Proceedings of the International Multiconference on
Computer Science and Information Technology pp. 535–542

ISBN 978-83-60810-14-9
ISSN 1896-7094

PAFSV: A Process Algebraic Framework for SystemVerilog

K. L. Man
Centre for Efficiency-Oriented Languages (CEOL)
Department of Computer Science
University College Cork (UCC), Ireland
Email: pafsv.team@gmail.com

***Abstract*—We develop a process algebraic framework, called process algebraic framework for IEEE 1800$^{\mathrm{TM}}$ SystemVerilog (PAFSV), for formal specification and analysis of IEEE 1800$^{\mathrm{TM}}$ 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$^{\mathrm{TM}}$ SystemVerilog. To show that PAFSV is useful for the formal specification and analysis of IEEE 1800$^{\mathrm{TM}}$ SystemVerilog designs, we illustrate the use of PAFSV with some examples: a MUX, a synchronous reset D flip-flop and an arbiter.**

I. 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 formally defined in *Computer Science* increase understanding of systems and clarity of specifications; and help solving problems and remove errors. Over the years, several flavours of formal languages have been gaining 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, process algebras describe the behaviour of processes and provide operations that allow to compose systems to obtain more complex systems. Moreover, the analysis and verification of systems described using 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 lies 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] and hybrid systems [3], [4]) 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 for analysis of many different systems.

On the other hand, the need for a formal and well-defined semantics of a *Hardware Description Languages* (HDL) is widely accepted and desirable for architects, engineers and researchers in the electronic design community. IEEE 1800$^{\mathrm{TM}}$ *SystemVerilog* [5] (in what follows, we abbreviate the IEEE 1800$^{\mathrm{TM}}$ SystemVerilog as SystemVerilog) is the industry's first unified hardware description and verification language (HDVL) standard; and SystemVerilog is a major extension of the established IEEE 1364$^{\mathrm{TM}}$ *Verilog* language [6] (see also [7]).

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 a process algebraic framework called **PAFSV** that is suitable for modelling and analysis of systems described in SystemVerilog (in a formal way). **PAFSV** covers the main features of SystemVerilog (i.e. a significant subset 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 is that, in contrast to other attempts to formalise semantics of SystemVerilog and HDLs, specifications described in **PAFSV** can be directly executable.

The formal semantics of $\mathbf{PAFSV}$ is defined by means of deduction rules in a standard structured operational semantics (SOS) [9] style that associate a *Time Transition System* (TTS) with a $\mathbf{PAFSV}$ process. A set of properties of $\mathbf{PAFSV}$ is presented for a notion of bisimilarity. Overview of process algebras, Verilog and SystemVerilog is not given in this paper. Some familiarity with them is required. The desirable background can, for example, be found in [1], [5], [6].

Over the years, different formal approaches have been studied and investigated for VHDL [10], Verilog [11], [12] and SystemC [14], [15]. Most of these works could only be considered as theoretical frameworks, except a few trails ([13], [17]), because they are not executable. Research work in the formal semantics of SystemVerilog based on *Abstract State Machines* (ASMs) [16] and rewrite rules already exist [5]. Also, ASM specifications and rewrite rules are not directly executable. It is also generally believed that a 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 a SOS style is a more immediate choice to give the formal semantics of SystemVerilog (these motivated us to develop $\mathbf{PAFSV}$ in a process algebraic way with SOS deduction rules). Based on the similar motivations and needs, three years ago, $SystemC^{\mathbb{FL}}$ [17], [18], [19] (a timed process algebra) was introduced for formal specification and analysis of SystemC designs. $SystemC^{\mathbb{FL}}$ initiated an attempt to extend the knowledge and experience of the field of process algebras to SystemC designs. Clearly, SystemVerilog and SystemC are similar and our research work in this direction was highly inspired by the theoretical aspects of $SystemC^{\mathbb{FL}}$. Hence, a formal comparison between them is indispensable (as future work). Furthermore, an introduction (paper) of $\mathbf{PAFSV}$ can be found in [20]. Such a paper only informally presented the syntax and semantics of $\mathbf{PAFSV}$. Also, no deduction rules were given, validation of the semantics was not discussed and no analysis example of $\mathbf{PAFSV}$ specifications was provided.

This paper is organised as follows. Section II shows the goals, the data types, formal syntax and formal semantics of our process algebraic framework $\mathbf{PAFSV}$. To illustrate the use, effectiveness and applicability of the deduction rules, in Section III, some simple specifications of $\mathbf{PAFSV}$ are provided. In Section IV, the correctness of the formal semantics of $\mathbf{PAFSV}$ defined in Subsection II-E is discussed; and a notion of equivalence is defined, which is shown to be a congruence for all $\mathbf{PAFSV}$ operators. Also, a set of useful properties of closed $\mathbf{PAFSV}$ process terms is given in the same section. Samples (modelling some SystemVerilog designs) of the application of $\mathbf{PAFSV}$ are shown in Section V. A formal analysis (by means of a complete mathematical proof) of a SystemVerilog design via $\mathbf{PAFSV}$ is presented in Section VI. Finally, concluding remarks are made in Section VII and the direction of future work is pointed out in the same section.

II. $\mathbf{PAFSV}$

Obviously, it is not possible to cover all the aspects of SystemVerilog and define a process algebraic framework for it in one paper. Hence, in this section, we outline the goals to achieve in this paper.

We propose a process algebraic framework namely $\mathbf{PAFSV}$ that 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 $\mathbf{PAFSV}$ is as follows:

- to give a formal semantics to a significant subset of SystemVerilog using the operational approach of [9];
- 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.

A. Data types

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 $(x_0, \ldots, x_n, \mathtt{time})$. 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 $(v_0, \ldots, v_m, \perp)$ that contains at least all $\mathrm{Integers}$, all Reals, all $\mathrm{Shortreals}$, all $2-\mathrm{statevalues}$ and all $4-\mathrm{statevalues}$ as defined in SystemVerilog (see [5] for details); all $\mathrm{Booleans}$ and $\perp$, where $\perp$ 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 $\{x_0 \mapsto v_0, \ldots, x_n, \mapsto v_n, \mathtt{time} \mapsto t\}$, 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 $\mathbf{PAFSV}$. However, the behaviour of elements in an array in SystemVerilog can be modelled in $\mathbf{PAFSV}$ by introducing fresh variables. As an example, for an array A[0:10] in SystemVerilog, we can introduce fresh variables $A_0, \ldots, A_{10}$ in $\mathbf{PAFSV}$ to associate correspondingly A[0] with A_0, A[1] with A_1 and so on.

B. Formal syntax

To avoid confusion with the 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 Subsection II-E) 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 $p \in P$:

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

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, $\eta_1, \ldots, \eta_n$ represent boolean functions with corresponding parameters $l_1, \ldots, l_n \in \mathrm{Var}$.

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: {**if**(_)**_else_**, **wait**(_)_, **while**(_)_, **assert**(_)_}, _; _, **_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 [3] (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. The **_disrupt_** is inspired by the analogy of the disrupt operator in HyPA [4]. This can be used to model event controls in **PAFSV** in a very efficient way. A concise explanation of the formal syntax of **PAFSV** is given below. Subsection II-E gives a more detailed account of its meaning.

C. 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 it 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 $p \in P$ (i.e. **any** p), delay 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) Similarly, 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 do 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.
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 of 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. The need and use of this operator will be illustrated in Section VI.

E. Formal semantics

In this subsection, we give a formal semantics to the syntax defined for $\mathbf{PAFSV}$ in the previous subsection, by constructing a kind of time transition system (TTS), for each process term and each possible valuation of variables.

Definition 1 *We use the convention $\langle p, \sigma\rangle$ to write a $\mathbf{PAFSV}$ process, where $p \in P$ and $\sigma \in \Sigma$.*

Definition 2 *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 $\mathbf{PAFSV}$ that can be freely instantiated.*

Definition 3 *We give a formal semantics for $\mathbf{PAFSV}$ processes in terms of a time transition system (TTS), and define the following transition relations on processes of $\mathbf{PAFSV}$:*

- $\_ \xrightarrow{\_} \langle \checkmark, \_\rangle \subseteq (P \times \Sigma) \times A_\tau \times \Sigma$, *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;*
- $\_ \overset{\_}{\longmapsto} \_ \subseteq (P \times \Sigma) \times \mathbb{R}_{>0} \times (P \times \Sigma)$, *denotes time transition (so-called delay).*

For $p, p' \in P$; $\sigma, \sigma' \in \Sigma$, $a \in A_\tau$ and $d \in \mathbb{R}_{>0}$, 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 $\langle p, \sigma\rangle \overset{d}{\longmapsto} \langle p', \sigma'\rangle$ is that the process $\langle p, \sigma\rangle$ may idle during a d time units and then behaves like $\langle p', \sigma'\rangle$.

F. Deduction rules

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

Apart from the syntax restriction as already shown in Subsection II-B (e.g. $x, w \in \mathrm{Var}$), for all deduction rules, we further require that $p, q, p', q' \in P$; $\sigma, \sigma', \sigma'' \in \Sigma$; $a, b \in A_\tau$, $d \in \mathbb{R}_{>0}$, $\mathrm{dom}(\sigma) = \mathrm{dom}(\sigma') = \mathrm{dom}(\sigma'')$; 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 $\mathrm{Var}^- = \{x^- \mid x \in \mathrm{Var}\}$ and $\mathrm{Var}^+ = \{x^+ \mid x \in \mathrm{Var}\}$, 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.

In order to increase the readability of the $\mathbf{PAFSV}$ deduction rules, the notation $\overset{z}{\twoheadrightarrow}$ is used as a short-hand for $\xrightarrow{a}$ and $\overset{d}{\longmapsto}$.

Deduction rules: 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.

1) Procedural assignment:

$$\frac{}{\langle x := e, \sigma\rangle \xrightarrow{aa(x, \bar{\sigma}(e))} \langle \checkmark, \sigma[\bar{\sigma}(e)/x]\rangle}\ 1$$

By means of a procedural assignment (see Rule 1), 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 $\bar{\sigma}(e)$.

2) Sequential composition:

$$\frac{\langle p, \sigma\rangle \xrightarrow{a} \langle \checkmark, \sigma'\rangle}{\langle p;\ q, \sigma\rangle \xrightarrow{a} \langle q, \sigma'\rangle}\ 2 \qquad \frac{\langle p, \sigma\rangle \overset{z}{\twoheadrightarrow} \langle p', \sigma'\rangle}{\langle p;\ q, \sigma\rangle \overset{z}{\twoheadrightarrow} \langle p';\ q, \sigma'\rangle}\ 3$$

The process term q is executed after (successful) termination of the process term p as defined by Rules 2 and 3.

3) Parallel composition:

$$\frac{\langle p, \sigma\rangle \xrightarrow{a} \langle \checkmark, \sigma'\rangle}{\langle p \parallel q, \sigma\rangle \xrightarrow{a} \langle q, \sigma'\rangle}\ 4 \qquad \frac{\langle q, \sigma\rangle \xrightarrow{a} \langle \checkmark, \sigma'\rangle}{\langle p \parallel q, \sigma\rangle \xrightarrow{a} \langle p, \sigma'\rangle}\ 5$$

$$\frac{\langle p, \sigma\rangle \xrightarrow{a} \langle p', \sigma'\rangle}{\langle p \parallel q, \sigma\rangle \xrightarrow{a} \langle p' \parallel q, \sigma'\rangle}\ 6 \qquad \frac{\langle q, \sigma\rangle \xrightarrow{a} \langle q', \sigma'\rangle}{\langle p \parallel q, \sigma\rangle \xrightarrow{a} \langle p \parallel q', \sigma'\rangle}\ 7$$

$$\frac{\langle p, \sigma\rangle \overset{d}{\longmapsto} \langle p', \sigma'\rangle,\ \langle q, \sigma\rangle \overset{d}{\longmapsto} \langle q', \sigma'\rangle}{\langle p \parallel q, \sigma\rangle \overset{d}{\longmapsto} \langle p' \parallel q', \sigma'\rangle}\ 8$$

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 from Rule 4 to Rule 7). 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 8.

III. 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 1, 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 3 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 1 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 2. However, this is not possible due to Rules 4 and 5.

IV. 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 (see also [3], [21]).

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.

A. Well-definedness of the semantics

The deduction rules defined for **PAFSV** constitute a *Transition System Specification* (TSS) as described in [22]. The transitions that can be proven from a TSS define a time transition system (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 [22] for details).

As we know that no negative premise is used in our deduction rules for **PAFSV**. So, it is not hard to see that the term deduction system of **PAFSV** is well-defined. This means that the system defines a unique transition system for each closed process term of **PAFSV**.

B. Bisimilarity

Two closed **PAFSV** process terms are considered equivalent if they have the same behaviour (in the bisimulation sense) from the current state.

We also assume that the valuation (of the current state) contains at least the free occurrences of variables in the two closed **PAFSV** process terms being equivalent.

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

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 **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* **PAFSV** *operators.*

Proof: *All deduction rules of* **PAFSV** *are in the process-tyft format of [21]. It follows from [21] that stateless bisimilarity is a congruence.*

C. 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.

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).

[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.

Proposition 1 (Properties) *A set of properties is introduced for* $\mathbf{PAFSV}$ *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*** $(p \circledast q)$,

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 [3].*

Intuition behind the properties

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.
- The any operator distributes to the right argument of a sequential composition.
- Alternative composition and parallel composition are commutative and associative.
- Alternative composition distributes over sequential composition from the left, but not from the right.
- The any operator distributes over the alternative composition.

V. EXAMPLES OF $\mathbf{PAFSV}$ SPECIFICATIONS

This section is a sample of the application of $\mathbf{PAFSV}$. It is meant to give a first impression of how one can describe the behaviour of some SystemVerilog designs in $\mathbf{PAFSV}$ (in a complete mathematical sense). We describe the behaviour of a simple MUX and a simple synchronous reset D flip-flop.

A. MUX

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 2 depicts such a MUX.

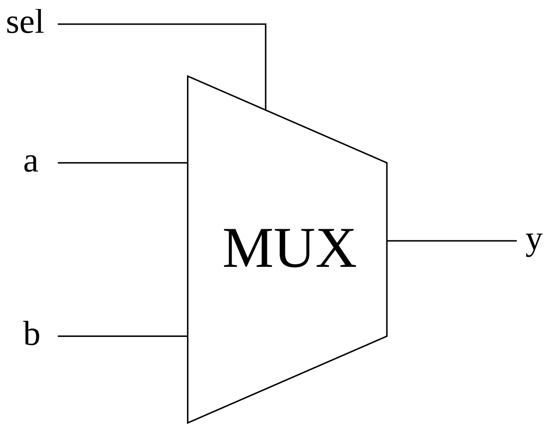

Fig. 1. A MUX.

```
module  simple_mux (
input  wire  a,
input  wire  b,
input  wire  sel,
output wire  y
);
assign y = (sel) ? a : b;
endmodule
```

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

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

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

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

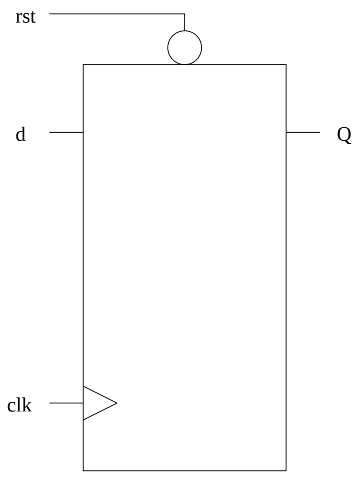

Fig. 2. A synchronous reset D flip-flop.

The formal **PAFSV** specification (as a process term) of the above synchronous reset D flip-flop (described as a module in SystemVerilog) is given as follows:

$$\begin{array}{ll} \mathrm{DFF} & \approx \textbf{repeat}(@_{(\eta_{negedge}(clk))}\mathrm{OUT}) \\ \mathrm{OUT} & \approx \textbf{if}(\neg rst)\ Q := 1'b0\ \textbf{else}\ Q := d \end{array}$$

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).

VI. Analysis of an PAFSV Specification

We have already shown in Section V that **PAFSV** specifications can be used to formally represent SystemVerilog designs. Therefore, in this section, we formally analyse a simple arbiter described in SystemVerilog via **PAFSV**.

A. 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 immediate assertion can be considered as a *liveness property* of the arbiter. If the assertion holds, this means that the arbiter works as expected. Below is a SystemVerilog design of the simple arbiter as described above:

```
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$, \sigma \rangle$, where

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

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

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. ***Remark:*** This also explains the need and the use of the "**disrupt** process term", because the execution of process term CLK must have a higher priority than the execution of process term ASSER (since the change of the clock causes the test to be run).

B. Formal analysis of the arbiter

The arbiter described in **PAFSV** was analysed by means of a complete mathematical proof via transition traces according to deduction rules of **PAFSV**. The liveness property (i.e. the immediate assertion holds at least for some times) of the arbiter was proved to hold. In this paper, due to the reason of spaces, the above-mentioned proof is omitted.

VII. Conclusions and Future Work

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 SystemVerilog designs. Nevertheless, we reached our goals (as indicated in Section II). 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 $SystemC^{\mathbb{FL}}$.

We have the idea that, like $SystemC^{\mathbb{FL}}$, **PAFSV** can serve as a *single-formalism-multi-solution*. This means that we can formally translate a **PAFSV** specification to the input languages (e.g. SMV [23], Promela [24] and timed automata [25]) of several verification tools (e.g. SMV [23], SPIN [24] and Uppaal [26]) and it can be verified in those verification tool environments.

Our future work will develop/investigate such translations. 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 [27].

VIII. Availability

The full set of **PAFSV** deduction rules and the complete mathematical proof of the correctness of the arbiter (see VI-B for details) are available by email at pafsv.team@gmail.com.

IX. Acknowledgement

The author wishes to thank Jos Baeten, Bert van Beek, Mohammad Mousavi, Koos Rooda, Ramon Schiffelers, Pieter Cuijpers, Michel Reniers, Kees Middelburg, Uzma Khadim and Muck van Weerdenburg for many stimulating and helpful discussions focusing on process algebras for distinct systems in the past few years.

Many thanks go to Michel Schellekens and Menouer Boubekeur for their contributions to the introduction paper of **PAFSV** (see [20]) and the industrial collaborators Solari—Hong Kong (http://www.solari-hk.com/), International Software and Productivity Engineering Institute—USA (http://www.intspei.com), Intelligent Support Ltd.—United Kingdom (http://www.isupport-ltd.co.uk) and Minteos—Italy (http://www.minteos.com) of the research work presented in this paper.

References

[1] J. C. M. Baeten, W. P. Weijland, *Process Algebra*, Number 18 in Cambridge Tracts in Theoretical Computer Science, Cambridge University Press, 1990.

[2] J. C. M. Baeten, C. A. Middelburg, *Process Algebra with Timing*, in EATCS Monographs Series, Springer-Verlag, 2002.

[3] D. A. van Beek, K. L. Man, M. A. Reniers, J. E. Rooda, R. R. H. Schiffelers, *Syntax and Consistent Equation Semantics of Hybrid Chi*, in Journal of Logic and Algebraic Programming, 68(1–2):129-210, 2006.

[4] P. J. L. Cuijpers, M. A. Reniers, *Hybrid Process Algebra*, in Journal of Logic and Algebraic Programming, 62(2):191–245, 2005.

[5] *IEEE Standard for SystemVerilog—Unified Hardware Design, Specification, and Verification Language*, IEEE Std 1800^{TM}-2005, IEEE Computer Society, 2005.

[6] *IEEE Standard for Verilog Hardware Description Language*, IEEE Std 1364-2005 (Revision of IEEE Std 1364-2001), IEEE Computer Society, 2006.

[7] *SystemVerilog 3.1a: Accellera's Extensions to Verilog*, Napa, CA, 2003. Available in PDF form at http://www.systemverilog.com/

[8] **PAFSV** *homepage*, http://digilander.libero.it/systemcfl/pafsv/

[9] G. D. Plotkin, *A Structural Approach to Operational Semantics*, in Report DAIMI FN-0.59, Computer Science Department, Aarhus University, 1981.

[10] P. T. Breuer, C. Delgado Kloos, *Formal Semantics for VHDL*, Kluwer Academic Publishers, 1995.

[11] G. Schneider, X. Qiwen, *Towards an Operational Semantics of Verilog*, UNU/IIST Report No. 147, International Institute for Software Technology, United Nations University, Macau, 1998.

[12] G. Schneider, X. Qiwen, *Towards a Formal Semantics of Verilog Using Duration Calculus*, IN A. Ravn, H. Rischel, editors, Formal Techniques for Real-Time and Fault Tolerant Systems (FTRTFT'98), LNCS, Springer-Verlag, 1998.

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

[14] W. Mueller, J. Ruf, D. Hofmann, J. Gerlach, T. Kropf and W.Rosenstiehl, *The Simulation Semantics of SystemC*, in Proceedings of DATE, 2001

[15] A. Salem, *Formal Semantics of Synchronous SystemC*, in Proceedings of DATE, 2003.

[16] W. Mueller, M. Zambaldi, W. Ecker, T. Kruse, *The Formal Simulation Semantics of SystemVerilog*, in Proceedings of the FDL, France, 2004.

[17] K. L. Man, *$SystemC^{\mathbb{FL}}$: Formalization of SystemC*, in IEEE Proceedings of the 12th Mediterranean Electrotechnical Conference—MELECON 2004, Dubrovnik, Croatia, May, 2004.

[18] K. L. Man, *Formal Communication Semantics of $SystemC^{\mathbb{FL}}$*, in IEEE Proceedings of the 8th Euromicro Conference on Digital System Design—DSD05, Porto, Portugal, September, 2005.

[19] $SystemC^{\mathbb{FL}}$ *homepage*, http://digilander.libero.it/systemcfl/

[20] K. L. Man, M. Boubekeur, M. P. Schellekens, *Process Algebraic Approach to SystemVerilog*, in IEEE Proceedings of the 20th IEEE Canadian Conference on Electrical and Computer Engineering, Vancouver, British Columbia, Canada, April, 2007.

[21] M. R. Mousavi, *Structuring Structural Operational Semantics*, Ph. D. Thesis, Department of Computer Science, Eindhoven University of Technology, September 2005.

[22] L. Aceto, W. Fokkink, C. Verhoef, *Structural Operational Semantics*, in Bergstra et al. BPS01, pp. 197–292, 1999.

[23] *The SMV model checker and user manual*, are available at http://www-2.cs.cmu.edu/~modelcheck/

[24] G. J. Holzmann, *The SPIN Model Checker*, Primer and Reference Manual, Addison-Wesley, 2004.

[25] R. Alur, D. L. Dill, *A Theory of Timed Automata*, Theoretical Computer Science, Vol. 126, No. 2, pp. 183-236, April, 1994.

[26] K. G. Larsen, P. Pettersson, W. Yi, *UPPAAL in a Nutshell*, Journal of Software Tools for Technology Transfer (STTT), Vol 1, No. 1-2, pp. 134–152, 1997.

[27] M. P. Schellekens, R. Agarwal, A. Fedeli, Y. F. Lam, K. L. Man, M. Boubekeur, E. Popovici, *Towards Fast and Accurate Static Average-Case Performance Analysis of Embedded Systems: The $\mathcal{MOQA}$ Approach*, in IEEE Proceedings of the East-West Design and Test International Symposium, September, 2007.