Specification and Analysis of Null Convention Logic (NCL) Circuits Using **PAFSV**

Ka Lok Man, Nan Zhang, Chi-Un Lei, Eng Gee Lim, T. O. Ting, Kaiyu Wan, Jieming Ma, Tomas Krilavičius, Dawei Liu and Yu Guo

***Abstract*—In this work, a process algebraic framework known as PAFSV is applied to the formal specication and analysis of IEEE 1800TM SystemVerilog design. The formal semantics of PAFSV is defined by means of deduction rules that associate a time transition system with a PAFSV process. In addition, a set of properties of PAFSV is defined for a notion of bisimilarity; and PAFSV may be regarded as the formal language of a significant subset of IEEE 1800TM SystemVerilog. The main aim of this paper is to demonstrate that PAFSV is effective and useful for the formal specification and analysis of IEEE 1800TM SystemVerilog design. To achieve the aim of this approach, we apply PAFSV to model and analyse classical circuits such as the Null Convention Logic (NCL) circuit.**

***Index Terms*—SystemVerilog, formal semantics, formal language, Null Convention Logic (NCL)**

I. Introduction

Process Algebra Framework for System Verilog (**PAFSV**) [1], [2] is the formalisation of a reasonable subset of IEEE 1800TM SystemVerilog based on the classical process algebras Algebra of Communicating Processes (ACP) [3]. The semantics of **PAFSV** has been defined by means of deduction rules in a Structured Operational Semantics (SOS) [4] style that associates a Time Transition System (TTS) [5] with a **PAFSV** process. In addition, a set of properties of **PAFSV** is defined for a notion of bisimilarity.

The introduction of **PAFSV** initiated an attempt to extend the knowledge and experience collected in the field of process algebras to SystemVerilog designs. **PAFSV** is aimed at giving formal specifications of SystemVerilog designs and to perform formal analysis of SystemVerilog processes. Furthermore, **PAFSV** is a single formalism that can be used for specifying concurrent systems, nite state systems and real-time systems (as in SystemVerilog). Desired properties of these systems specified in **PAFSV** can be verified with existing formal verification tools by translating them into different formats that are the input languages of formal verification tools. Hence, **PAFSV** can be purportedly used for formal verification of SystemVerilog designs. In order to show that **PAFSV** is useful for the formal specification and analysis of SystemVerilog designs, in this paper, we apply **PAFSV** to model and analyse classical circuits like Null Convention Logic (NCL) circuits [6]–[8] via the formal verification tool PAT [9].

Manuscript received December 10, 2013; revised January 23, 2014.

Ka Lok Man, Nan Zhang, Kaiyu Wan, Jieming Ma and Dawei Liu are with the Department of Computer Science and Software Engineering, Xi'an Jiaotong-Liverpool University, China. Email: {ka.man, nan.zhang, kaiyu.wan, dawei.liu}@xjtlu.edu.cn, jieming.ma08@student.xjtlu.edu.cn

Chi-Un Lei is with Department of Electrical and Electronic Engineering, the University of Hong Kong, Hong Kong. Email: culei@eee.hku.hk

Eng Gee Lim and T. O. Ting are with the Department of Electrical and Electronic Engineering, Xi'an Jiaotong-Liverpool University, China. Email: {enggee.lim, toting}@xjtlu.edu.cn

Tomas Krilavičius is with Baltic Institute of Advanced Technologies, Vilnius, Lithuania. Email: t.krilavicius@bpti.lt

Yu Guo is with the Department of Computer Science and Technology, University of Science and Technology of China. Email: guoyu@ustc.edu.cn

This paper is organised as follows. Section II briefly presents the formal syntax and formal semantics of our process algebraic framework **PAFSV**. Further, in Section III, the effectiveness and applicability of **PAFSV** is demonstrated by applying **PAFSV** to model and analyse common Null Convention Logic Circuits (NCL). Finally, the concluding remarks are made in Section IV.

II. PAFSV

We briefly present the process algebraic framework **PAFSV** in this paper. For more details, we refer to [1], [2].

A. Goals of PAFSV

PAFSV has a formal and compositional semantics based on a time transition system for formal specification and analysis of SystemVerilog designs. The intention of our process algebraic framework **PAFSV** is as follows:

- to give a formal semantics to a significant subset of SystemVerilog using the operational approach of [4];
- to serve as a mathematical basis for improvement of design strategies of SystemVerilog and possibilities to analyse SystemVerilog designs;
- to initiate an attempt to extend the knowledge and experience of the field of process algebras to SystemVerilog designs;
- to be used as the formal language for a significant subset of SystemVerilog.

B. Formal syntax

Process terms are the core elements of the **PAFSV**. The set of process terms **P** in **PAFSV** is defined according to the following grammar for the process terms $p \in \mathbf{P}$:

$$\begin{array}{rll} p ::= & \mathbf{deadlock} \mid \mathbf{skip} \mid x := e \\ & \mid \mathbf{delay}(n) \mid \mathbf{any}\ p \mid \mathbf{if}(b)\ p\ \mathbf{else}\ p \\ & \mid p; p \mid \mathbf{wait}(b)\ p \mid \mathbf{while}(b)\ p \\ & \mid \mathbf{assign}\ w := e \mid @_{(\eta_1(l_1),\ldots,\eta_n(l_n))}\ p \\ & \mid p \circledast p \mid p \parallel p \mid \mathbf{repeat}\ p \\ & \mid \mathbf{assert}(b)\ p \mid p\ \mathbf{disrupt}\ p \end{array}$$

Here, x and w are variables taken from the set Var which consists of all variables; and $n \in \mathbb{R}_{\geq 0}$. b and e denote a Boolean expression and an expression over variables from Var, respectively. Moreover, $\eta_1, \ldots, \eta_n$ represent Boolean functions with corresponding parameters $l_1, \ldots, l_n \in$ Var.

In $\mathbf{PAFSV}$, we allow the use of common arithmetic operators (e.g. $+, -$), relational operators (e.g. $=, \geq$) and logical operators (e.g. $\wedge, \vee$) as in mathematics to construct expressions over variables from Var.

The operators are listed in descending order of their binding strength as follows:
{**if**(_)**_else**_, **wait**(_)_, **while**(_)_, **assert**(_)_}, _; _, _**disrupt**_, {_ $\circledast$ _, _ $\|$ _}.
The operators inside the braces have equal binding strength. In addition, operators of equal binding strength associate to the right, and parentheses may be used to group expressions. For example, $p;\ q;\ r$ means $p;\ (q;\ r)$, where $p, q, r \in \mathbf{P}$

Apart from process terms: **deadlock**, **skip**, **any**_, _**disrupt**_, and _ $\circledast$ _, all other syntax elements in $\mathbf{PAFSV}$ are the formalisation of the corresponding language elements (based on classical process algebra tenets) in SystemVerilog.

Process terms **deadlock** and **skip**; and operator _ $\circledast$ _ are mainly introduced for calculation and axiomatisation purposes. The **any**_ operator was originally introduced in Hybrid Chi [11]. It is used to give an arbitrary delay behaviour to a process term. We can make use of this operator to simplify our deduction rules in a remarkable way.

The _**disrupt**_ is inspired by the analogy of the disrupt operator in HyPA [12]. This can be used to model event controls in $\mathbf{PAFSV}$ in a very efficient way.

C. Atomic process terms

The atomic process terms of $\mathbf{PAFSV}$ are process term constructors that cannot be split into smaller process terms. They are:

1) The *deadlock* process term **deadlock** is introduced as a constant, which represents no behaviour. This means that it cannot perform any actions or delays.
2) The *skip* process term **skip** can only perform the internal action τ to termination, which is not externally visible.
3) The *procedural assignment* process term $x := e$ assigns the value of expression e to variable x (in an atomic way).
4) The *continuous assignment* process term **assign** $w := e$ continuously watches for changes of the variables that occur on the expression e. Whenever there is a change, the value of e is re-evaluated and then propagated immediately to w.
5) The *delay* process term **delay**(n) denotes a process term that first delays for n time units, and then terminates by means of the internal action τ.

D. Operators

Atomic process terms can be combined using the following operators. The operators are:

1) By means of the application of the *any* operator to process term $p \in \mathbf{P}$ (i.e. **any** p), delaying behaviour of arbitrary duration can be specified. The resulting behaviour is such that arbitrary delays are allowed. As a consequence, any delay behaviour of p is neglected. The action behaviour of p remains unchanged. This operator can even be used to add arbitrary behaviour to an undelayable process term.
2) The *if_else* process term **if**(b) p **else** q first evaluates the boolean expression b. If b evaluates to *true*, then p is executed, otherwise $q \in \mathbf{P}$ is executed.
3) The *sequential composition* of process terms p and q (i.e. $p;\ q$) behaves as process term p until p terminates, and then continues to behave as process term q.
4) The *wait* process term **wait**(b) p can perform whatever p can perform under the condition that the boolean expression b evaluates to $true$. Otherwise, it is blocked until b becomes $true$.
5) The *while* process term **while**(b) p can perform whatever p can do under the condition that the boolean expression b evaluates to $true$ and then followed by the original iteration process term (i.e. **while**(b) p). In case b evaluates to *false*, the while process term **while**(b) p terminates by means of the internal action τ.
6) The *event* process term $@_{(\eta_1(l_1),\ldots,\eta_n(l_n))}$ p can perform whatever p can perform under the condition that any of the boolean functions $\eta_1(l_1), \ldots, \eta_n(l_n)$ returns to $true$. If there is no such a function, p will be triggered by $\eta_1(l_1), \ldots, \eta_n(l_n)$. Intuitively, functions $\eta_1, \ldots, \eta_n$ are used to model event changes as event controls $levelchange$, $posedge$ and $negedge$ in SystemVerilog.
7) The *alternative composition* of process terms p and q (i.e. $p \circledast q$) allows a non-deterministic choice between different actions of the process term either p or q. With respect to time behaviour, the participants in the alternative composition have to synchronise.
8) The *parallel composition* of process terms p and q (i.e. $p \parallel q$) executes p and q concurrently in an interleaved fashion. For the time behaviour, the participants in the parallel composition have to synchronise.
9) The *repeat* process term **repeat** p represents the infinite repetition of process term p. Note that the idea behind the *repeat* statement in SystemVerilog is slightly different from **repeat** p in $\mathbf{PAFSV}$. The repeat statement specifies the number of times a loop to be repeated. The same goal can be achieved by using the repeat process term in combination with the if_else process term in $\mathbf{PAFSV}$.
10) The *assert* process term **assert**(b) p checks immediately the property b (expressed as a boolean expression). If b holds, p is executed.
11) The *disrupt* process term p **disrupt** q intends to give priority of the execution of process term p over process term q.

E. Formal semantics

A $\mathbf{PAFSV}$ process is a tuple $\langle p, \sigma \rangle$, where $p \in \mathbf{P}$ and $\sigma \in \Sigma$ which denotes the set of all states. In $\mathbf{PAFSV}$, a state is a partial function which maps variables to values. The formal semantics of $\mathbf{PAFSV}$ is defined by constructing the Timed Transition System (TTS) [5] for each process term and each possible valuation of variables (i.e. a state). In such a TTS, three different kinds of transition relations are dened, namely:

1) one associated with termination transition;
2) one associated with action transition (for discrete action);

TABLE I: Boolean truth tables with NULL value Boolean functions: (a)AND, (b) OR and (C) NOT.

(a) AND

	T	*F*	*N*
T	T	F	N
F	F	F	N
N	N	N	N

(b) OR

	T	*F*	*N*
T	T	T	N
F	T	F	N
N	N	N	N

(c) NOT

T	F
F	T
N	N

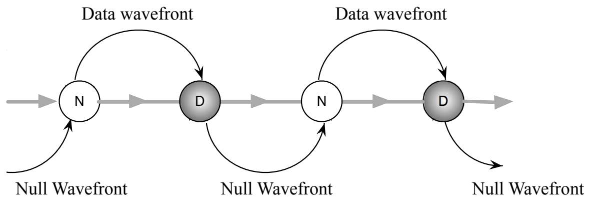

Fig. 1: Wavefronts in a data flow.

3) one associated with time transition (delay behavior).

III. Modelling and Analysis of Null Convention Logic (NCL) Circuits in **PAFSV**

In this section, we apply **PAFSV** to specify and analyse common Null Convention Circuits (NCL)

A. Null Convention Logic

Boolean functions determine the output values only based on the input value and as the speed of different signal paths varies, a chaos of intermediate result transitions may be delivered ahead of valid stable transitions. It is hard to express the boundaries of instantiation and resolution by traditional time-dependent and symbolic-value-dependent Boolean logic. Null Conventional Logic (NCL) [6]–[8] is a deviation from conventional Boolean system where the value of the signal itself is used to show its arrival/presence/validity. Thus each variable in the expression has two values, DATA (indicating the value as well as validity) and NULL (indicating the absence of data).

There are two conceptual flows for signals in an NCL implementation: the flow of valid data items called the data-wavefront and the flow of NULL items (to clear all states) called the null-wavefront. Details are shown in Figure 1.

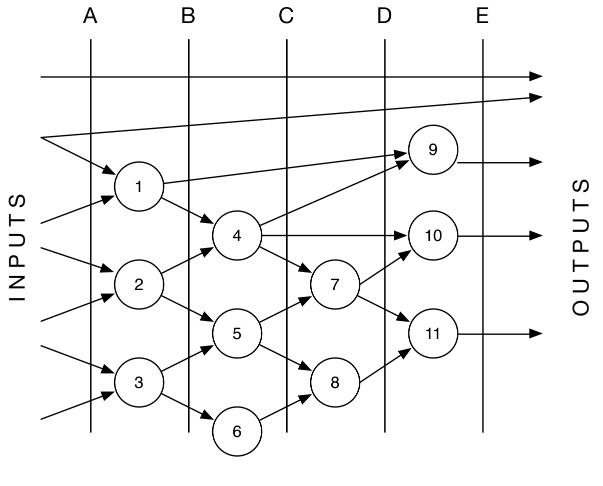

Fig. 2: Presentation (validation) boundaries for input variables.

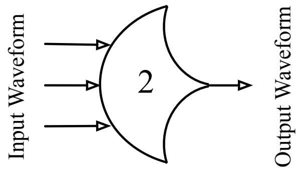

Fig. 3: Presentation diagram for a basic 2-of-3 NCL gate.

TABLE II: Dual-rail encoding.

Logic Value	Encoding D0	Encoding D1
Data1	0	1
Data0	1	0
NULL	0	0
Invalid	1	1

One set of input values will lead to one set of output values. The final generation of output can be easily detected by using a completion detection circuit. To read a new set of input values, we need to flush out all the previously generated outputs by making all the inputs NULL. For example, Figure 2 shows a connection of NCL gates (1-11) forming a multi-level logic implementation. Here A, B, C, D and E are the logical boundaries for the representation of a signal. The final set of generated outputs (at the boundary E) can be valid only when all the outputs of boundary D are valid, and so on. Following the chain, one can conclude that all the output elements can be generated only when all the inputs have arrived [14].

1) Threshold gate: Basic elements of NCL circuits are threshold gates, referred as Thmn gate. It has n input terminals with m thresholds, where $1 \leq m \leq n$. A sample NCL gate is shown in Figure 3. The gate has three inputs and has a threshold of two (the number written inside the gate). It is therefore named 2-of-3 NCL gate. To make the output of the gate to logic, at least two out of three input lines must be at logic. For instance, if a, b and c are the inputs to the gate then the logical equation represented by the gate can be written as $Output = a \cdot b + b \cdot c + c \cdot a$. To reset the output to logic, all the inputs must however go to logic irrespective of the threshold. Until all the inputs are reset, the gate holds the previous state. Note that an N-of-N NCL gate is equivalent to an N input C-element [7]. These threshold gates can be used to express a burst of inputs required to produce an output.

2) Representing Data values in NCL: 2NCL is a kind of NCL logic that obtains only one data value, which indicates that the signal path can transit between DATA and NULL without intermediate values. Multiple mutually exclusive values are normally expressed by multiple signal paths. In a binary system, a dual-rail signal D, which is transmitted by two mutually exclusive wires (D0 and D1), is applied to express True and False. As seen in Table II, Boolean logic 0 and 1 are equivalent to the Data0 (D0=1, D1=01) and Data1 (D0=0, D0=1) respectively. NULL state comes only when the inputs receive the empty set; and the state D0=1, D1=1 is not permitted.

3) An AND gate: Consider the Boolean AND operation represented by the equation Z=A·B, where Z, A and B can be

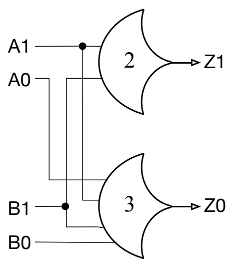

Fig. 4: Schematic view of an AND gate.

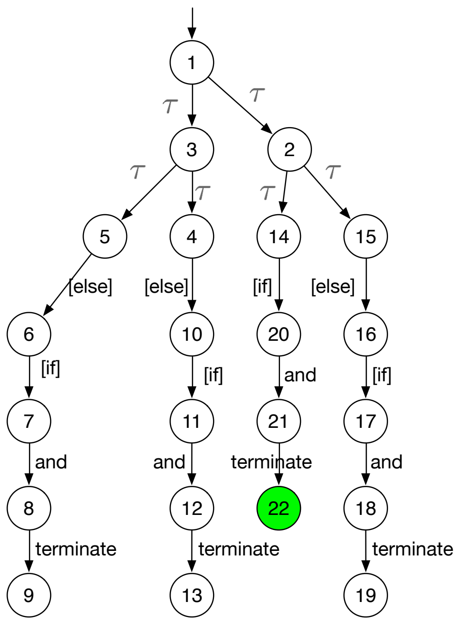

Fig. 5: A simulation run of an AND gate represented as a transition graph (simulated by PAT).

either logical 1 or logical 0. 2NCL combinational expressions present logical 1 and 0 by dual-rail signal Z and have to be described by two individual equations. The equations are given below:

```
Z1 = A1B1
Z0 = A1A0 + B1B0 + A1B0 + B1A0 + A0B0
```

Note that Z1 can be mapped to TH22 gate and Z0 can be expressed using TH34w22 gate [9], where w22 means two inputs have the weight of 2. The NCL diagram for the AND-gate is shown in Figure 4.

The description for the gate is given below. For brevity, we do not include the corresponding SystemVerilog representation for all examples in this section.

$$\begin{array}{l}
\textbf{if}(A1 \wedge B1)\ (Z1 := TRUE\ ;\ Z0 := FALSE)\ \textbf{else skip}\circledast \\
\textbf{if}(A1 \wedge B0)\ (Z0 := TRUE\ ;\ Z1 := FALSE)\ \textbf{else skip}\circledast \\
\textbf{if}(A0 \wedge B1)\ (Z0 := TRUE\ ;\ Z1 := FALSE)\ \textbf{else skip}\circledast \\
\textbf{if}(A0 \wedge B0)\ (Z0 := TRUE\ ;\ Z1 := FALSE)\ \textbf{else skip}
\end{array}$$

We propose a two-level translation process in this paper. NCL is mapped to PASFV, and then PASFV specification is translated to CSP# process expressions [15] and verified using PAT tool [9]. A similar process is used in [16], where NCL is translated to DISP (a flavor of CSP [17] and DI algebra [18]) and then verified using tool chain `di2pn` [19] and `petrify` [20]; or in [21], where NCL is translated to DISP and then mapped to CSP# and verified using PAT. In this paper we just provide a general outline of NCL to PASFV and PASFV to CSP# mapping, and do not provide formal translations.

The CSP# expression for the AND-gate is:

$$var A0; var A1; var B0; var B1; var Z0; var Z1;$$

$$\begin{array}{l}
Init() \\
= ((tau\{A0 = false; A1 = true;\} \to Skip)[] \\
\quad (tau\{A0 = true; A1 = false;\} \to Skip)); \\
\quad ((tau\{B0 = false; B1 = true;\} \to Skip)[] \\
\quad (tau\{B0 = true; B1 = false;\} \to Skip));
\end{array}$$

$$\begin{array}{l}
Adesc() \\
= if(A1 == true\ \&\&\ B1 == true) \\
\quad \{or\{Z0 = false; Z1 = true;\} \to Skip\} \\
\quad else\ if((A1 == true\ \&\&\ B0 == true) \\
\qquad ||(A0 == true\ \&\&\ B1 == true) \\
\qquad ||(A0 == true\ \&\&\ B0 == true)) \\
\quad \{or\{Z0 = true; Z1 = false;\} \to Skip\} \\
\quad else\ \{Skip\};
\end{array}$$

$$\begin{array}{l}
AND() \\
= Init()\ ;\ Adesc()\ ;
\end{array}$$

Assertion-based verification is a methodology that has been dormant for many years and is now widely applied in hardware design. Besides plenty modeling features, a number of useful assertions are supported in PAT. Assertions assist to capture the design intent. They monitor behaviors during simulation, detect and report errors. By means of assertions, verification can start in earlier design stage, bugs can be detected and resolved easily. Design engineers can incorporate their intent into programs to minimize integration issues. Given P() as a process, the basic assertions used are described below:

- `#assert P() deadlockfree`: performs Depth-First-Search or Breath-First-Search algorithm to detect the state with no further move except for successfully terminated states.
- `#assert P() divergencefree`: checks if there is a process performing transitions forever without executing useful events.
- `#assert P() deterministic`: asks if there are no two out-going transitions pointing to different states but with the same events.
- `#assert P() nonterminating`: Depth-First-Search or Breath-First-Search algorithm is applied to detect the state with no further move, including successfully terminated states.

All the examples are verified with the above assertions. The output of the PAT tool has showed the validity of the assertions. In Figure 5 we show the simulation runs for the AND operator.

The NCL schematic, **PAFSV** descriptions and CSP# expressions along with verification results for some more constructs are shown below.

4) An OR gate: The NCL schematic is shown in Figure 6. In Figure 7 we show the simulation run for the OR operator.

The **PAFSV** description:

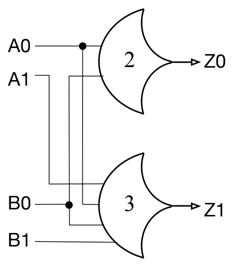

Fig. 6: Schematic view of an OR gate.

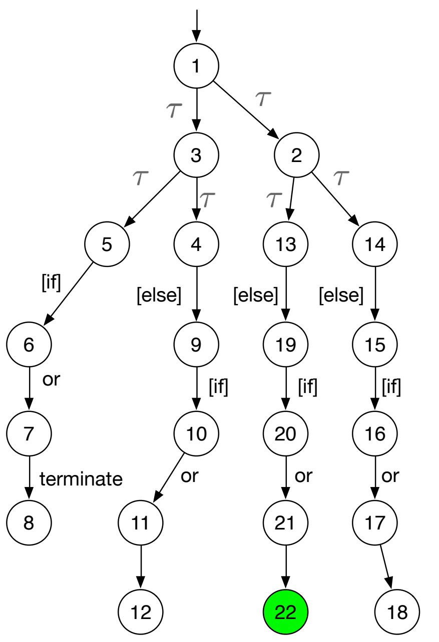

Fig. 7: A simulation run of an OR gate represented as a transition graph (simulated by PAT).

if$(A0 \wedge B0)$ $(Z0 := TRUE\ ;\ Z1 := FALSE)$ **else skip**$\circledast$
if$(A1 \wedge B0)$ $(Z1 := TRUE\ ;\ Z0 := FALSE)$ **else skip**$\circledast$
if$(A1 \wedge B1)$ $(Z1 := TRUE\ ;\ Z0 := FALSE)$ **else skip**$\circledast$
if$(A0 \wedge B1)$ $(Z1 := TRUE\ ;\ Z0 := FALSE)$ **else skip**

The CSP# expressions for the OR-gate is:

```
var A0; var A1; var B0; var B1; var Z0; var Z1;

Init()
= ((tau{A0 = false; A1 = true; } → Skip)[]
    (tau{A0 = true; A1 = false; } → Skip));
  ((tau{B0 = false; B1 = true; } → Skip)[]
    (tau{B0 = true; B1 = false; } → Skip));

Odesc()
= if(A0 == true && B0 == true)
        {or{Z0 = true; Z1 = false; } → Skip}
else if((A1 == true && B0 == true)
            ||(A0 == true && B1 == true)
            ||(A1 == true && B1 == true))
        {or{Z0 = false; Z1 = true; } → Skip}
else {Skip};

OR()
= Init() ; Odesc() ;
```

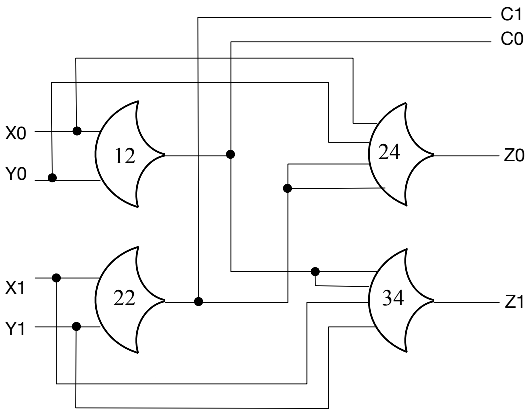

Fig. 8: Schematic view of an half adder.

5) A half adder: The NCL schematic is shown in Figure 8. In Figure 9 we show the simulation run for the half adder. **PAFSV** description:
if$((X1 \wedge Y1) \vee (X0 \wedge Y0))$ $(Z1 := FALSE\ ;\ Z0 := TRUE)$ **else skip**$\circledast$
if$((X0 \wedge Y1) \vee (X1 \wedge Y0))$ $(Z1 := TRUE\ ;\ Z0 := FALSE)$ **else skip**$\circledast$
if$(X1 \wedge Y1)$ $(C1 := TRUE\ ;\ C0 := false)$ **else skip**$\circledast$
if$(X0 \vee Y0)$ $(Z0 := FALSE\ ;\ Z1 := true)$ **else skip**

The CSP# expressions for the half adder is:

```
var X0; var X1; var Y0; var Y1; var Z0; var Z1; var C0; var C1;

Init()
= ((tau{X0 = false; X1 = true; } → Skip)[]
    (tau{X0 = true; X1 = false; } → Skip));
  ((tau{Y0 = false; Y1 = true; } → Skip)[]
    (tau{Y0 = true; Y1 = false; } → Skip));

Hgate()
= if((X1 == true && Y1 == true)
       ||(X0 == true && Y0 == true))
         {s{Z0 = true; Z1 = false; } → Skip}
  else if((X1 == true && Y0 == true)
           ||(X0 == true && Y1 == true))
         {s{Z0 = false; Z1 = true; } → Skip}
  else {Skip};

Hcarry()
= if(X1 == true && Y1 == true)
        {c{C0 = false; C1 = true; } → Skip}
  else if((X1 == true && Y0 == true)
            ||(X0 == true && Y1 == true)
            ||(X0 == true && Y0 == true))
        {c{C0 = true; C1 = false; } → Skip}
  else {Skip};

HalfAdder()
= Init() ; (Hcarry()||Hgate()) ;
```

IV. Conclusion

In this paper, the formal syntax and semantics of **PAFSV** have been briefly described. The applicability of **PAFSV** for modeling and analysis of SystemVerilog design has been illustrated through several feasible examples on NCL circuits.

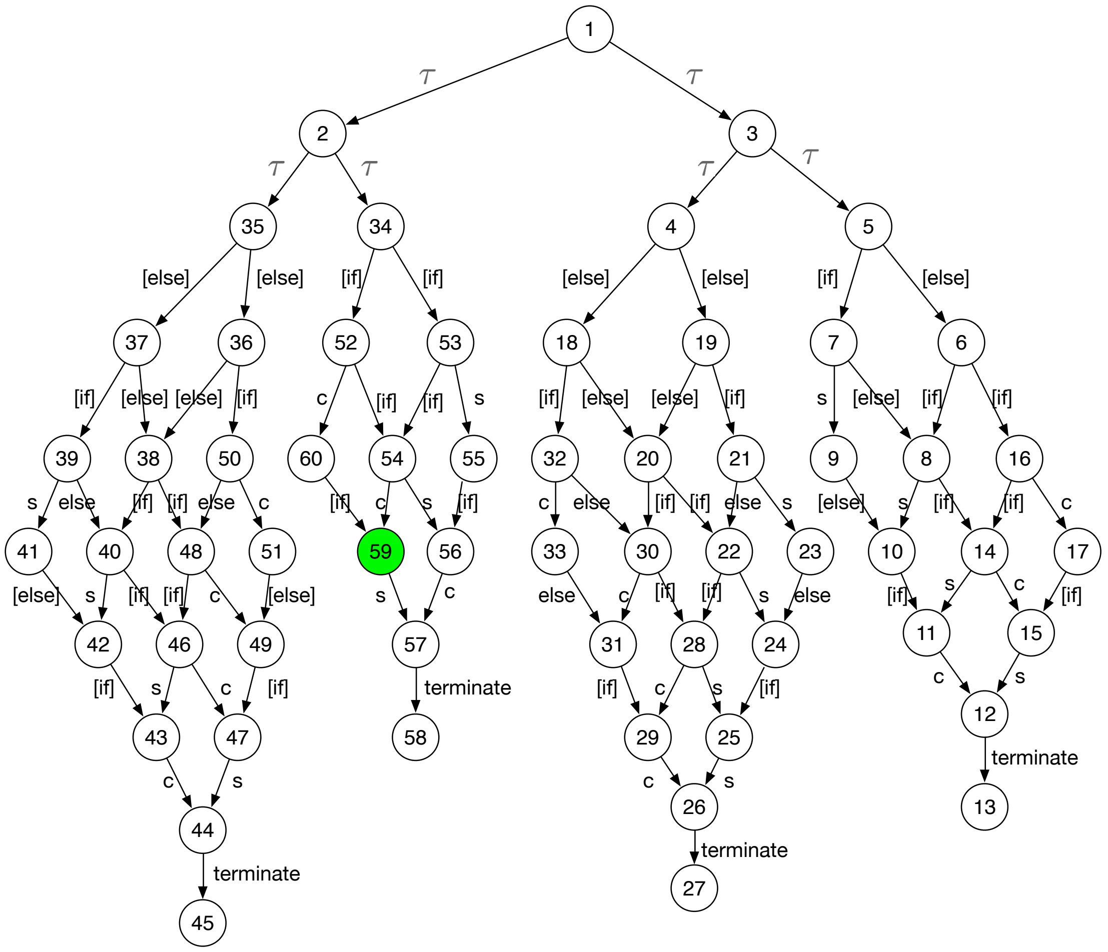

Fig. 9: A simulation run of a half adder represented as a transition graph (simulated by PAT).

Although our experience of using **PAFSV** is fruitful and positive, some industrial case studies are needed in order to validate and further develop our approach on using **PAFSV** for specification and verification of SystemVerilog design.

REFERENCES

[1] K. Man, M. Boubekeur, and M. Schellekens, "Algebraic approach to systemverilog," in *Proc. IEEE Canadian Conference on Electrical and Computer Engineering*, 2007. I, II

[2] K. Man, "Pafsv: A process algebraic framework for systemverilog," in *Proc. IEEE International Multiconference on Computer Science and Information Technology*, 2008, pp. 535–542. I, II

[3] J. Baeten and W. Weijland, *Process Algebra*. Cambridge University Press, 1990. I

[4] L. Aceto, W. Fokkink, and C. Verhoef, "Structural operational semantics," in *Handbook of Process Algebra*, 2001, pp. 197–292. I, II-A

[5] D. A. van Beek, K. L. Man, M. A. Reniers, J. E. Rooda, and R. R. H. Schiffelers, "Syntax and semantics of timed chi," Eindhoven University of Technology), Tech. Rep., 2005. I, II-E

[6] K. M. Fant and S. A. Brandt, "Null convention logic: A complete and consistent logic for asynchronous digital circuit synthesis," in *Proc. IEEE International Conference on. Application-specific Systems, Architectures and Processors*, 1996, pp. 261–273. I, III-A

[7] K. M. Fant, *Logically Determined Design: Clockless System Design with Null Conventional Logic*. New Jersey: Wiley Interscience, 2005. I, III-A, III-A1

[8] K. M. Fant and S. Brandt, *Null Convention Logic*. New Jersey: Wiley Interscience, 1994. I, III-A

[9] H. K. Kapoor, J. Ma, T. Krilavicius, K. L. Man, and C.-U. Lei, "Formal verification and synthesis of null conventional logic circuits," *IAENG Transactions on Engineering Technologies*, vol. 7, no. 1, pp. 320–333, Mar. 2012. I, III-A3, III-A3

[10] J. C. M. Baeten, T. Basten, and M. A. Reniers, *Process Algebra: Equational Theories of Communicating Processes*. Cambridge University Press, 2009.

[11] D. van Beek, K. Man, M. Reniers, J. Rooda, and R. Schiffelers, "Syntax and consistent equation semantics of hybrid chi," *Journal of Logic and Algebraic Programming*, vol. 68, no. 2, pp. 129–210, 2006. II-B

[12] P. Cuijpers and M. Reniers, "Hybrid process algebra," *Journal of Logic and Algebraic Programming*, vol. 62, no. 2, pp. 191–245, 2005. II-B

[13] IEEE-1800, "Ieee standard for systemverilog - unified hardware design, specification, and verification language," IEEE Std 1800$^{\text{TM}}$-2009, IEEE Computer Society, Tech. Rep., 2009.

[14] C. L. Seitz, *Introduction to VLSI Systems*. Addison-Wesley, 1980. III-A

[15] J. Sun, Y. Liu, J. S. Dong, and C. Chen, "Integrating specification and programs for system modeling and verification," in *Proc. IEEE International Symposium on Theoretical Aspects of Software Engineering*, 2009. III-A3

[16] H. Kapoor, A. Asthana, T. Krilavičius, W. Zeng, J. Ma, and K. Man, "Towards a language based synthesis of ncl circuits," in *Proc. IAENG International MultiConference of Engineers and Computer Scientists*, 2011, pp. 1033–1038. III-A3

[17] C. Hoare., "Communicating sequential processes," *Comm. ACM*, vol. 21, no. 8, pp. 666–677, Aug 1978. III-A3

[18] M. B. Josephs and J. T. Udding, "An overview of DI algebra," in *Proc. IEEE Hawaii Int. Conference on System Science*, 1993, pp. 329–338. III-A3

[19] D. Furey and M. B. Josephs., "Asynchronous circuit design via automated petri net generation," Tech. Rep., 2003. III-A3

[20] J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno, and A. Yakovlev, "Petrify: A tool for manipulating concurrent specifications and synthesis of asynchronous controllers," *IEICE Transactions on Information and Systems*, vol. 80, no. 3, pp. 315–325, 1997. III-A3

[21] J. Ma, H. Kapoor, T. Krilavičius, K. Man, N. Zhang, E. Lim, T. Jeong, S. Guan, and J. Seon., "Specification and analysis of ncl circuits," *IAENG Engineering Letters*, vol. 19, no. 3, pp. 215–222, 1978. III-A3