" and "x_2" as

$$\hat{x} = \hat{x}_1 \cup \hat{x}_2 \tag{7}$$

The data dependency graph of merging is

$$x_1 \xrightarrow{\hat{x}_1} x \xleftarrow{\hat{x}_2 \setminus \hat{x}_1} x_2 \tag{8}$$

Sampling: This is another data-flow control-related instruction. The following equation in the SIGNAL language defines the downsampling of "x_1" when signal "c" is present and holds the value true.

$$x := x_1 \ when \ \ c \tag{9}$$

The type of "x" and "x_1" is the same while "c" is a Boolean signal or conditional expression evaluated to a Boolean value. The implicit clock relation of equation (9) is

$$\hat{x} = \hat{x}_1 \cap [c] \tag{10}$$

The clock of "x" is the intersection of "$\hat{x}_1$" and $[c]$. The clock of Boolean signal "c" (denoted by $\hat{c} = [c] \cup [\neg c]$) has two sub-clock $[c]$ and $[\neg c]$. The sub-clock $[c]$ indicates that the "c" is present and holds the value true while $[\neg c]$ means "c" is present and holds the value false. The data dependency shows that the value of "x" depends on the value of "x_1" at the clock occurrence "$\hat{x}$" as

$$x_1 \xrightarrow{\hat{x}} x \tag{11}$$

Processes Compositions: If (P_1, P_2, ..., P_n) are "n" process then ($|P_1|P_2|...|P_n|$) denotes their parallel composition. Processes communicate using their common signals. The following composition of processes represents the systems of equations at the same time index and contains only two equations.

$$(|\ x := pre\_x + y \ |\ pre\_x := x \ \$1 \ c\ |)$$

The above system of equations with time index t ≥ 1 is

$$x_t = pre\_x_t + \ y_t \ \ and \ \ \ pre\_x_t \ = x_{t-1}, \ \ \ pre\_x_1 = c$$

By combining both equations, we can write

$$x_t = \ x_{t-1} + y_t \ with \ initial \ value \ x_0 = c \ \ \because pre\_x_1 = x_0$$

Restriction: The "P/x" denotes the restriction of a signal "x" to the process "P". It means "x" is a local signal to process "P" and invisible outside the process "P".

The interested readers are referred to [22], [23], [24] for detailed semantics and syntax. *TABLE I* shows the summary of primitive operators of SIGNAL language.

TABLE I

SIGNAL PROCESSES, CLOCK RELATIONS, AND DATA DEPENDENCIES

Process Name	Process Equation	Data Dependency	Clock Relation
Function	$x:=f(x_1,x_2,...,x_n)$	$\forall i \in \mathbb{N}_{>0}, x_i \xrightarrow{\hat{x}} x$	$\hat{x}=\hat{x}_1=...=\hat{x}_n$
Delay	$x := x_1\$1 \ v_0$		$\hat{x} = \hat{x}_1$
Merge	$x :=x_1 \ default \ x_2$	$x_1 \xrightarrow{\hat{x}_1} x \xleftarrow{\hat{x}_2 \setminus \hat{x}_1} x_2$	$\hat{x} = \hat{x}_1 \cup \hat{x}_2$
Sampling	$x := x_1 \ when \ \ c$	$x_1 \xrightarrow{\hat{x}} x$	$\hat{x} = \hat{x}_1 \cap [c]$
Composition	$(\|P_1\|P_2\| \ldots \|P_n\|)$		
Restriction	P/x		

III. FORMAL MODEL GENERATION OF SIGNAL PROGRAM

The Galois field $\mathcal{F}_p$ or $\mathbb{Z}/p\mathbb{Z}$ has a finite number of elements (i.e., integer mode p where p is a prime number). The SIGNAL equations in a program are encoded over Galois field [25] or finite field $\mathbb{Z}/3\mathbb{Z}$ (i.e., integer modulo 3) for analyzing their dynamical behavior. After algebraic encoding of SIGNAL programs over $\mathbb{Z}/3\mathbb{Z}$, we obtain a Polynomial Dynamical System (PDS) model in the format acceptable by SIGALI model checker. The Polychrony Toolset can automatically generate formal PDS model of the submitted SIGNAL program over the $\mathcal{F}_3$ finite field when compiled with the option "-z3z". After loading the resultant model file, the SIGALI model checker offers verification of various system properties and discrete controller synthesis [26]. *TABLE II* shows the translation of primitive processes of the SIGNAL language in $\mathcal{F}_3$.

TABLE II

SIGNAL PRIMITIVE PROCESSES TRANSLATION TO POLYNOMIALS OVER □3

SIGNAL Instruction	Non-Boolean Signals	Boolean Signals
$x :=f(x_1,\ldots,x_n)$	$x^2 = x_1^2 = \cdots = x_n^2$	
$x := \ not \ x_1$		$x \ = -x_1, \ \ x^2 = x_1^2$
$x := x_1 \ and \ \ x_2$		$x=x_1x_2(x_1x_2 -x_1 -x_2 -1)$ $x^2 = x_1^2 = x_2^2$
$x := x_1 \ or \ \ x_2$		$x=x_1x_2(1 -x_1 -x_2 -x_1x_2)$ $x^2 = x_1^2 = x_2^2$
$x := x_1 \ when \ \ c$	$x^2 = x_1^2(-c - c^2)$	$x = x_1(-c - c^2)$
$x :=x_1 \ default \ x_2$	$x^2=x_1^2 + x_2^2 - x_1^2x_2^2$	$x \ = \ x_1 + (1 - x_1^2)x_2$
$x := x_1\$1 \ v_0$	$x^2 = x_1^2$	$\acute{s} \ = x_1 + (1 - x_1^2)s$ $x \ = x_1^2 s$ $s_0 \ = v_0$
$(\|P_1\|P_2\| \ldots \|P_n\|)$	Union of the encoding of $(\|P_1\|P_2\| \ldots \|P_n\|)$	
P/x	Same as the encoding of process P	

In $\mathcal{F}_3$ field, we encode a signal with three possible values (i.e., 1, -1 and 0) proposed in [27]. The encoding of non-Boolean signals only shows the status of presence and absence with "±1" and "0" respectively. Boolean signals encoding has both information for clocks and values (i.e., present∧true↔1, present∧false↔-1 and absent↔0). Therefore, if a signal is "x" then "x^2=1" indicates its present status (in either case when x=1 or x= -1, the x^2=1). The equation "x^2=y^2" shows the synchronization of two signals "x" and "y". The translation of delay equation in $\mathcal{F}_3$ introduces a state variable "s" for memorization of the latest previous value of a signal "x_1" with initial value "s_0" and "$\acute{s}$" indicates its next value.

After translating all the instructions of a SIGNAL program into polynomials in $\mathcal{F}_3$, we have a system of PDS equations as a model for dynamical behavior analysis. The PDS model has three types of sub-systems of polynomial equations as

$$PDS = \begin{cases} \acute{X} = P(X,Y) \\ Q(X,Y) = 0 \\ Q_0(X) \quad = 0 \end{cases} \tag{12}$$

Where

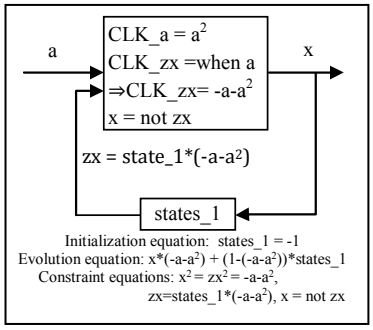
- X is a vector of "n" state variables represented by $(\mathbb{Z}/3\mathbb{Z})^n$, comes from the translation of delay operator
- Y is a set of "m" event variables, represented by a vector in $(\mathbb{Z}/3\mathbb{Z})^m$
- $X' = P(X, Y)$ is the evolution equation of the system. It is considered the vector-valued function $[P_1, P_2, \ldots, P_n]$ from $(\mathbb{Z}/3\mathbb{Z})^{n+m}$ to $(\mathbb{Z}/3\mathbb{Z})^n$, it describes the dynamical behavior of the system
- $Q(X, Y) = 0$ is the constraint equation. It is a vectorial equation $[Q_1, Q_2, \ldots, Q_l]$. It specifies when an event can occur in a given state and characterizes the static aspect of the system
- $Q_0(X) = 0$ is the initialization equation of the system, it assigns initial values to state variables

In the PDS system, a polynomial or a variable can only take values {-1, 0, 1} which belongs to the finite field $\mathcal{F}_3$. In SIGALI, the polynomials are represented by Ternary Decision Diagram (TDD). The TDD [28] is a slight extension of the Binary Decision Diagram (BDD) [29] in which non-terminal nodes have three outgoing edges. The PDS is a finite state transition system [30], describes the state transition from current state $x \in (\mathbb{Z}/3\mathbb{Z})^n$ to next state x' characterizes by $X' = P(X, Y)$, when the event $y \in (\mathbb{Z}/3\mathbb{Z})^m$ occurs meanwhile satisfying the constraint equation $Q(X, Y) = 0$.

Let we have a SIGNAL program saved in a file (altern.SIG) has a synchronized Boolean input "a" with Boolean output "x" as shown in *Fig. 1 (a)* and its translation into polynomial equations in *Fig. 1(b)*. The next value of output in this program depends on input and the state of the system. The use of delay operator introduces a register for memorization of the latest previous value of "x" let it is "states_1". The signal "x" and "zx" have the same clock. The state of the system will change when the input "a" is present with value "true".

```
%alternating the signal x %
process ALTERN =
(? boolean a;
 ! boolean x)
(| x ^= when a
 | x := not(zx)
 |zx := x$1
 |)
where boolean zx init false;
end;
```

(a)



(b)

Fig. 1 (a) SIGNAL process ALTERN (b) Translation to PDS

When we compile it by the command "signal –z3z altern.SIG", the SIGNAL Toolbox generates a file "ALTERN_F3.z3z" for SIGALI model checker with contents as given in Listing 1. The initialization, evolution, and constraint equations of the PDS are observable in the following listing.

Listing 1: Formal model of ALTERN generated by SIGNAL Toolbox

```
declare(a, states_1);
CLK_a : a ^2;
CLK_zx : when a;
zx : states_1 * CLK_zx;
x : not zx;
events: [a];
states: [states_1];
initialisations: [states_1 = - 1];
evolutions: [x * CLK_zx + (1 - CLK_zx) * states_1];
constraints: [      ];
controllables: [      ];
free_cond: [      ];
```

After loading and building the PDS model of "ALTERN_F3.z3z" in SIGALI, we can check the dynamical properties such as liveness, invariance, attractivity, and reachability.

IV. Model Generation of Verilog Code and its Translation for SIGALI

For translation validation, we have to build a formal model of the source program and the target program in a common semantics framework. The previous section describes the formal model generation for the source SIGNAL program. Here, we describe the formal model building of the target program of Verilog, and its translation into the "z3z" format.

A. Formal Model Generation from Verilog Program

The Yosys [19] is a free and open source tool under the ISC license which can generate formal model of submitted Verilog program into SMT-LIB [31] v2 (current version is 2.6) language format. Almost all Satisfiability Modulo Theories (SMT) solvers (e.g., Z3, Yices2, Boolector) supports the SMT-LIB standard format as a common input language. Another associated Python script "yosys-smtbmc" supports bounded model checking on the model generated by Yosys. Listing 2 shows the generated Verilog code from The Polychrony Toolset for the SIGNAL process ALTERN in a file "ALTERN.vl".

Listing 2: Verilog code generated by The Polychrony Toolset

```
/*This file has been generated by Polychrony version 4.22*/
`include "VerilogPredefTypes.vl"
module ALTERN(a,x,VL_TICK,VL_RESET);
  input `WIRE_reg a, VL_TICK, VL_RESET;
  output `REG_reg x=0;
   `include "VerilogPredefFunctions.vl"
  always @(posedge VL_TICK)
  begin      if (VL_RESET) begin          x = `FALSE;          end
    else
     begin
      if (a)
         begin
         x = !x;
         end
     end
  end
  endmodule
```

Fig. 2 shows the Yosys commands to load Verilog module "ALTERN" and prepare it for generating the formal model.

```
yosys> read_verilog ALTERN.vl
1. Executing Verilog-2005 frontend.
Parsing Verilog input from `ALTERN.vl' to AST representation.
Generating RTLIL representation for module `\ALTERN'.
Successfully finished Verilog frontend.

yosys> prep -top ALTERN
```

Fig. 2 Loading and synthesis of Verilog module in Yosys

Finally, The Yosys backend generates the model in SMT-LIBv2 format, as shown in Listing 3 where the function "ALTERN_t" indicates the transition relation and "ALTERN_i" represents the initial state values.

Listing 3: Model generated by Yosys in SMT-LIBv2 format

```
(declare-sort |ALTERN_s| 0)
(declare-fun |ALTERN_is| (|ALTERN_s|) Bool)
(declare-fun |ALTERN#0| (|ALTERN_s|) Bool)
(define-fun |ALTERN_n VL_RESET| ((state |ALTERN_s|)) Bool
    (|ALTERN#0| state))
(declare-fun |ALTERN#1| (|ALTERN_s|) Bool)
(define-fun |ALTERN_n VL_TICK| ((state |ALTERN_s|)) Bool
    (|ALTERN#1| state))
(declare-fun |ALTERN#2| (|ALTERN_s|) Bool)
(define-fun |ALTERN_n a| ((state |ALTERN_s|)) Bool (|ALTERN#2|
    state))
(declare-fun |ALTERN#3| (|ALTERN_s|) (_ BitVec 1))
(define-fun |ALTERN_n x| ((state |ALTERN_s|)) Bool (= ((_ extract 0
    0) (|ALTERN#3| state)) #b1))
(define-fun |ALTERN#4| ((state |ALTERN_s|)) Bool (not (or  (= ((_
    extract 0 0) (|ALTERN#3| state)) #b1) false)))
(define-fun |ALTERN#5| ((state |ALTERN_s|)) (_ BitVec 1) (ite
    (|ALTERN#2| state) (ite (|ALTERN#4| state) #b1 #b0)
    (|ALTERN#3| state)))
(define-fun |ALTERN#6| ((state |ALTERN_s|)) (_ BitVec 1) (ite
    (|ALTERN#0| state) #b0 (|ALTERN#5| state)))
(define-fun |ALTERN_a| ((state |ALTERN_s|)) Bool true)
(define-fun |ALTERN_u| ((state |ALTERN_s|)) Bool true)
(define-fun |ALTERN_i| ((state |ALTERN_s|)) Bool
(= (= ((_ extract 0 0) (|ALTERN#3| state)) #b1) false))
(define-fun |ALTERN_h| ((state |ALTERN_s|)) Bool true)
(define-fun |ALTERN_t| ((state |ALTERN_s|) (next_state |ALTERN_s|))
    Bool(= (|ALTERN#6| state) (|ALTERN#3| next_state)))
```

B. *Translation of Generated Model of Verilog for SIGALI*

For Boolean relations, the PDS model has information for both values and clocks while for numerical relations, it contains only the information of clock constraints [26]. Therefore, we selected a simple Boolean value program "ALTERN" as a case study to demonstrate the correctness of translation by symbolic simulation in SIGALI environment. We developed a Java application, which can translate the SMT2 file generated by Yosys to the file acceptable for SIGALI. It translates the inputs variables of SMT2 model of Verilog to event variables and declares them in the declaration statement of corresponding ".z3z" file. For each output variable register, which retains its previous value, we introduce a state variable in the output file. In the model shown in Listing 3, "ALTERN_i" indicates the initial state function and is translated to corresponding initialization equations in ".z3z" output file. The "ALTERN_t" indicates transition relation function, which is translated to the evolution equation in the corresponding "XXX.z3z" file. The Fig. 3 shows the translation mechanism and output generated automatically with our Java application from "ALTERN.smt2" file to "ALTERN.z3z". In this program, there are no constraints among input clocks; therefore, left empty otherwise, our application translates and writes constraints equations automatically in the output model.

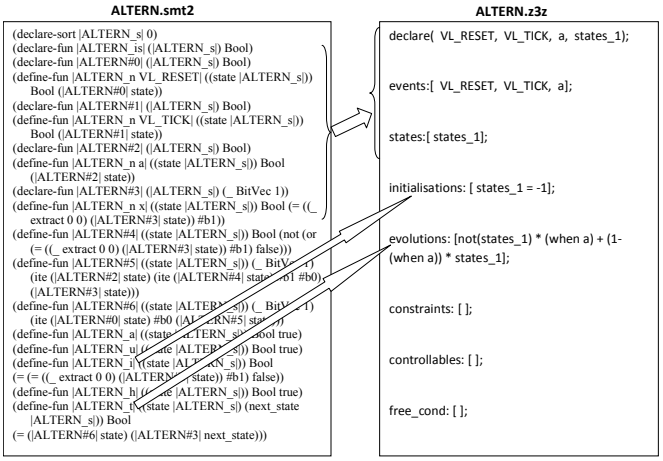


Fig. 3 Translation of model generated by Yosys for SIGALI

V. VERIFICATION OF CORRECT TRANSLATION

After translating the model of target code, we have both models for the source and the target program in common semantics of PDS. *Fig. 4* shows the overall semantics preservation verification scheme of our methodology.

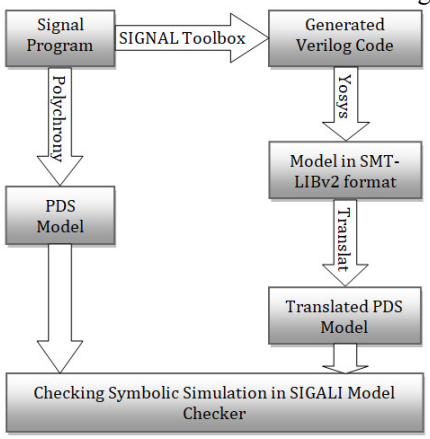


Fig. 4 Overall scheme for checking semantics preservation

A. Description of Correct Translation or Refinement

The PDS represents the Labeled Transition System (LTS) intensionally called Intensionally LTS (ILTS) where polynomials represent the actions of automata [32]. Instead of describing each event and transition extensionally, the ILTS can interpret the system as classical LTS. The Intensionally Labeled Transition System (ILTS) over $\mathbb{Z}/3\mathbb{Z}$ is defined as

Definition 1: ILTS is an m-dimension structure S = (X, I, Y, T) where X is a set of states, $I \subseteq X$ is a set of initial states, Y is a set of "m" variables $Y_1, Y_2, \ldots, Y_m$ and $T \subseteq X \times \mathbb{Z}/3\mathbb{Z}[Y] \times X$ is the transition relation. Each state transition is labeled by a polynomial $\mathbb{Z}/3\mathbb{Z}[Y]$ over the set of Y.

The ILTS in *Definition 1* can represent the PDS model and can be extracted directly from eq (12). $\mathbb{Z}/3\mathbb{Z}[Y]$ represents the set of polynomials over "*m*" variables $Y_1, Y_2, \ldots, Y_m$ whose coefficients range over finite field $\mathbb{Z}/3\mathbb{Z}$. For a typical element, we write the polynomial *P(Y)* or only *P*. Therefore, state transition over polynomial is written $x \xrightarrow{P(Y)} \acute{x}$ or simply $x \xrightarrow{P} \acute{x}$. The solution set of a given polynomial $P(Y) \in \mathbb{Z}/3\mathbb{Z}[Y]$ is represented by $Sol(P) \subseteq (\mathbb{Z}/3\mathbb{Z})^m$. Each arrow of ILTS is labeled by *P(Y)* intensionally indicates all the arrows labeled by $y \in Sol(P)$.

The transition systems S_1 and S_2 are bisimulation equivalent ($S_1 \sim S_2$) if they match each state and transition and can simulate each other in a stepwise manner [33].

Theorem 1: (Symbolic Bisimulation) Let $S_1 = (X_1, I_1, Y, T_1)$ and $S_2 = (X_2, I_2, Y, T_2)$ be two ILTS. A symbolic bisimulation between S_1 and S_2 ($S_1 \sim S_2$) is a binary relation $\mathcal{R} \subseteq X_1 \times X_2$ Such that $(x_1, x_2) \in \mathcal{R}$ whenever

(i) $\forall x_1 \in I_1\,,\ \exists x_2 \in I_2$ *and* $(x_1, x_2) \in \mathcal{R}$ *and*

$\forall \left(x_1 \xrightarrow{P} \acute{x}_1\right) \in T_1 \exists$ *a finite set of transitions* $\left(x_2 \xrightarrow{P_i} x_2^i\right)_{i \in \mathcal{I}} \in T_2$ *with*

- $(1 - P^2) * \prod_i P_i = 0$, *which implies* $Sol(P) \subseteq Sol(\prod_i P_i)$, *and*
- $(\acute{x}_1, x_2^i) \in \mathcal{R},\ \forall i \in \mathcal{I}$ $(where\ \mathcal{I}\ is\ a\ set\ of\ indexes)$, *and*

(ii) *Vice versa*

Theorem 1 states that every initial state of S_1 have a relation with the initial state of S_2 and every state transition in S_1 labeled by a set of events represented by *Sol(P)*, there exist some matching state transition in S_2 labeled by the same set of events and vice versa. The Polynomial $(1 - P^2) * \prod_i P_i = 0$ intended meaning is that $Sol(\ (1 - P^2) * \prod_i P_i\ ) = Sol(0) = (\mathbb{Z}/3\mathbb{Z})^m$ which implies $Sol(P) \subseteq Sol(\prod_i P_i)$.

We describe the ILTS for implementation purpose as

Definition 2: The $S = (X, \acute{X}, I, Y, T)$ is an (n, m)-dimensional structure for ILTS where X is a set of "n" current states, X′ is a set of "n" next states, I is a set of initial states, Y is a set of "m" labelling event variables and $T \subseteq X \times \mathbb{Z}/3\mathbb{Z}[Y] \times X'$ is a set of transition relations described by polynomials.

Let $L_v = (X_1, \acute{X}_1, I_1, Y, T_1)$ and $L_s = (X_2, \acute{X}_2, I_2\, Y, T_2)$ represent ILTS for the target Verilog and the source SIGNAL programs respectively. The ILTS "L_v" of the target program may not have the same event variables of ILTS "L_s" because of concretization or adding the implementation details in the target program. Therefore, we consider only the common event variables set "*Y*" in both "L_v" and "L_s" and other variables as hiding event variables [28]. We know that the target program is translated from the source program. The target program may exhibit fewer behaviors than the source program due to transformations/optimizations. Therefore, we can relax the criteria of symbolic bisimulation given in *Theorem 1* and check only one directional symbolic simulation from target to source model as

Theorem 2: Let $L_v = (X_1, \acute{X}_1, I_1, Y, T_1)$ and $L_s = (X_2, \acute{X}_2, I_2\, Y, T_2)$ represents the target and the source ILTS over the same set of the event variables "Y". A symbolic simulation is a binary relation (L_v, L_s) such that $(x_1, x_2) \in \mathcal{R}$ whenever

- $\forall x_1 \in I_1\,,\ \exists x_2 \in I_2$ *and* $(x_1, x_2) \in \mathcal{R}$
- $\forall \left(x_1 \xrightarrow{P} \acute{x}_1\right) \in T_1 \Rightarrow \exists$ *a finit set* $\left(x_2 \xrightarrow{P_i} x_2^i\right)_{i \in \mathcal{I}} \in T_2$ *and* $(\acute{x}_1, x_2^i) \in \mathcal{R}$

Theorem 2 states that for any event, which triggers a transition in the target model, there are some matching transitions in the source model for the same event and every initial state of the target model there exit some related initial state in the source model. The transitions are labeled by a set of events represented by solutions of polynomial *P* or *P(y)* over common event variables *Y*. It means if symbolic simulation (*Lv, Ls*) exists then target program implies the source program or refines the source program ($Lv \subseteq Ls$), and the target program preserves the clock semantics of the source program. Thus translation is correct.

B. Implementation of Symbolic Simulation in SIGALI

For the implementation of symbolic simulation, we describe the ILTS of the source and the target programs in SIGALI as a 5-tuple $(X, \acute{X}, I, Y, T)$, where *X* is a set of current state variables, *X′* is a set of next state variables, *Y* is a set of event variables, *I* represents initialization equation "$I(X_0) = 0$" and *T* represents a group of polynomials "$T(X, Y, X') = 0$" for transition relations. If "*S*" describes the PDS system, then SIGALI command "*implicit_sys(S)*" builds the corresponding ILTS and gives access to its five components. The following algorithm based on *Theorem 2* provides a method to compute symbolic simulation by fix-point computation in SIGALI.

ALGORITHM 1: Computation of symbolic simulation

Given: $L_v = (X_1, \acute{X}_1, I_1, Y, T_1)$ and $L_s = (X_2, \acute{X}_2, I_2\, Y, T_2)$

Required: $\mathcal{R}(X_1, X_2)$

$\mathcal{R}_0(X_1, X_2) = 0$ % define the polynomial %

$$Do \left\{ \begin{pmatrix} \mathcal{R}_{k+1}(X_1, X_2) is\ the\ canonical\ generator\ of\ the\ \equiv -class\ of \\ \mathcal{R}_k(X_1, X_2) \cap \\ \left\{ \forall \acute{X}_1 \forall Y \left[T_1(X_1, Y, \acute{X}_1) \Rightarrow \exists \acute{X}_2 \left(T_2(X_2, Y, \acute{X}_2) \cap \mathcal{R}_k(\acute{X}_1, \acute{X}_2) \right) \right] \right\} \end{pmatrix} \right\}$$

$While\ \left(\mathcal{R}_k(X_1, X_2)\right)_k\ not\ converging$

$$if\ \left(\forall X_1 \left[I_1(X_1) \Rightarrow \exists X_2 \left(I_2(X_2) \cap \mathcal{R}(X_1, X_2) \right) \right] \right)$$

$$Output\ result = \mathcal{R}(X_1, X_2)$$

$$else \qquad Output\ result = \mathcal{R}(X_1, X_2) = 1$$

The above algorithm gives the result $\mathcal{R}(X_1, X_2) = 0$, if there exists a symbolic simulation (L_v, L_s). The SIGALI commands implementing the *Algorithm 1* are given in *Fig 5*. It is the verification by symbolic simulation that the PDS model of the target program preserves the clock semantics of the source PDS model of process "ALTERN". We used the SIGALI implementation given in [6]. The SIGALI model checker treats the text as comments starting with "%" character.

```
read("ALTERN.z3z");          % Loading  target model in SIGALI%
S1:processus(events, states, evolutions, initialisations,
[gen(constraints)],controllables);
S_I1:implicit_sys(S1);       % Building corresponding ILTS %

read("ALTERN_F3.z3z");       % Loading source model in SIGALI%
S2:processus(events, states, evolutions, initialisations,
[gen(constraints)],controllables);
S_I2:implicit_sys(S2);       % Building corresponding ILTS %

read("Property.lib");        % Loading necessary library file %
def implies(P1,P2):          % P1 => P2 is internal function %
      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                        % defining the 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                           % computing the state 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
      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),
      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),
      X_2 = declare_suff(X_2_d),  % renaming the states variables %
      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                           %computing the systems simulation%
      intersection(states_sim,
      forall(implies(I_1,exist(intersection(states_sim,I_2),X_2)),X_1));
```

Fig. 5 Translation validation implementation in SIGALI

The "S_I1" and the "S_I2" represent the target and the source ILTS respectively as shown in *Fig. 5*. After loading the both models, the SIGALI gives the output result "%0" if the symbolic simulation computed by the command "ilts_simulation(S_I1, S_I2)" is correct, as shown in *Fig. 6* otherwise it give the output "%1" as shown in *Fig 7*.

```
Sigali : ilts_simulation(S_I1,S_I2);

------------------------------------------------------------
%0

------------------------------------------------------------
Utilisateur : 0.00, Systeme : 0.00, Clock = 0.00

Sigali :
```

Fig. 6 SIGALI output for correct symbolic simulation

We deliberately added a bug in the source model given in Listing 1 by assigning a different initialization value to the state variable as "states_1= 1" than the target model initialization value (states_1= -1). The SIGALI gives the output result "%1" as shown in *Fig. 7* indicates that the source model does not simulate the target model.

```
Sigali : ilts_simulation(S_I1,S_I2);

------------------------------------------------------------
%1

------------------------------------------------------------
Utilisateur : 0.00, Systeme : 0.00, Clock = 0.00

Sigali :
```

Fig. 7 SIGALI output for incorrect symbolic simulation

We developed an application to translate the model generated by Yosys to the model acceptable by SIGALI and is available at https://github.com/amjadphool/SMT2Z3Z with some examples programs. This translator is written for Boolean value signals, and by adopting the same methodology, it can be extended for complex programs as well. The symbolic simulation can be characterized by the size of fix-point computation measured as the number of TDD nodes to manipulate polynomial equations. The SIGALI does not show the number of TDD nodes when the size of the symbolic simulation is small. *TABLE III* shows the results of symbolic simulation of some example programs where the columns X, Y, TDD Nodes and Correct indicate the number of state variables, number of common event variables, number of TDD nodes, and correctness of symbolic simulation respectively. The first column of *TABLE III* shows the name of the symbolic simulation of the target to the source model.

TABLE III

SYMBOLIC SIMULATION RESULTS

Simulation Name	X	Y	TDD Nodes	Correct
(ALTERN.z3z, ALTERN_F3.z3z)	1	1	Small	Yes
(andgate.z3z, andgate_F3.z3z)	1	2	Small	Yes
(orgate.z3z, orgate_F3.z3z)	1	2	Small	Yes
(andornotgate.z3z, andornotgate_F3.z3z)	3	2	Small	Yes

VI. CONCLUSION

The design and implementation of safety-critical embedded systems require rigorous analysis before handed over to the end-user. One promising option for designing the safety-critical systems is by using the framework of the synchronous languages. We used the synchronous language SIGNAL to

specify the behavior of the system at a higher abstraction level. The SIGNAL compiler is available in the Polychrony Toolset, which provides a common framework for formal analysis and automatic code generation for various general-purpose languages like C, C++, and Java. We are working on automatic and verified HDL Verilog code generation from the SIGNAL language. In this paper, we present a technique for the verification of a Verilog code generator by translation validation. Translation validation by model checking requires the source and the target program models to be defined in the same semantics model. For this purpose, The Polychrony Toolset generates the PDS model of the source Signal program, and Yosys generates the model of target program of Verilog in SMT-LIBv2. The target model in the SMT-LIB format is translated into the PDS model, which is acceptable to SIGALI model checker. The SIGALI model checker takes both models and performs symbolic simulation for producing the proof of correct translation. Since Yosys is an open source tool, therefore another way is to modify its backend to generate the PDS model directly from Verilog code. In the future, we will perform translation validation by building a clock formula of the target and the source program in the SMT-LIB format to mitigate state space explosion problem.

ACKNOWLEDGMENT

This work was supported by the National Natural Science Foundation of China under Grant 61672074, 91538202, and Project of National Key Research and Development of China under Grant 2018YFB1402702.

REFERENCES

[1] D. C. Schmidt, "Model-driven engineering," *Computer.*, vol. 39, no. 2, pp. 25–31, 2006.

[2] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud, "The synchronous data flow programming language LUSTRE," *Proc. IEEE*, vol. 79, no. 9, pp. 1305–1320, 1991.

[3] G. Berry and G. Gonthier, "The ESTEREL synchronous programming language: design, semantics, implementation," *Sci. Comput. Program.*, vol. 19, no. 2, pp. 87–152, 1992.

[4] P. Le Guernic, T. Gautier, M. Le Borgne, and C. Le Maire, "Programming Real-Time Applications with SIGNAL," in *Proceedings of the IEEE*, 1991, vol. 79, no. 9, pp. 1321–1336.

[5] A. Benveniste and G. Berry, "The synchronous approach to reactive and real-time systems," *Proc. IEEE*, vol. 79, no. 9, pp. 1270–1282, 1991.

[6] V. C. Ngo, "Formal Verification of a Synchronous Data-flow Compiler : from Signal to C," Ph.D. dissertation, University of Rennes 1, France, 2014.

[7] A. Gamatié., *Designing Embedded Systems with the SIGNAL Programming Language*. Springer, 2010.

[8] "Polychrony." [Online]. Available: http://polychrony.inria.fr/.

[9] H. M. Amjad, J. Niu, K. Hu, N. Akram, and L. Besnard, "Verilog Code Generation Scheme from Signal Language," in *16th International Bhurban Conference on Applied Sciences and Technology (IBCAST)*, 2019, pp. 457–462.

[10] M. Belhadj, "Using VHDL for Link to Synthesis Tools," in *The Third Annual Atlantic Test Workshop*, 1994.

[11] M. Belhadj, "VHDL & Signal: A Cooperative Approach," in *International Conference on Simulation and Hardware Description Languages (ICSHDL)*, 1994, pp. 76–81.

[12] A. A. Kountouris and C. Wolinski, "Hierarchical conditional dependency graphs as a unifying design representation in the CODESIS high-level synthesis system," in *The 13th International Symposium on System Synthesis*, 2000, pp. 66–71.

[13] C. Wolinski and M. Belhadj, "High Level Synthesis of Globally Asynchronous Locally Synchronous Circuits," in *The Third Annual Atlantic Test Workshop*, 1994.

[14] A. A. Kountouris and C. Wolinski, "Method for the generation of HDL code at the RTL level from a high-level formal specification language," in *Midwest Symposium on Circuits and Systems*, 1997, vol. 2, pp. 1095–1098.

[15] A. Pnueli, M. Siegel, and E. Singerman, "Translation validation," in *4th International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, 1998, vol. 1384, pp. 151–166.

[16] "The Coq Proof Assistant." [Online]. Available: https://coq.inria.fr/.

[17] T. Bourke, L. Brun, P.-É. Dagand, X. Leroy, M. Pouzet, and L. Rieg, "A formally Verified Compiler for Lustre," in *38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2017, vol. 52, no. 6, pp. 586–601.

[18] "The CompCert Project." [Online]. Available: http://compcert.inria.fr.

[19] Clifford Wolf, "Yosys Open SYnthesis Suite." [Online]. Available: http://www.clifford.at/yosys/.

[20] V. C. Ngo, J. P. Talpin, T. Gautier, P. Le Guernic, and L. Besnard, "Formal Verification of Compiler Transformations on Polychronous Equations," in *Integrated Formal Methods*, 2012, pp. 113–127.

[21] P. Le Guernic, J. P. Talpin, and J. C. Le Lann, "Polychrony for system design," *J. CIRCUITS Syst. Comput.*, vol. 12, no. 3, pp. 261–303, 2003.

[22] A. Benveniste, P. Le Guernic, Y. Sorel, and M. Sorine, "A denotational theory of synchronous communicating systems," Inria Research Report 685, Rennes, France, 1987.

[23] P. Le Guernic and A. Benveniste, "Real-time,synchronous,data-flow programming: the language "SIGNAL" and its mathematical semantics," Research Report 533 (revised version:620), INRIA, France, 1986.

[24] L. Besnard, T. Gautier, and P. Le Guernic, "SIGNAL V4–INRIA version: Reference Manual," 2015.

[25] M. Le Borgne, A. Benveniste, and P. Le Guernic, "Polynomial dynamical systems over finite fields," in *Algebraic Computing in Control*, 1991, pp. 212–222.

[26] H. Marchand, P. Bournai, M. Le Borgne, and P. Le Guernic, "Synthesis of Discrete-Event Controllers Based on the Signal Environment," *Discret. Event Dyn. Syst.*, vol. 10, no. 4, pp. 325–346, Oct. 2000.

[27] A. Benveniste and P. Le Guernic, "Hybrid Dynamical Systems Theory and the SIGNAL Language," *IEEE Trans. Automat. Contr.*, vol. 35, no. 5, pp. 535–546, 1990.

[28] S. Pinchinat, H. Marchand, and M. Le Borgne, "Symbolic Abstractions of Automata and their application to the Supervisory Control Problem," RR-3814, 1999.

[29] R. E. Bryant, "Graph-Based Algorithms for Boolean Function Manipulation," *IEEE Trans. Comput.*, vol. C–35, no. 8, pp. 677–691, 1986.

[30] M. Le Borgne, H. Marchand, É. Rutten, and M. Samaan, "Formal verification of Signal programs: Application to a power transformer station controller," in *Algebraic Methodology and Software Technology*, 1996, pp. 271–285.

[31] C. Barrett, P. Fontaine, and C. Tinelli, "The Satisfiability Modulo Theories Library (SMT-LIB)," 2016. [Online]. Available: www.SMT-LIB.org.

[32] O. Kushnarenko and S. Pinchinat, "Intensional Approaches for Symbolic Methods," RR-3448, INRIA, Rennes, 1998.

[33] C. Baier and J.-P. Katoen, *Principles of Model Checking*. The MIT Press, 2008.