Computing and Informatics, Vol. 35, 2016, 143–176

PAFSV: A FORMAL FRAMEWORK FOR SPECIFICATION AND ANALYSIS OF SYSTEMVERILOG

Ka Lok Man

Department of Computer Science and Software Engineering
Xi'an Jiaotong-Liverpool University, Suzhou, China
e-mail: `ka.man@xjtlu.edu.cn`

Chi-Un Lei

The University of Hong Kong, Hong Kong

Hemangee K. Kapoor

Indian Institute of Technology Guwahati, India

Tomas Krilavičius

Vytautas Magnus University, Lithuania
&
Baltic Institute of Advanced Technology, Lithuania

Jieming Ma

Suzhou University of Science and Technology, China

Nan Zhang

Department of Computer Science and Software Engineering
Xi'an Jiaotong-Liverpool University, China
&
CITIC Securities, China

Abstract. We develop a process algebraic framework PAFSV for the formal specification and analysis of IEEE 1800^{TM} SystemVerilog designs. The formal semantics of PAFSV is defined by means of deduction rules that associate a time transition system with a PAFSV process. A set of properties of PAFSV is presented for a notion of bisimilarity. PAFSV may be regarded as the formal language of a significant subset of IEEE 1800^{TM} SystemVerilog. To show that PAFSV is useful for the formal specification and analysis of IEEE 1800^{TM} SystemVerilog designs, we illustrate the use of PAFSV with a multiplexer, a synchronous reset D flip-flop and an arbiter.

Keywords: SystemVerilog, process algebras, formal semantics, PAFSV, formal specification and analysis, circuit verification

Mathematics Subject Classification 2010: 68Q45

1 INTRODUCTION

The goal of developing a formal semantics is to provide a complete and unambiguous specification of the language. It also contributes significantly to the sharing, portability and integration of various applications in simulation, synthesis and formal verification. Formal languages with a semantics well-defined in *Computer Science* increase understanding of systems, increase clarity of specifications and help problem solving and remove errors. Over the years, several flavours of formal languages have gained the industrial acceptance.

Process algebras [1] are formal languages that have formal syntax and semantics for specifying and reasoning about different systems. They are also useful tools for verification of various systems. Generally speaking, the process algebras describe behaviour of processes and provide operations that allow to compose systems in order to obtain more complex systems.

Moreover, the analysis and verification of systems described using the process algebras can be partially or completely carried out by mathematical proofs using equational theory.

In addition, the strength of the field of process algebras is in the ability to use *Algebraic Reasoning* [1] (also known as equational reasoning) that allows rewriting processes using axioms (e.g. for commutativity and associativity) to a simpler form. By using axioms, we can also perform calculations with processes. These can be advantageous for many forms of analysis.

Process algebras have also helped to achieve a deeper understanding of the nature of concepts like observable behaviour in the presence of non-determinism, system composition by interconnection of system components modelled as processes in a parallel context, and notions of behavioural equivalence (e.g. bisimulation [1]) of such systems.

Serious efforts have been made in the past to deal with systems (e.g. real-time systems [2, 3] and hybrid systems [4, 5]) in a process algebraic way. Over the years, process algebras have been successfully used in a wide range of problems and in practical applications in both academia and industry fields for analysis of many different systems.

On the other hand, the need for a formal and well-defined semantics of a *Hardware Description Languages* (HDLs) is widely accepted and desirable for architects, engineers and researchers in the electronic design community. IEEE 1 800$^{\text{TM}}$ *SystemVerilog* [6] (SystemVerilog) is the industry's first unified hardware description and verification language (HDVL) standard [7, 8, 9]; and SystemVerilog is a major extension of the established IEEE 1 364$^{\text{TM}}$ *Verilog* language [10, 11]. However, the standard semantics of SystemVerilog is informal.

We believe that the fundamental tenets of process algebras are highly compatible with the behavioural approach of systems described in SystemVerilog. Hence, in this paper, we present PAFSV [12, 13] (Process Algebra Framework for SystemVerilog) that is suitable for modelling and analysis of systems described in SystemVerilog.

The formal semantics of PAFSV is defined by means of deduction rules in a standard *Structured Operational Semantics* (SOS) [14] style that associate a *Time Transition System* (TTS) [15] with a PAFSV process. A set of properties of PAFSV is presented for a notion of bisimilarity.

PAFSV covers the main features of SystemVerilog including decision statements and immediate assertions, and also aims to achieve a satisfactory level of abstraction and a more faithful modelling of concurrency. Although it is desirable and very important to have pure parallelism for hardware simulation, the SystemVerilog simulators "*in-use*" at this moment still implement parallelism via non-determinism. Therefore, we realise that it is more fruitful to develop our process algebraic framework for SystemVerilog such that the execution of a system described in such a framework (PAFSV) consists of interleaving transitions from concurrent processes.

Moreover, we adopt the view that a system described in PAFSV is a system in which an instantaneous state transition occurs on the system performing an action and a delay takes place on the system idling between performing successive actions. A technical advantage of our work (see also Section 7 for details) is that, in contrast to other attempts to formalise semantics of SystemVerilog, specifications described in PAFSV can be directly executable.

This paper is organised as follows. Section 2 presents a brief review focusing on concepts and features of SystemVerilog that are relevant to PAFSV. Section 3 shows the goals, the data types, time model, formal syntax and formal semantics of our process algebraic framework PAFSV. To illustrate the use, effectiveness and applicability of the deduction rules, in Section 4, some simple specifications of PAFSV are provided. In Section 5, correctness of the formal semantics of PAFSV defined in Section 3 is discussed; and a notion of equivalence is defined, which is shown to be a congruence for all PAFSV operators. Also, a set of useful properties of closed PAFSV process terms is given in the same section. Samples (modelling SystemVerilog designs) of the application of PAFSV and a formal analysis (by means of a complete mathematical proof) of a SystemVerilog design via PAFSV are shown in Section 6. The comparison with other formal approaches and the direction of future work are given in Section 7 and Section 9, respectively. Finally, concluding remarks are made in Section 8.

2 SYSTEMVERILOG

SystemVerilog is a unified hardware design, specification and verification language that is originally based on Accellera SystemVerilog 3.1a, as defined in [11]. Principally, SystemVerilog extends the use of the traditional hardware description language Verilog to efficiently and flexibly specify designs of ever increasing complexity and size, and to easily and effectively verify these complex designs.

This section presents a brief review focusing on concepts and features of SystemVerilog that are relevant to PAFSV. For an extensive treatment of SystemVerilog, the reader can be referred to [6, 10].

2.1 2-State Modelling

SystemVerilog extends the Verilog variable types by adding 2-state types, where each bit can be "0" or "1". Modelling *Register Transfer Level* (RTL) designs using 2-state logic leads simulation performance/enables simulators to be more efficient, because these variables can be used whenever the values "X" and "Z" are not needed, for example, in test benches and as for-loop variables.

2.2 Hardware-Specific Procedures

In Verilog, the **always** procedural block is used to represent RTL designs of sequential logic, combinational logic and latched logic. Synthesisers and other software tools need to infer the intent of the **always** procedural block from the context of the statements within the **always** procedural block. This infer may lead to mismatches between simulation and synthesis.

In addition to Verilog, three procedural blocks can be explicitly used in SystemVerilog to show the intent of the logic of a blocks. This increases the readability of the codes and avoids the mismatches between simulation and synthesis. The three procedural blocks are as follows:

always_ff – represents sequential logic block;

always_comb – represents combinational logic block;

always_latch – represents latched logic block.

2.3 Unique and Priority Statements

The **if_else** and **case** statements in Verilog can be a source of mismatches between RTL simulation and how synthesiser interprets such a RTL design. Similarly, an-other common mistake in Verilog RTL designs is the misuse of the **full_case** and **parallel_case** pragmats. Such misuses can be effectively avoided in SystemVerilog by using statements **unique** and **priority**, because these statements are used to instruct simulators, synthesisers and other software tools the specific type of hardware intended.

A brief description of the idea behind the statements **unique** and **priority** is given below:

unique – enforces completeness and uniqueness of the conditional, this means that only one branch of the conditional should be considered at run-time;

priority – enforces a less rigorous set of checks, it checks only that at least one branch of the conditional is considered.

2.4 Enhanced Fork-Join Statement Block

A Verilog **fork_join** statement block groups two or more statements together in parallel, so that all statements are evaluated concurrently. The block itself terminates only if all parallel statements have completed. Thus, the execution of any statements after a Verilog **fork_join** statement block is blocked until all parallel statements of such a **fork_join** statement block have completed execution successfully.

SystemVerilog allows the use of **fork_join** statement block construct in a more flexible way by means of two statement blocks **join_none** and **join_any**:

join_none – statements that follow a **join_none** statement block are not blocked when the parallel statements are executing;

join_any – statements that follow a **join_any** statement block are not blocked from execution as long as the first of any of parallel statements has completed execution successfully.

2.5 Clocking Blocks

Rather than applying traditional event-based methodology of specifying transition times for each test signals in test-benches, SystemVerilog allows the test-benches to be defined using a cycle-based methodology. In SystemVerilog, this requirement can be achieved by using the clocking block **clocking_endclocking**. The SystemVerilog **clocking_endclocking** clocking block specifies a clock signal, the timing and synchronisation requirements of the block in which the clock is used.

2.6 Assertion-Based Verification

SystemVerilog provides special language constructs that can be used to validate (through assertions) the behaviour of a design. An assertion is basically a sort of statement expressing something that must be true.

In SystemVerilog, there are two kinds of assertions namely *immediate assertions* and *concurrent assertions*:

Immediate assertions – they principally are intended to be used in simulation. They follow (event-based) simulation semantics for their execution and are executed as a statement in a procedural block.

Concurrent assertions – they are built on clock semantics and use sampled values of variables.

3 PAFSV

We propose a process algebraic framework namely PAFSV in this paper. Since it is not possible to cover all the aspects of SystemVerilog and define a process algebraic framework for it in one paper, we would outline the goals of our process algebraic framework PAFSV. Then, we present the data types, time model, formal syntax and formal semantics of PAFSV.

3.1 Our Goals

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 [14];
- to serve as a mathematical basis for improvement of design strategies of SystemVerilog and possibilities to analyse SystemVerilog designs;
- to serve as a coherent first step for a semantics interoperability analysis on semantics domain such as SystemC and $SystemC^{\mathbb{FL}}$;
- 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.

3.2 Data Types and Time Model

In order to define the semantics of processes, we need to make some assumptions about the data types:

1. Let Var denote the set of all variables ($x_0, \ldots, x_n, \texttt{time}$). Besides the variables $x_0, \ldots, x_n$, the existence of the predefined reserved global variable `time` which denotes the current time, the value of which is initially zero, is assumed. This variable cannot be declared.

2. Let Value denote the set of all possible values $(v_0, \dots, v_m, \bot)$ that contains at least all Integers, all Reals, all Shortreals, all 2-state values and all 4-state values as defined in SystemVerilog (see [6] for details); all Booleans and $\bot$, where $\bot$ denotes the "*undefinedness*".
3. We then define a *valuation* as a partial function from variables to values. Syntactically, a valuation is denoted by a set of pairs $\{x_0 \mapsto v_0, \dots, x_n, \mapsto v_n, \mathtt{time} \mapsto t\}$, where x_i represents a variable and v_i its associating value; and $t \in \mathbb{R}_{\geq 0}$.
4. Further to this, the set of all valuations is denoted by Σ.

Note that the type "array" in SystemVerilog is not formalised yet in PAFSV. However, the behaviour of elements in an array in SystemVerilog can be modelled in PAFSV by introducing fresh variables. As an example, for an array $A[0\!:\!10]$ in SystemVerilog, we can introduce fresh variables $A_0, \dots, A_{10}$ in PAFSV to associate correspondingly $A[0]$ with A_0, $A[1]$ with A_1 and so on.

The time in PAFSV is dense. So, timing is measured on a continuous time scale. PAFSV has a strong time determinism principle [3]. This means that passage of time cannot result in making a choice between the two operands of the choice. Also, the maximal progress [15], which is a process that can delay only if it cannot do anything else, is not implicit in PAFSV. According to our industrial experience in hardware system design, the time model used in PAFSV is well-suited for modelling the timing behaviour of hardware systems.

3.3 Formal Syntax

To avoid confusion with the informal definition of a process in SystemVerilog, it is important to clearly state that, in our process algebraic framework PAFSV, we choose the terminology "***a process term***" as a formal term (generated restrictively through the formal syntax of PAFSV) to describe the possible behaviour of a PAFSV process (see Section 3.5) and not a process, as defined in SystemVerilog.

Furthermore, process terms $p \in P$ are the core elements of the PAFSV. The semantics of those process terms is defined in terms of the core process terms given in this subsection. The set of process terms P is defined according to the following grammar for the process terms $p \in P$:

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

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

In 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: $\{\mathbf{if}(\_)\_\mathbf{else}\_, \mathbf{wait}(\_)\_, \mathbf{while}(\_)\_, \mathbf{assert}(\_)\_\}$, $\_;\_$, $\_\mathbf{disrupt}\_$, $\{\_ \circledast \_, \_ \parallel \_\}$.

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 $p, q, r \in P$.

Apart from process terms: **deadlock**, **skip**, **any**_, _**disrupt**_, and $\_ \circledast \_$, all other syntax elements in 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 [4] (to be precise, in Hybrid Chi, such an operator is called "*the any delay operator*" and denoted by "[ ]"). 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 (see Section 3.7 for details).

The _**disrupt**_ is inspired by the analogy of the disrupt operator in HyPA [5]. This can be used to model event controls in PAFSV in a very efficient way.

As in many programming languages, non-primitive language elements can be easily defined in terms of other primitive language elements. For instance, a **forever** S statement in SystemVerilog can be expressed as **while** (1=1) S, where S denotes a Verilog program. In this paper, for brevity, we do not include the formal syntax of the formalisation of some SystemVerilog statements and constructs (e.g. Non-blocking assignment (<=), **Fork_join**, **Join_any**, **Join_non**, **Priority**, **Unique**, **For** and **Case**), because they can be easily rewritten in terms of other syntax given above. Nevertheless, by means of illustrative examples shown in Section 3.8, the interpretation of some SystemVerilog statements and constructs (as indicated above) in PAFSV is given in terms of varied PAFSV process terms.

Also, in SystemVerilog, three procedures **always_ff**, **always_comb** and **always_latch**; and initial block **initial** are not formalised in PAFSV yet, because the three procedures are relevant (mainly) in simulation and synthesis results; and the use of an initial block in SystemVerilog can be captured by the initialisation of a PAFSV process (as shown in Section 6.3).

A concise explanation of the formal syntax of PAFSV is given below. Section 3.6 gives a more detailed account of its meaning.

3.4 Atomic Process Terms

The atomic process terms of 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.

3.5 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 $p \in P$ (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 $q \in P$ 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 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 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 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. The need and use of this operator will be illustrated in Section 6.3.

3.6 Formal Semantics

A PAFSV process is a tuple $\langle p, \sigma \rangle$, where $p \in P$ and $\sigma \in \Sigma$. In this subsection, we give a formal semantics to the syntax defined for PAFSV in the previous subsection, by constructing the Timed Transition System (TTS), for each process term and each possible valuation of variables (see [15] for details). In such TTS, three different kinds of transition relations are defined, namely:

1. one associated with termination transition;
2. one associated with action transition (for discrete action);
3. one associated with time transition (delay behaviour).

Definition 1. The set of actions A_τ contains at least $aa(x, v)$ and τ, where $aa(x, v)$ is the assignment action (i.e. the value of v is assigned to x) and τ is the internal action. The set A_τ is considered as a parameter of PAFSV that can be freely instantiated.

Definition 2. We give a formal semantics for PAFSV processes in terms of the TTS, and define the following transition relations on processes of PAFSV:

- $\_ \xrightarrow{\_} \langle \checkmark, \_ \rangle \subseteq (P \times \Sigma) \times A_\tau \times \Sigma$, denotes termination, where $\checkmark$ is used to indicate a successful termination, and $\checkmark$ is not a process term;
- $\_ \xrightarrow{\_} \_ \subseteq (P \times \Sigma) \times A_\tau \times (P \times \Sigma)$, denotes action transition;
- $\_ \overset{\_}{\longmapsto} \_ \subseteq (P \times \Sigma) \times \mathbb{R}_{\geq 0} \times (P \times \Sigma)$, denotes time transition (so-called delay).

For $p, p' \in P$; $\sigma, \sigma' \in \Sigma$, $a \in A_\tau$ and $d \in \mathbb{R}_{\geq 0}$, the three kinds of transition relations can be explained as follows:

1. Firstly, a termination $\langle p, \sigma \rangle \xrightarrow{a} \langle \checkmark, \sigma' \rangle$ is that the process executes the action a followed by termination.

2. Secondly, an action transition $\langle p, \sigma\rangle \xrightarrow{a} \langle p', \sigma'\rangle$ is that the process $\langle p, \sigma\rangle$ executes the action a starting with the current valuation σ and by this execution p evolves into p', where σ' represents the accompanying valuation of the process after the action a is executed.
3. Thirdly, a time transition $\langle p, \sigma\rangle \xmapsto{d} \langle p', \sigma'\rangle$ is that the process $\langle p, \sigma\rangle$ may idle during a d time units and then behaves like $\langle p', \sigma'\rangle$.

3.7 Deduction Rules

The above transition relations are defined through deduction rules (SOS style). These rules (of the form $\frac{\textit{permises}}{\textit{conclusions}}$) have two parts: on the top of the bar we put *premises* of the rule, and below it the *conclusions*. If the premises hold, then we infer that the conclusion (can be more than one) holds as well. In case there is no premise, the deduction rule becomes an axiom.

Apart from the syntax restriction as already shown in Section 3.3 (e.g. $x, w \in \text{Var}$), for all deduction rules, we further require that $p, q, p', q' \in P$; $\sigma, \sigma', \sigma'' \in \Sigma$; $a, b \in A_\tau$, $d \in \mathbb{R}_{\geq 0}$, $\text{dom}(\sigma) = \text{dom}(\sigma') = \text{dom}(\sigma'')$; σ, σ', σ'' and $\bar{\sigma}(e)$ are defined, where the notation $\bar{\sigma}(e)$ is used to represent the value of expression e in σ.

Also, we make use of the sets of variables $\text{Var}^- = \{x^- \mid x \in \text{Var}\}$ and $\text{Var}^+ = \{x^+ \mid x \in \text{Var}\}$, modelling the current and future value of a variable, respectively. Similarly, e^- and e^+ are used to represent the current and future value of e respectively. Furthermore, in order to increase the readability of the PAFSV deduction rules, two abbreviations may be used:

1. In case the deduction rules are defined in a similar way for both action transition and time transition, the notation $\overset{z}{\twoheadrightarrow}$ is used as a short-hand for $\xrightarrow{a}$ and $\xmapsto{d}$.
2. $\Theta \in \{\checkmark, \hbar \mid \hbar = p' \in P \vee \hbar = q' \in P\}$.

We want to state that it is not our intention to define deduction rules for all inductive cases for all operators in this paper. For simplicity, only relevant deduction rules for the use of this paper are shown in this subsection.

3.7.1 Skip

$$\frac{}{\langle \mathbf{skip}, \sigma\rangle \xrightarrow{\tau} \langle \checkmark, \sigma\rangle} \tag{1}$$

Rule (1) states that the process term **skip** performs the τ action followed by termination, and has no effect on the valuation.

3.7.2 Procedural Assignment

$$\frac{}{\langle x := e, \sigma\rangle \xrightarrow{aa(x,\bar{\sigma}(e))} \langle \checkmark, \sigma[\bar{\sigma}(e)/x]\rangle} \tag{2}$$

By means of a procedural assignment (see Rule (2)), the value of e is assigned to x. Notice that $\sigma[\bar{\sigma}(e)/x]$ denotes the update of valuation σ such that the new value of variable x is $\bar{\sigma}(e)$.

3.7.3 Delay

$$\frac{n = 0}{\langle \mathbf{delay}(n), \sigma \rangle \xrightarrow{\tau} \langle \checkmark, \sigma \rangle} \tag{3}$$

$$\frac{d \leq n}{\langle \mathbf{delay}(n), \sigma \rangle \overset{d}{\longmapsto} \langle \mathbf{delay}(n - d), \sigma \rangle} \tag{4}$$

Rule (3) is used to model a delay of 0 duration by means of performing the internal τ action. Rule (4) states that a delay process term can perform a delay which is smaller than or equal to the value of the argument of the delay process term, and has no effect on the valuation.

3.7.4 Any

$$\frac{\langle p, \sigma \rangle \xrightarrow{a} \langle \Theta, \sigma' \rangle}{\langle \mathbf{any}\ p, \sigma \rangle \xrightarrow{a} \langle \Theta, \sigma' \rangle} \tag{5}$$

$$\frac{t \in \mathbb{R}_{>0}}{\langle \mathbf{any}\ p, \sigma \rangle \overset{t}{\longmapsto} \langle \mathbf{any}\ p, \sigma \rangle} \tag{6}$$

Rule (5) shows that the any operator does not affect the action behaviour of p. The any operator allows arbitrary time transitions for p, as seen in Rule (6).

3.7.5 If_else

$$\frac{\langle p, \sigma \rangle \xrightarrow{a} \langle \Theta, \sigma' \rangle, \sigma \models b}{\langle \mathbf{if}(b)\ p\ \mathbf{else}\ q, \sigma \rangle \xrightarrow{a} \langle \Theta, \sigma' \rangle} \tag{7}$$

$$\frac{\langle q, \sigma \rangle \xrightarrow{a} \langle \Theta, \sigma' \rangle, \sigma \models \neg b}{\langle \mathbf{if}(b)\ p\ \mathbf{else}\ q, \sigma \rangle \xrightarrow{a} \langle \Theta, \sigma' \rangle} \tag{8}$$

$$\frac{\langle p, \sigma \rangle \overset{d}{\longmapsto} \langle p', \sigma' \rangle, \sigma \models b}{\langle \mathbf{if}(b)\ p\ \mathbf{else}\ q, \sigma \rangle \overset{d}{\longmapsto} \langle p', \sigma' \rangle} \tag{9}$$

$$\frac{\langle q, \sigma \rangle \overset{d}{\longmapsto} \langle q', \sigma' \rangle, \sigma \models \neg b}{\langle \mathbf{if}(b)\ p\ \mathbf{else}\ q, \sigma \rangle \overset{d}{\longmapsto} \langle q', \sigma' \rangle} \tag{10}$$

If b evaluates to *true* in σ (denoted by $\sigma \models b$), the if_else process term ($\mathbf{if}\,(b)\,p$ **else** q) behaves as process term p. If b evaluates to *false* (denoted by $\sigma \models \neg b$), then it behaves as process term q (see from Rule (7) to Rule (10)).

3.7.6 Sequential Composition

$$\frac{\langle p, \sigma\rangle \xrightarrow{a} \langle \checkmark, \sigma'\rangle}{\langle p;\ q, \sigma\rangle \xrightarrow{a} \langle q, \sigma'\rangle} \tag{11}$$

$$\frac{\langle p, \sigma\rangle \overset{z}{\twoheadrightarrow} \langle p', \sigma'\rangle}{\langle p;\ q, \sigma\rangle \overset{z}{\twoheadrightarrow} \langle p';\ q, \sigma'\rangle} \tag{12}$$

The process term q is executed after (successful) termination of the process term p, as defined by Rules (11) and (12).

3.7.7 Wait

$$\frac{\langle p, \sigma\rangle \xrightarrow{a} \langle \Theta, \sigma'\rangle,\ \sigma \models b}{\langle \mathbf{wait}(b)\ p, \sigma\rangle \xrightarrow{a} \langle \Theta, \sigma'\rangle} \tag{13}$$

3[mm]

$$\frac{\langle p, \sigma\rangle \xmapsto{d} \langle p', \sigma'\rangle,\ \sigma \models b}{\langle \mathbf{wait}(b)\ p, \sigma\rangle \xmapsto{d} \langle p', \sigma'\rangle} \tag{14}$$

$$\frac{\sigma \models \neg b}{\langle \mathbf{wait}(b)\ p, \sigma\rangle \xrightarrow{\tau} \langle \mathbf{any}(\mathbf{wait}(b)\ p), \sigma\rangle} \tag{15}$$

If b evaluates to *true* in σ, the wait process term $\mathbf{wait}(b)\ p$ behaves as process term p, as defined by Rules (13) and (14). Otherwise, it is blocked until b becomes *true* (see Rule (15)).

3.7.8 While

$$\frac{\langle p, \sigma\rangle \xrightarrow{a} \langle \checkmark, \sigma'\rangle,\ \sigma \models b}{\langle \mathbf{while}(b)\ p, \sigma\rangle \xrightarrow{a} \langle \mathbf{while}(b)\ p, \sigma'\rangle} \tag{16}$$

$$\frac{\langle p, \sigma\rangle \overset{z}{\twoheadrightarrow} \langle p', \sigma'\rangle,\ \sigma \models b}{\langle \mathbf{while}(b)\ p, \sigma\rangle \overset{z}{\twoheadrightarrow} \langle p';\ \mathbf{while}(b)\ p, \sigma'\rangle} \tag{17}$$

$$\frac{\sigma \models \neg b}{\langle \mathbf{while}(b)\ p, \sigma\rangle \xrightarrow{\tau} \langle \checkmark, \sigma\rangle} \tag{18}$$

If the condition $\sigma \models b$ holds and process term p terminates, the while process term $\mathbf{while}(b)\ p$ restarts, as shown in Rule (16). If the condition $\sigma \models b$ holds in Rule (17) and p can perform an action or a time transition, the while process term **while**(b) p behaves as the sequential composition of the resulting process term after performing such a transition (i.e. p') and **while**(b) p (i.e. p'; **while**(b) p). If b evaluates to *false* in σ, the while process term **while**(b) p terminates immediately by means of the internal action τ (see Rule (18)).

3.7.9 Continuous Assignment

$$\frac{e^- \neq e^+}{\begin{array}{c}\langle \textbf{assign } w := e, \sigma\rangle \xrightarrow{aa(w,\bar{\sigma}(e))} \\ \langle \textbf{assign } w := e, \sigma[\bar{\sigma}(e)/x]\rangle\end{array}} \tag{19}$$

$$\frac{e^- = e^+}{\langle \textbf{assign } w := e, \sigma\rangle \xrightarrow{\tau} \langle \textbf{any}(\textbf{assign } w := e), \sigma\rangle} \tag{20}$$

If the condition $e^- \neq e^+$ in Rule (19) holds, the value of e is assigned to w. Rule (20) is similar to Rule (15).

3.7.10 Event

$$\frac{\langle p, \sigma\rangle \xrightarrow{a} \langle \checkmark, \sigma'\rangle,\ \eta_1(l_1) \vee \ldots \vee \eta_n(l_n)}{\langle @_{(\eta_1(l_1),\ldots,\eta_n(l_n))}\ p\rangle \xrightarrow{a} \langle \checkmark, \sigma'\rangle} \tag{21}$$

$$\frac{\langle p, \sigma\rangle \overset{z}{\twoheadrightarrow} \langle p', \sigma'\rangle,\ \eta_1(l_1) \vee \ldots \vee \eta_n(l_n)}{\langle @_{(\eta_1(l_1),\ldots,\eta_n(l_n))}\ p, \sigma\rangle \overset{z}{\twoheadrightarrow} \langle p', \sigma'\rangle} \tag{22}$$

$$\frac{\neg(\eta_1(l_1) \vee \ldots \vee \eta_n(l_n))}{\begin{array}{c}\langle @_{(\eta_1(l_1),\ldots,\eta_n(l_n))}\ p, \sigma\rangle \xrightarrow{\tau} \\ \langle \textbf{any}\ (@_{(\eta_1(l_1),\ldots,\eta_n(l_n))}\ p), \sigma\rangle\end{array}} \tag{23}$$

In case that $\eta_1(l_1) \vee \ldots \vee \eta_n(l_n)$ evaluates to *true*, process term $@_{(\eta_1(l_1),\ldots,\eta_n(l_n))}\ p$ can perform whatever p can do, as shown in Rules (21) and (22). The intuition behind Rule (23) is similar to Rule (20).

Note that $\eta_1, \ldots, \eta_n$ are Boolean functions of the form of η_* that are defined to indicate a change in variables $l_1, \ldots, l_n \in$ Var. For $k \in$ Var and $* \in \{sentitive, posedge, negedge\}$, η_* is defined as follows:

1. Indicating a level change of the value in k,

$$\eta_{sensitive}(k) = \left\{ \begin{array}{ll} true, & \text{if } k^- \neq k^+ \\ false, & \text{otherwise} \end{array} \right\}.$$

2. Indicating a positive change of the value in k,

$$\eta_{posedge}(k) = \left\{ \begin{array}{ll} \mathit{true}, & \text{if } k^- < k^+ \\ \mathit{false}, & \text{otherwise} \end{array} \right\}.$$

3. Indicating a negative change of the value in k,

$$\eta_{negedge}(k) = \left\{ \begin{array}{ll} \mathit{true}, & \text{if } k^- > k^+ \\ \mathit{false}, & \text{otherwise} \end{array} \right\}.$$

3.7.11 Alternative Composition

$$\frac{\langle p, \sigma \rangle \xrightarrow{a} \langle \checkmark, \sigma' \rangle}{\langle p \circledast q, \sigma \rangle \xrightarrow{a} \langle \checkmark, \sigma' \rangle} \tag{24}$$

$$\frac{\langle q, \sigma \rangle \xrightarrow{a} \langle \checkmark, \sigma' \rangle}{\langle p \circledast q, \sigma \rangle \xrightarrow{a} \langle \checkmark, \sigma' \rangle} \tag{25}$$

$$\frac{\langle p, \sigma \rangle \xrightarrow{a} \langle p', \sigma' \rangle}{\langle p \circledast q, \sigma \rangle \xrightarrow{a} \langle p', \sigma' \rangle} \tag{26}$$

$$\frac{\langle q, \sigma \rangle \xrightarrow{a} \langle q', \sigma' \rangle}{\langle p \circledast q, \sigma \rangle \xrightarrow{a} \langle q', \sigma' \rangle} \tag{27}$$

$$\frac{\langle p, \sigma \rangle \overset{d}{\longmapsto} \langle p', \sigma' \rangle,\ \langle q, \sigma \rangle \overset{d}{\longmapsto} \langle q', \sigma' \rangle}{\langle p \circledast q, \sigma \rangle \overset{d}{\longmapsto} \langle p' \circledast q', \sigma' \rangle} \tag{28}$$

The effect of applying the alternative composition to process terms p and q (i.e. $p \circledast q$) is that the execution of a termination or an action transition by either one of them results in a definite choice as shown from Rule (24) to Rule (27).

With respect to time transition, process terms p and q have to synchronise, as stated in Rule (28).

3.7.12 Parallel Composition

$$\frac{\langle p, \sigma \rangle \xrightarrow{a} \langle \checkmark, \sigma' \rangle}{\langle p \parallel q, \sigma \rangle \xrightarrow{a} \langle q, \sigma' \rangle} \tag{29}$$

$$\frac{\langle q, \sigma \rangle \xrightarrow{a} \langle \checkmark, \sigma' \rangle}{\langle p \parallel q, \sigma \rangle \xrightarrow{a} \langle p, \sigma' \rangle} \tag{30}$$

$$\frac{\langle p, \sigma \rangle \xrightarrow{a} \langle p', \sigma' \rangle}{\langle p \parallel q, \sigma \rangle \xrightarrow{a} \langle p' \parallel q, \sigma' \rangle} \tag{31}$$

$$\frac{\langle q, \sigma \rangle \xrightarrow{a} \langle q', \sigma' \rangle}{\langle p \parallel q, \sigma \rangle \xrightarrow{a} \langle p \parallel q', \sigma' \rangle} \tag{32}$$

$$\frac{\langle p, \sigma \rangle \overset{d}{\longmapsto} \langle p', \sigma' \rangle,\ \langle q, \sigma \rangle \overset{d}{\longmapsto} \langle q', \sigma' \rangle}{\langle p \parallel q, \sigma \rangle \overset{d}{\longmapsto} \langle p' \parallel q', \sigma' \rangle} \tag{33}$$

The parallel composition of the process terms p and q (i.e. $p \parallel q$) has as its behaviour with respect to action transitions the interleaving of the behaviours of process terms p and q (see Rules from (29) to (32)). If both process terms p and q can perform the same delay, then the parallel composition of process terms p and q (i.e. $p \parallel q$) can also perform that delay, as defined by Rule (33).

3.7.13 Repeat

$$\frac{\langle p, \sigma \rangle \xrightarrow{a} \langle \checkmark, \sigma' \rangle}{\langle \textbf{repeat}\ p, \sigma \rangle \xrightarrow{a} \langle \textbf{repeat}\ p, \sigma' \rangle} \tag{34}$$

$$\frac{\langle p, \sigma \rangle \overset{z}{\twoheadrightarrow} \langle p', \sigma' \rangle}{\langle \textbf{repeat}\ p, \sigma \rangle \overset{z}{\twoheadrightarrow} \langle p'; \textbf{repeat}\ p, \sigma' \rangle} \tag{35}$$

If the argument (i.e. p) of a repeat process term **repeat** p can perform a transition to termination, the repeat process term **repeat** p just repeats itself after performing such a transition, as shown in Rule (34). Rule (35) states that if the process term p can perform an action transition or time transition, then the repeat process term **repeat** p can also perform that action transition or time transition followed by its repetition (i.e. p'; **repeat** p).

3.7.14 Immediate Assertion

$$\frac{\langle p, \sigma\rangle \xrightarrow{a} \langle \Theta, \sigma'\rangle,\ \sigma \models b}{\langle \mathbf{assert}(b)\ p, \sigma\rangle \xrightarrow{a} \langle \Theta, \sigma'\rangle} \tag{36}$$

$$\frac{\langle p, \sigma\rangle \overset{d}{\longmapsto} \langle p', \sigma'\rangle,\ \sigma \models b}{\langle \mathbf{assert}(b)\ p, \sigma\rangle \overset{d}{\longmapsto} \langle p', \sigma'\rangle} \tag{37}$$

$$\frac{\sigma \models \neg b}{\langle \mathbf{assert}(b)\ p, \sigma\rangle \xrightarrow{\tau} \langle \checkmark, \sigma\rangle} \tag{38}$$

If the property (expressed as a Boolean expression) holds in Rule (36) (i.e. $\sigma \models b$), then p is executed. Rule (37) is similar to Rule (36). If the property does not hold, as stated in Rule (38), it terminates immediately by means of the internal action τ and no effect on the valuation.

Remark 1. In SystemVerilog, an immediate assertion statement is a test of an expression performed when the statement is executed. The expression is non-temporal and is interpreted the same way as an expression in the condition of an **if** statement. This means that the semantics of "immediate assertion" in PAFSV can also be defined in terms of other language elements in PAFSV. Although defining deduction rules for SystemVerilog immediate assertion statement in PAFSV is not very useful, in order to allow more intuitive specifications in PAFSV (as in the example provided in Section 6.3), such deduction rules are given here.

3.7.15 Disrupt

$$\frac{\langle p, \sigma\rangle \xrightarrow{a} \langle \checkmark, \sigma'\rangle}{\langle p\ \mathbf{disrupt}\ q, \sigma\rangle \xrightarrow{a} \langle \checkmark, \sigma'\rangle} \tag{39}$$

$$\frac{\langle q, \sigma\rangle \xrightarrow{a} \langle \checkmark, \sigma'\rangle}{\langle p\ \mathbf{disrupt}\ q, \sigma\rangle \xrightarrow{a} \langle \checkmark, \sigma'\rangle} \tag{40}$$

$$\frac{\langle p, \sigma\rangle \xrightarrow{a} \langle p', \sigma'\rangle, \langle q, \sigma\rangle \xrightarrow{b} \langle q', \sigma''\rangle, a \neq b,\ a \neq \tau}{\langle p\ \mathbf{disrupt}\ q, \sigma\rangle \xrightarrow{a} \langle p'\ \mathbf{disrupt}\ q', \Omega(\sigma, \sigma', \sigma'')\rangle} \tag{41}$$

$$\frac{\langle p, \sigma\rangle \overset{d}{\longmapsto} \langle p', \sigma'\rangle}{\langle p\ \mathbf{disrupt}\ q, \sigma\rangle \overset{d}{\longmapsto} \langle p'\ \mathbf{disrupt}\ q, \sigma'\rangle} \tag{42}$$

Rules (39) and (40) show that either process term p or process term q terminates, and the disrupt process term p **disrupt** q terminates as well. If process terms p and q can perform two different action transitions, the disrupt process term p **disrupt** q

gives a higher priority to the action transition performed by p than the action transition performed by q, and the valuation associating to the disrupt process after performing the action transition (as p performed) is the merge of the valuations after the execution of (two different) action transitions performed by p and q, as defined by Rule (41). The disrupt process term p **disrupt** q can perform whatever delays p can perform regardless the delay behaviour of q, as shown in Rule (42).

Clearly, $a \neq b$ and $a \neq \tau$, as stated in Rule (41), for $\sigma, \sigma', \sigma'' \in \Sigma$, $\Omega(\sigma, \sigma', \sigma'')$ is formally defined as follows:

$$\Omega(\sigma,\sigma',\sigma'') = \sigma' \upharpoonright \{x \mid x \in \mathrm{dom}(\sigma), \sigma(x) \neq \sigma'(x)\} \cup \sigma'' \upharpoonright \{x \mid x \in \mathrm{dom}(\sigma), \sigma(x) \neq \sigma''(x)\}.$$

Note that $\upharpoonright$ is the restriction operator on function, which is formally defined below: If f is a function, $\mathrm{dom}(f)$ and $\mathrm{range}(f)$ denote the domain and range of f, respectively. If S is a set, $f \upharpoonright S$ denotes the restriction of f to S, that is, the function g with $\mathrm{dom}(g) = \mathrm{dom}(f) \cap S$, such that $g(c) = f(c)$ for each $c \in \mathrm{dom}(g)$.

3.8 Non-Primitive Statements

For illustration purposes, the interpretation of some SystemVerilog non-primitive statements and constructs in PAFSV, through simple examples, is given below.

3.8.1 Non-Blocking Assignment

In SystemVerilog,

```
begin
  #3 b <= a;
  #6 x <= c;
end
```

In PAFSV,

$$\textbf{delay}(3);\ b := a \parallel \textbf{delay}(6);\ x := c.$$

In both cases, the value of a (at `time` $= 0$) is assigned to b at `time` $= 3$ and the value of c (at `time` $= 0$) is assigned to x at `time` $= 6$. A more tricky case in SystemVerilog,

```
begin
  a = 67;
  # 10;
  a <= 4;
  c <= #15 a;
  d <= #10 9;
  b <= 3;
end
```

In PAFSV,

$a := 67;\ a' := a;\ \mathbf{delay}(10);\ (a := 4 \parallel \mathbf{delay}(15);\ c := a' \parallel \mathbf{delay}(10);\ d := 9 \parallel b := 3).$

In both cases, $a = 4$ and $b = 3$ (at `time` $= 10$), $d = 9$ (at `time` $= 20$) and $c = 67$ (at `time` $= 10 + 15$, the value of a' is assigned to c).

Remark 2. In the PAFSV specification, variable a' is introduced (as a copy of a) to save the temporary value of a. Similar things can be seen in the non-blocking assignments in the SystemVerilog design. The right-hand side variables of the non-blocking assignments are read and stored in temporary memory locations. As we can see, it is not difficult to give formal specification of SystemVerilog non-blocking assignment (with the help of modelling technique) in PAFSV using the formal syntax presented in Section 3.3. Therefore, deduction rules for non-blocking assignment are not specifically defined in this paper.

3.8.2 Fork_join

In SystemVerilog,

```
begin
  c = 0;
  #5;
  fork
    #5  a = 0;
    #10 b = 0;
  join
  c =1;
end
```

In PAFSV,

$$c := 0;\ \mathbf{delay}(5);\ (\mathbf{delay}(5);\ a := 0 \parallel \mathbf{delay}(10);\ b := 0);\ c = 1.$$

Note that c becomes 1 when `time` $= 15$ for both cases.

3.8.3 Priority

In SystemVerilog,

```
priority if (a == 0) y = in1;
    else if (a == 1) y = in2;
    else y = in3;
```

In PAFSV,

$$\mathbf{if}(a = 0)\ y := in1\ \mathbf{else}\ (\mathbf{if}(a = 1)\ y := in2\ \mathbf{else}\ (y := in3)).$$

Both maintain the decision order (as specified) for execution/transition.

3.8.4 Unique

In SystemVerilog,

```
unique case(a)
   0 : y = in1;
   1 : y = in2;
endcase
```

In PAFSV,

$$\textbf{if}(a = 0 \wedge \neg(a = 0 \wedge a = 1))\ y := in1\ \textbf{else deadlock} \circledast$$

$$\textbf{if}(a = 1 \wedge \neg(a = 0 \wedge a = 1))\ y := in2\ \textbf{else deadlock} \circledast$$

$$\textbf{if} \neg(a = 0 \vee a = 1)\ \textbf{skip else deadlock}.$$

Both ensure that the decisions are mutually exclusive.

4 EXAMPLES

Deduction rules offer preciseness, because they come with a mathematically defined semantics. Formal specifications can be analysed using deduction rules providing an absolute notion of correctness. Also, these deduction rules can ensure the correctness of PAFSV specifications and can help modellers to make correct specifications.

In order to demonstrate the effectiveness and applicability of the deduction rules, two toy specifications in PAFSV are given in this section. These specifications also show how (illustrated by means of transition traces) process evolves during transitions. Using the deduction rules, for instance, we can show that:

1. the process $\langle x := 5;\ y := 7, \{x \mapsto 0, y \mapsto 1\}\rangle$ can terminate successfully after a finite number of transitions.

   Transition traces: According to Rule (2), the process $\langle x := 5, \{x \mapsto 0, y \mapsto 1\}\rangle$ can always perform an assignment action to a terminated process as follows: $\langle x := 5, \{x \mapsto 0, y \mapsto 1\}\rangle \xrightarrow{aa(x,5)} \langle \checkmark, \{x \mapsto 5, y \mapsto 1\}\rangle$. Due to this, we can apply Rule (12) to obtain $\langle x := 5;\ y := 7, \{x \mapsto 0, y \mapsto 1\}\rangle \xrightarrow{aa(x,5)} \langle y := 7, \{x \mapsto 5, y \mapsto 1\}\rangle$. Applying Rule (2) again, we have $\langle y := 7, \{x \mapsto 5, y \mapsto 1\}\rangle \xrightarrow{aa(y,7)} \langle \checkmark, \{x \mapsto 5, y \mapsto 7\}\rangle$.

2. the process $\langle (x := 1 \parallel y := 2);\ z := 3, \sigma\rangle$ cannot terminate successfully in two transitions.

   Semantical proof: We assume to have $\langle (x := 1 \parallel y := 2);\ z := 3, \sigma\rangle \xrightarrow{a} \langle z := 3, \sigma'\rangle$ for some a and σ' in such a way that the process can terminate successfully in two transitions. This means that we must have the action transition

$\langle x := 1 \parallel y := 2, \sigma\rangle \xrightarrow{a} \langle \checkmark, \sigma'\rangle$ as a premise necessarily for Rule (11). However, this is not possible due to Rules (29) and (30).

5 VALIDATION OF THE SEMANTICS

This section first shows that the term deduction system of PAFSV is well-defined. Then a notion of equivalence called *Stateless Bisimilarity* is defined [4, 16]. It is also shown that this relation is an equivalence and a *Congruence* [1] (which also means that compositionality preserved operationally in PAFSV) for all PAFSV operators.

A set of useful properties of PAFSV is sound with respect to the stateless bisimilarity that is also introduced.

5.1 Well-Definedness of the Semantics

The deduction rules defined in Section 3.7 constitute a *Transition System Specification* (TSS) as described in [17]. The transitions that can be proven from a TSS define TTS.

The TTS of PAFSV contains terminations, action transitions and time transitions that can be proven from the deduction rules. In general, TSSs with negative premises[1] might not be *meaningful* (see [17] for details).

Well-definedness of the term deduction system can be obtained by providing a *stratification* [18]. A stratification is a metric on formulas that, for each deduction rule of the TTS, does not increase from conclusion to all positive premises and strictly decreases from conclusion to negative premises.

We define the mapping that associates the value 0 with every positive termination and action transition and the value 1 with every positive time transition. Then, it is not hard to see that the PAFSV deduction rules of the TTS are stratifiable (note that no negative premise is used in our deduction rules for PAFSV). This also means that the system defines a unique transition system for each closed process term of PAFSV.

5.2 Bisimilarity

Two closed PAFSV process terms are considered equivalent if they have the same behaviour (in the bisimulation sense) in case both are considered, from the current state and the valuation of variables. We also assume that the valuation (of the current state) contains at least free occurrences of variables in the two closed PAFSV process terms being equivalent.

Definition 3 (Stateless bisimilarity)**.** A stateless bisimilarity on closed process terms is a relation $R \subseteq P \times P$ such that $\forall (p, q) \in R$, the following holds:

[1] We write a negative premise for action transition as $\langle p, \sigma\rangle \overset{a}{\nrightarrow}$ for the set of all transitions formulas $\neg(\langle p, \sigma\rangle \xrightarrow{a} \langle p', \sigma'\rangle)$, where $p, p' \in P$, $a \in A_\tau$ and $\sigma, \sigma' \in \Sigma$. In a similar way, we can define negative premises for termination and time transition.

1. $\forall \sigma, a, \sigma' : \langle p, \sigma\rangle \xrightarrow{a} \langle \checkmark, \sigma'\rangle \Leftrightarrow \langle q, \sigma\rangle \xrightarrow{a} \langle \checkmark, \sigma'\rangle$,
2. $\forall \sigma, a, p', \sigma' : \langle p, \sigma\rangle \xrightarrow{a} \langle p', \sigma'\rangle \Rightarrow \exists q' : \langle q, \sigma\rangle \xrightarrow{a} \langle q', \sigma'\rangle \wedge (p', q') \in R$,
3. $\forall \sigma, a, q', \sigma' : \langle q, \sigma\rangle \xrightarrow{a} \langle q', \sigma'\rangle \Rightarrow \exists p' : \langle p, \sigma\rangle \xrightarrow{a} \langle p', \sigma'\rangle \wedge (p', q') \in R$,
4. $\forall \sigma, d, p', \sigma' : \langle p, \sigma\rangle \overset{d}{\mapsto} \langle p', \sigma'\rangle \Rightarrow \exists q' : \langle q, \sigma\rangle \overset{d}{\mapsto} \langle q', \sigma'\rangle \wedge (p', q') \in R$,
5. $\forall \sigma, d, q', \sigma' : \langle q, \sigma\rangle \overset{d}{\mapsto} \langle q', \sigma'\rangle \Rightarrow \exists p' : \langle p, \sigma\rangle \overset{d}{\mapsto} \langle p', \sigma'\rangle \wedge (p', q') \in R$.

Two closed process terms p and q are stateless bisimilar, denoted by $p \underline{\leftrightarrow} q$, if there exists a stateless bisimilarity relation R such that $(p, q) \in R$.

Stateless bisimilarity is proved to be a congruence with respect to all operators PAFSV operators. As a consequence, algebraic reasoning is facilitated, since it is allowed to replace equals by equals in any context.

Theorem 1 (Congruence)**.** Stateless bisimilarity is a congruence with respect to all operators of PAFSV.

Proof. All deduction rules of PAFSV are in the process-tyft format of [16]. It follows from [16] that stateless bisimilarity is a congruence. □

5.3 Properties

In this subsection, some properties of the operators of PAFSV that hold with respect to stateless bisimilarity are discussed. Most of these correspond well with our intuitions, and hence this can be considered as an additional validation of the semantics. But it is not our intention to provide a complete list of such properties (complete in the sense that every equivalence between closed process terms is derivable from those properties).

Proposition 1 (Properties)**.** A set of properties is introduced for PAFSV described in this paper for $p, q, r \in P$. These properties are sound with respect to the stateless bisimilarity.

1. **skip** $\underline{\leftrightarrow}$ **delay**(0),
2. **deadlock**; $p \underline{\leftrightarrow}$ **deadlock**,
3. $(p;\ q);\ r \underline{\leftrightarrow} p;\ (q;\ r)$,
4. **any** $p;\ q \underline{\leftrightarrow}$ **any** $(p;\ q)$,
5. $p \circledast q \underline{\leftrightarrow} q \circledast p$,
6. $(p \circledast q);\ r \underline{\leftrightarrow} p;\ r \circledast q;\ r$,
7. $(p \circledast q) \circledast r \underline{\leftrightarrow} p \circledast (q \circledast r)$,
8. $p \parallel q \underline{\leftrightarrow} q \parallel p$,
9. $(p \parallel q) \parallel r \underline{\leftrightarrow} p \parallel (q \parallel r)$,
10. **any** $p \circledast$ **any** $q \underline{\leftrightarrow}$ **any** $(p \circledast q)$,

11. $p \parallel q \underline{\leftrightarrow} p;\ q \circledast q;\ p$.

Proof. We leave out the proofs, because most of the proofs are proofs for distributivity, commutativity and associativity as in classical process algebras. Similar proofs can also be found in [4]. □

The intuition of the above properties is as follows:

- Since **skip** and **delay**(0) can only perform the internal action τ to termination, both process terms are equivalent.
- A deadlock process term followed by some other process terms is equivalent to the **deadlock** itself because the deadlock process term does not terminate successfully, i.e. **deadlock** is a left-zero element for sequential composition.
- Sequential composition is associative.
- Any operator distributes the argument of a sequential composition to the right.
- Alternative composition and parallel composition are commutative and associative.
- Alternative composition distributes over sequential composition from the left, but not from the right.
- Any operator distributes over the alternative composition.
- Parallel composition can be eliminated by means of sequential composition and alternative composition.

6 EXAMPLES OF PAFSV SPECIFICATIONS

This section shows samples of the application of PAFSV, i.e., to give a first impression of how one can describe the behaviour of a SystemVerilog design in a complete mathematical sense using PAFSV. In order to illustrate our work clearly, only simple examples were given in this paper. Nevertheless, the use of PAFSV is generally applicable to all sizes and levels of hardware systems. We describe the behaviour of a multiplexer and a simple synchronous reset D flip-flop. Then, we formally analyse a simple arbiter.

6.1 A Simple Modelling of a Multiplexer

In electronic designs, a multiplexer (MUX) is a device that encodes information from two or more data inputs into a single output (i.e. multiplexers function as multiple-inputs and single-output switches).

A multiplexer described below (in SystemVerilog) has two inputs and a selector that connects a specific input to the single output. Figure 1 depicts such MUX.

```
module  simple_mux (
input  wire  a,
```

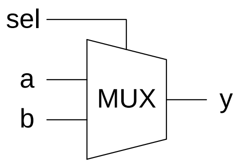


Figure 1. MUX

```
input  wire  b,
input  wire  sel,
output wire  y
);

assign y = (sel) ? a : b;

endmodule
```

The formal PAFSV specification (as a process term) below can be regarded as the (formal) mathematical expression of the above multiplexer (described as a SystemVerilog module):

$$\mathbf{if}(sel)\ y := a\ \mathbf{else}\ y := b$$

Needless to say that, in SystemVerilog, the conditional operator "(condition) ? (result if true) : (result if false)" can be considered as an **if**(_)_**else**_ statement. In the PAFSV specification, an if_else process term is used to model the behaviour of such a MUX.

6.2 A Simple Modelling of a Synchronous Reset D Flip-Flop

Synchronous reset D flip-flops are among the basic building blocks of RTL designs. A synchronous reset D flip-flop has a clock input (*clk*) in the event list, a data input (*d*), a reset (*rst*) and a data output (*Q*). Figure 2 depicts such a synchronous reset D flip-flop.

A synchronous reset D flip-flop described below (as a module in SystemVerilog) is inferred by using posedge clause for the clock *clk* in the event list.

```
module dff_sync_reset (
input  wire d,
input  wire clk,
input  wire rst,
output reg  Q
);
```

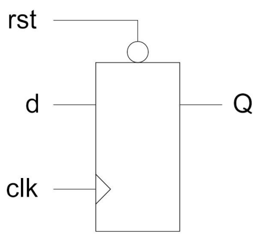


Figure 2. A synchronous reset D flip-flop

```
always_ff @ (posedge clk)
if (~reset) begin
  Q = 1'b0;
end  else begin
  Q = d;
end

endmodule
```

The formal PAFSV specification (as a process term) of the above synchronous reset D flip flop is given as follows:

$$\begin{aligned} \text{DFF} &\approx \textbf{repeat}(@_{(\eta_{posedge}(clk))}\text{OUT}) \\ \text{OUT} &\approx \textbf{if}(\neg rst)\ Q := \mathit{1'b0}\ \textbf{else}\ Q := d \end{aligned}$$

In the PAFSV specification (i.e. process term DFF), the behaviour of the synchronous reset D flip-flop is modelled by means of the if_else process term using "$\neg rst$ (active low reset)" as the condition of such a process term.

This if_else process term is further triggered repeatedly by the event process term, which is positively sensitive to the clock (i.e. *clk*).

6.3 A Detailed Analysis of an Arbiter

Arbiter circuits are standard digital hardware verification benchmark circuits. In general, the role of an arbiter is to grant access to the shared resource by raising the corresponding *grant* signal and keeping it that way until the *request* signal is removed.

A test for the arbiter can be generated by an immediate assertion as follows:

$$\text{“}assertion : grant \wedge request\text{”}.$$

This test can be considered as a "*liveness property*" of the arbiter. If the assertion holds, this means that the arbiter works as expected. Below is the SystemVerilog description of the simple arbiter:

```
module assert_immediate();
reg clk, grant, request;
time current_time;
initial begin
   clk = 0;
   grant = 0;
   request = 0;
   #4 request = 1;
   #4 grant = 1;
   #4 request = 0;
   #4 $finish;
end
always #5 clk = ~ clk;
always @ (negedge clk)
begin
if (grant == 1) begin
 CHECK_REQ_WHEN_GNT:
   assert(grant && request) begin
   current_time = $time;
    $display {``Works as expected'');
    end
 end
endmodule
```

A formal PAFSV specification of the above SystemVerilog arbiter is given as follows:

$\langle$ INIT $\|$ ARB $\|$ CLK **disrupt** ASSER, σ $\rangle$,
where

$$\begin{aligned}
\mathrm{INIT} &\approx \mathit{clk} := 0;\ \mathit{grant} := 0;\ \mathit{request} := 0\\
\mathrm{ARB} &\approx \mathrm{R}_1;\ \mathrm{G};\ \mathrm{R}_0;\ \mathrm{S}\\
\mathrm{R}_1 &\approx \mathbf{delay}(4);\ \mathit{request} := 1\\
\mathrm{G} &\approx \mathbf{delay}(4);\ \mathit{grant} := 1\\
\mathrm{R}_0 &\approx \mathbf{delay}(4);\ \mathit{request} := 0\\
\mathrm{S} &\approx \mathbf{delay}(4);\ \mathbf{skip}\\
\mathrm{CLK} &\approx \mathbf{repeat}(\mathbf{delay}(5);\ \mathit{clk} := \neg \mathit{clk})\\
\mathrm{ASSER} &\approx \mathbf{repeat}(@_{(\eta_{negedge}(clk))}\mathrm{PROP};\ \mathbf{skip})\\
\mathrm{PROP} &\approx \mathbf{assert}(\mathit{grant} \wedge \mathit{request})\ t := \mathtt{time}
\end{aligned}$$

$$\sigma = \{\mathit{clk} \mapsto \bot, \mathit{grant} \mapsto \bot, \mathit{request} \mapsto \bot, t \mapsto \bot, \mathtt{time} \mapsto 0\}.$$

The formal specification of the arbiter is a parallel composition of process terms INIT, ARB and CLK **disrupt** ASSER:

- INIT – It assigns the initial values to variables *clk*, *grant* and *request* (i.e. the initialisation).

- ARB – It models the change of behaviour of variables *clk*, *grant* and *request* according to time.
- CLK – It models the behaviour of a clock (i.e. *clk*) which swaps the values between “0” and “1” every 5 time units.
- ASSER – It expresses the immediate assertion for the arbiter (as indicated above).
- CLK **disrupt** ASSER – It models the fact that the test of the immediate assertion is executed whenever there is a negative change in *clk*. When this happens, the current time is assigned to the variable t.

6.3.1 Formal Analysis of the Arbiter

We formally analyse (the immediate assertion of) the arbiter described in PAFSV by means of a complete mathematical proof via transition traces according to deduction rules defined in Section 3.7.

To increase the readability of the following proof, we often apply the commutativity property and associativity property of the parallel composition without explicitly referring to the deduction rules and such properties. We also do not specifically mention which assignment actions are used in the action transitions. We just mention them as some actions a and a'. Also, several unimportant brackets are introduced to group expressions, which may help the reader to follow the proof in a more intuitive way. In addition, we only consider the maximum duration for a possible time transition and the transitions of intermediate time points for such a time transition are not shown. For example, we only show $\langle \mathbf{delay}(5), \sigma\rangle \overset{5}{\longmapsto} \langle \mathbf{delay}(0), \sigma\rangle$ and not $\langle \mathbf{delay}(5), \sigma\rangle \overset{t_i}{\longmapsto}, \ldots, \overset{t_j}{\longmapsto} \langle \mathbf{delay}(0), \sigma\rangle$ for some $t_i, t_j \in \mathbb{R}_{>0}$ such that $t_i + \ldots + t_j = 5$.

1. We start with the above PAFSV process:

$$\langle \text{INIT} \parallel \text{ARB} \parallel \text{CLK } \mathbf{disrupt} \text{ ASSER}, \sigma\rangle.$$

2. Applying Rules (31), (29), (12) and (11) (for several times), we obtain:

$$\langle \text{INIT} \parallel \text{ARB} \parallel \text{CLK } \mathbf{disrupt} \text{ ASSER}, \sigma\rangle$$
$$\xrightarrow{a,\ldots,a} \langle \text{ARB} \parallel \text{CLK } \mathbf{disrupt} \text{ ASSER}, \sigma_2\rangle,$$

where $\sigma_2 = \{\mathit{clk} = \mathit{grant} = \mathit{request} \mapsto 0, t \mapsto \bot, \texttt{time} \mapsto 0\}$.

3. Due to Rules (33), (35), (12), (4), (31), (11), (3) and (42), the process has to perform a time transition of 4 time units first and then to execute the internal

action τ as follows:

$$\begin{aligned}&\langle \text{ARB} \parallel \text{CLK } \textbf{disrupt } \text{ASSER}, \sigma_2\rangle \overset{4}{\longmapsto}\overset{\tau}{\to}\\&\langle(\text{request} := 1;\ \text{G};\ \text{R}_0;\ \text{S}) \parallel (\textbf{delay}(1);\\&\mathit{clk} := \neg \mathit{clk});\ \text{CLK } \textbf{disrupt } \text{ASSER}, \sigma_3\rangle,\end{aligned}$$

where $\sigma_3 = \{\mathit{clk} = \mathit{grant} = \mathit{request} \mapsto 0, t \mapsto \bot, \texttt{time} \mapsto 4\}$.

4. Followed by Rules (31), (11) and (2), we have:

$$\begin{aligned}&\langle(\text{request} := 1;\ \text{G};\ \text{R}_0;\ \text{S}) \parallel (\textbf{delay}(1);\\&\mathit{clk} := \neg \mathit{clk});\ \text{CLK } \textbf{disrupt } \text{ASSER}, \sigma_3\rangle \overset{a}{\to}\\&\langle(\text{G};\ \text{R}_0;\ \text{S}) \parallel (\textbf{delay}(1);\ \mathit{clk} := \neg \mathit{clk});\ \text{CLK}\\&\textbf{disrupt } \text{ASSER}, \sigma_4\rangle,\end{aligned}$$

where $\sigma_4 = \{\mathit{request} \mapsto 1, \mathit{clk} = \mathit{grant} \mapsto 0, t \mapsto \bot, \texttt{time} \mapsto 4\}$.

5. Using Rules (33) and (12) together with (4), (3), (32), (11) and (2), we get:

$$\begin{aligned}&\langle(\text{G};\ \text{R}_0;\ \text{S}) \parallel (\textbf{delay}(1);\ \mathit{clk} := \neg \mathit{clk});\\&\text{CLK } \textbf{disrupt}\text{ASSER}, \sigma_4\rangle \overset{1}{\longmapsto}\overset{\tau}{\to}\ \overset{a}{\to} \langle(\\&\textbf{delay}(3);\ \text{grant} := 1;\ \text{R}_0;\ \text{S}) \parallel \text{CLK}\\&\textbf{disrupt } \text{ASSER}, \sigma_5\rangle,\end{aligned}$$

where $\sigma_5 = \{\mathit{clk} = \mathit{request} \mapsto 1, \mathit{grant} \mapsto 0, t \mapsto \bot, \texttt{time} \mapsto 5\}$.

6. Similarly, applying Rules (33), (42), (31), (12), (4), (3) and (30), we obtain:

$$\begin{aligned}&\langle(\textbf{delay}(3);\ \text{grant} := 1;\ \text{R}_0;\ \text{S}) \parallel \text{CLK}\\&\textbf{disrupt}\text{ASSER}, \sigma_5\rangle \overset{3}{\longmapsto}\overset{\tau}{\to}\ \overset{a}{\to}\ \overset{2}{\longmapsto}\overset{\tau}{\to}\\&\langle(\textbf{delay}(2);\ \text{request} := 0;\ \text{S}) \parallel \mathit{clk} := \neg \mathit{clk};\\&\text{CLK } \textbf{disrupt}\text{ASSER}, \sigma_6\rangle,\end{aligned}$$

where $\sigma_6 = \{\mathit{clk} = \mathit{request} = \mathit{grant} \mapsto 1, t \mapsto \bot, \texttt{time} \mapsto 10\}$.

7. At this stage, we know that the process term:

$$\begin{aligned}&\textbf{delay}(2);\ \text{request} := 0;\ \text{S} \parallel \mathit{clk} := \neg \mathit{clk};\ \text{CLK}\\&\textbf{disrupt } \text{ASSER}\end{aligned}$$

can only perform an action transition, because the first sub-process term of the right-argument (i.e. $\mathit{clk} := \neg \mathit{clk}$) of the parallel cannot delay (see also Rules (33) and (12)).

(a) We know that we can have: $\langle \mathbf{assert}(\mathit{grant} \wedge \mathit{request})\ t := \mathtt{time}, \sigma_6\rangle \xrightarrow{\tau} \xrightarrow{a'} \langle \checkmark, \sigma_7'\rangle$ for some a' using Rules (36) and (2), where $\sigma_7' = \{\mathit{clk} = \mathit{request} = \mathit{grant} \mapsto 1, t \mapsto 10, \mathtt{time} \mapsto 10\}$.

(b) Since $\eta_{negedge}(\mathit{clk})$ holds, because it is caused by the process term $\mathit{clk} := \neg \mathit{clk}$; CLK (also the left-argument of the disrupt process term $\mathit{clk} := \neg\mathit{clk}$; CLK **disrupt** ASSER) performing an action a from σ_6. So, we can apply Rule (21) to obtain: $\langle @_{(\eta_{negedge}(\mathit{clk}))}\mathrm{PROP}, \sigma_6\rangle \xrightarrow{a} \langle \checkmark, \sigma_7'\rangle$.

(c) According to Rule (11), we further get: $\langle @_{(\eta_{negedge}(\mathit{clk}))}\mathrm{PROP};\ \mathbf{skip}, \sigma_6\rangle \xrightarrow{a} \langle \mathbf{skip}, \sigma_7'\rangle$.

(d) By Rule (35), we can have: $\langle \mathrm{ASSER}, \sigma_6\rangle \xrightarrow{a'} \langle \mathbf{skip};\ \mathrm{ASSER}, \sigma_7'\rangle$.

(e) Clearly, $a \neq a'$ and $a \neq \tau$, based on the above-mentioned transition traces and using Rules (41) and (12), it is possible to have: $\langle \mathit{clk} := \neg\mathit{clk};\ \mathrm{CLK}\ \mathbf{disrupt}\ \mathrm{ASSER}, \sigma_6\rangle \xrightarrow{a} \langle \mathrm{CLK}\ \mathbf{disrupt}\ \mathbf{skip};\ \mathrm{ASSER}, \sigma_7\rangle$, where $\sigma_7 = \{\mathit{clk} \mapsto 0, \mathit{request} = \mathit{grant} \mapsto 1, t \mapsto 10, \mathtt{time} \mapsto 10\}$.

(f) From Rules (32), (41), (12) and (2), it is not hard to see that: $\langle (\mathbf{delay}(2);\ \mathrm{request} := 0;\ \mathrm{S}) \parallel \mathit{clk} := \neg\mathit{clk};\ \mathrm{CLK}\ \mathbf{disrupt}\ \mathrm{ASSER}, \sigma_6\rangle \xrightarrow{a} \langle (\mathbf{delay}(2);\ \mathrm{request} := 0;\ \mathrm{S}) \parallel \mathrm{CLK}\ \mathbf{disrupt}\ \mathbf{skip};\ \mathrm{ASSER}, \sigma_7\rangle$.

(g) Following the same fashion, more transition traces can be performed according to the deduction rules.

8. In σ_7, the variable t is mapped to the value of the current time (when the property is checked). This also means that the property holds, i.e. the arbiter worked as expected at least for *one time*.

6.3.2 Analysis Result

In simple words, through a complete formal (mathematical) proof of a desirable property of the above arbiter described in PAFSV, we showed that such arbiter worked as expected. Meanwhile, the operational semantics of PAFSV, by means of deduction rules, has been illustrated to allow for direct execution of formal specifications of PAFSV. Such deduction rules are also shown to be useful and suitable for formal analysis of SystemVerilog designs via our process algebraic framework PAFSV.

7 COMPARISON WITH OTHER FORMAL APPROACHES

Over the years, different formal approaches have been studied and investigated for VHDL [19], Verilog [20, 21, 22] and SystemC [23, 24]. Most of these works could only be considered as theoretical frameworks, except a few trails [22, 25], because they are not executable. Research works in formal semantics of SystemVerilog based on *Abstract State Machines* (ASMs) [26] and rewrite rules already exist (see Annex E in [6]). Also, ASM specifications and rewrite rules are not directly executable. However, it is also generally believed that structured operational semantics (SOS) provides more clear intuitions; and ASM specifications and rewrite rules appear to be less suited to describe the dynamic behaviour of processes. Since processes are the basic units of execution within SystemVerilog that are used to simulate the behaviour of a system, a process algebraic framework in SOS style is a more immediate choice to give the formal semantics of SystemVerilog (these motivated us to develop PAFSV in a process algebraic way with SOS deduction rules).

Based on the similar motivations and needs, a timed process algebra $SystemC^{\mathbb{FL}}$ [25, 27, 33] was introduced for formal specification and analysis of SystemC designs recently. $SystemC^{\mathbb{FL}}$ initiated an attempt to extend the knowledge and experience of the field of process algebras to SystemC designs. However, we believe that the formal semantics of $SystemC^{\mathbb{FL}}$ is not intuitive for designers and engineers in the electronic design community. For instance, the use/definition of two valuations (e.g. previous accompanying valuation and current valuation) in the quintuple of a $SystemC^{\mathbb{FL}}$ process is highly unintuitive for the users (see [28] for details).

On the other hand, as presented in this paper, only one valuation is defined in a PAFSV process. For a given current valuation σ and a given future valuation σ' (after a transition), deduction rules in PAFSV have been defined in such a way that, starting from the current state/valuation (i.e. σ), a more specific transition (e.g. $\langle p, \sigma\rangle \xrightarrow{a_{specific}} \langle p', \sigma'\rangle$) is precisely defined to reach the future state/valuation (i.e. σ'). In other words, such a transition is highly restricted by the current state/valuation and future state/valuation. The advantage of this approach to define deduction rules is that the defined deduction rules become simpler and more intuitive. It also helps to reduce non-determinism regarding the behaviour of the PAFSV process. Nevertheless, a formal comparison between $SystemC^{\mathbb{FL}}$ and PAFSV is indispensable.

8 CONCLUSIONS

PAFSV has been presented for the formal specification and analysis of IEEE SystemVerilog designs in this paper. We reached our goals (as indicated in Section 3.1). We also believe that our process algebraic framework PAFSV can serve as a mathematical basis for improvement of the design strategies of SystemVerilog, and possibilities to analyse SystemVerilog designs, because PAFSV

1. comprises mathematical expressions for SystemVerilog;
2. allows for analysis of specifications in a compositional way;
3. allows for equational reasoning on specifications;
4. contributes significantly to the investigation of interoperabilites of SystemVerilog with SystemC and $SystemC^{\mathbb{FL}}$.

Furthermore, we have successfully applied PAFSV to model and analyse null convention logic circuits.

Further to our knowledge, PAFSV is the first formalisation framework for a significant subset of SystemVerilog using a standard operational semantics (SOS) and such a semantics allows for direct execution of formal specifications.

9 FUTURE WORK

Our future work will focus on the development of a formal translation between PSL and PAFSV to reach the goal of verification of concurrent assertions in PAFSV as indicated above. We also investigate the translations of PAFSV to SMV [34], Promela [30] and timed automata that are input languages of the verification tools SMV and Uppaal [31], respectively.

For practical applications, we will apply PAFSV to formally represent SystemVerilog designs (for formal analysis purposes) in the design flow of the project: "$\mathcal{MOQA}$ Processor: An Entirely New Type of Processor for Modular Quantitative Analysis", as reported in [32].

REFERENCES

[1] Baeten, J.—Weijland, W.: Process Algebra. Cambridge University Press, 1990.

[2] Baeten, J.—Middelburg, C.: Process Algebra with Timing. Springer-Verlag, 2002.

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

[4] van Beek, D. et al.: Syntax and Consistent Equation Semantics of Hybrid Chi. Journal of Logic and Algebraic Programming, Vol. 68, 2006, No. 2, pp. 129–210.

[5] Cuijpers, P.—Reniers, M.: Hybrid Process Algebra. Journal of Logic and Algebraic Programming, Vol. 62, 2005, No. 2, pp. 191–245.

[6] IEEE-1800, IEEE Standard for SystemVerilog – Unified Hardware Design, Specification, and Verification Language. Technical report, IEEE Std 1 800$^{\mathrm{TM}}$, IEEE Computer Society, 2009.

[7] Sutherland, S.—Davidmann, S.—Flake, P.: SystemVerilog for Design: A Guide to Using SystemVerilog for Hardware Design and Modeling. Springer, 2003.

[8] Datla, S.—Thornton, M.—Hendrix, L.—Henderson, D.: Quaternary Addition Circuits Based on SUSLOC Voltage-Mode Cells and Modeling with SystemVerilog. Proceedings International Symposium on Multiple-Valued Logic, 2009, pp. 147–156.

[9] Spear, C.: SystemVerilog for Verification. Springer, 2008.

[10] IEEE-1364, IEEE Standard for Verilog Hardware Description Language. Technical report, IEEE Std 1364-2005, IEEE Computer Society, 2009.

[11] SystemVerilog: SystemVerilog 3.1a: Accellera's Extensions to Verilog. Available on: `http://www.systemverilog.com/`, 2003.

[12] Man, K.—Boubekeur, M.—Schellekens, M.: Algebraic Approach to SystemVerilog. Proceedings IEEE Canadian Conference on Electrical and Computer Engineering, 2007.

[13] Man, K.: PAFSV: A Process Algebraic Framework for SystemVerilog. Proceedings IEEE International Multiconference on Computer Science and Information Technology, 2008, pp. 535–542.

[14] Aceto, L.—Fokkink, W.—Verhoef, C.: Structural Operational Semantics. Handbook of Process Algebra, 2001, pp. 197–292.

[15] van Beek, D. A. et al.: Syntax and Semantics of Timed Chi. Technical Report, Eindhoven University of Technology, 2005.

[16] Mousavi, M.: Structuring Structural Operational Semantics. Ph.D. Thesis. Eindhoven University of Technology, 2005.

[17] Aceto, L.—Fokkink, W.—Verhoef, C.: Structural Operational Semantics. Proc. BPS, 1999, pp. 197–292.

[18] Mousavi, M. R.—Reniers, M. A.—Groote, J. F.: Notions of Bisimulation and Congruence Formats for SOS with Data. Information and Computation, Vol. 200, 2005, No. 1, pp. 107–147.

[19] Breuer, P.—Kloos, C. D.: Formal Semantics for VHDL. Kluwer Academic Publishers, 1995.

[20] Andraus, Z. S.—Liffiton, M. H.—Sakallah, K. A.: Reveal: A Formal Verification Tool for Verilog Designs. Logic for Programming, Artificial Intelligence, and Reasoning, 2008, pp. 343–352.

[21] Raffelsieper, M. et al.: Formal Analysis of Non-Determinism in Verilog Cell Library Simulation Models. Formal Methods for Industrial Critical Systems, 2009, pp. 133–148.

[22] Bowen, J.: Animating the Semantics of Verilog Using Prolog. Technical report, International Institute for Software Technology, United Nations University UNU/IIST Report 1999, No. 176.

[23] Razavi, N. et al.: Sysfier: Actor-Based Formal Verification of SystemC. ACM Transactions on Embedded Computing Systems, Vol. 10, 2010, No. 2, pp. 1–35.

[24] Vardi, M. Y.: Formal Techniques for SystemC Verification. Proceedings ACM/IEEE Design Automation Conference, 2007, pp. 188–192.

[25] Man, K.: $SystemC^{\mathbb{FL}}$: Formalization of SystemC. Proceedings IEEE Mediterranean Electrotechnical Conference, 2004.

[26] Mueller, W.—Zambaldi, M.—Ecker, W.—Kruse, T.: The Formal Simulation Semantics of SystemVerilog. Proceedings IEEE Forum on Specification and Design Languages, 2004.

[27] Man, K.: Formal Communication Semantics of $SystemC^{\mathbb{FL}}$. Proceedings IEEE Euromicro Conference on Digital System Design, 2005.

[28] Man, K. et al.: $SystemC^{\mathbb{FL}}_{\mathrm{tlm}}$: Motivation and Development. Proceedings IAENG International MultiConference of Engineers and Computer Scientists, 2008.

[29] IEEE Standard for Property Specification Language. Technical report, IEEE Std 1850-2010, IEEE Computer Society, 2010.

[30] Holzmann, G.: The SPIN Model Checker – Primer and Reference Manual. Technical report, Addison-Wesley, 2004.

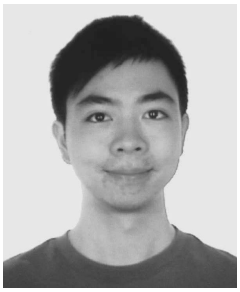
[31] Larsen, K.—Pettersson, P.—Yi, W.: UPPAAL in a Nutshell. Journal of Software Tools for Technology Transfer, Vol. 1, 1997, No. 1-2, pp. 134–152.

[32] Schellekens, M. et al.: Towards Fast and Accurate Static Average-Case Performance Analysis of Embedded Systems: The MOQA Approach. Proceedings IEEE East-West Design and Test International Symposium, 2007.

[33] *SystemC*$^{\mathbb{FL}}$. Available on: `http://digilander.libero.it/systemcfl/`, 2009.

[34] SMV: The SMV Model Checker and User Manual. Available on: `http://www-2.cs.cmu.edu/modelcheck/`, 2009.

Ka Lok Man is currently Associate Professor in the Department of Computer Science and Software Engineering at Xi'an Jiaotong-Liverpool University in Suzhou, China. Furthermore, he has a good publication record and to date has more than 300 published academic articles.



Chi-Un Lei is Honorary Assistant Professor in the Department of Electrical and Electronics Engineering, University of Hong Kong.

Hemangee K. Kapoor is Associate Professor with the Department of Computer Science and Engineering, Indian Institute of Technology Guwahati, Assam, India.

Tomas Krilavičius is Professor at Vytautas Magnus University, Kaunas, Lithuania. His main research interests are data mining, visualization and language technologies. He works as the Head of IT Department in Baltic Institute of Advanced Technologies.

Jieming Ma is Lecturer with School of Electronic and Information Engineering, Suzhou University of Science and Technology, Suzhou, China. His current research interests lie in the field of modeling and control of solar power systems.

Nan Zhang is currently working in the CITIC Securities, Shenzhen, China, as a quantitative trading strategy designer and developer.