Extensions of $\text{SystemC}^{\text{FL}}$ for Mixed-Signal Systems and Formal Verification

K.L. Man
Formal Methods Group, Department of Mathematics and Computer Science
Eindhoven University of Technology, P.O.Box 513, 5600 MB Eindhoven, The Netherlands
Phone: +31 (0)40 2472993, Fax: +31 (0)40 2475361
E-mail: kman@win.tue.nl

Abstract—The formal language $\text{SystemC}^{\text{FL}}$ is the formalization of SystemC. The language semantics of $\text{SystemC}^{\text{FL}}$ was formally defined in a standard structured operational semantics (SOS) style. In this paper, we first provide an overview of the current status of the formal language $\text{SystemC}^{\text{FL}}$ and show some practical applications of $\text{SystemC}^{\text{FL}}$. Then, we give an outline for the latest developments of $\text{SystemC}^{\text{FL}}$. These developments include extensions of $\text{SystemC}^{\text{FL}}$ for modeling mixed-signal systems and formal verification.

Keywords—SystemC, $\text{SystemC}^{\text{FL}}$, Hardware/software, Process Algebra, Structured Operational Semantics (SOS), System Level Design Modeling, Mixed-signal Systems, Formal Verification

I. INTRODUCTION

SystemC [1] is a modeling and simulation language (without formal semantics defined) based on C++ for hardware and system level design modeling. Recently, SystemC has received an extreme increase in industrial acceptence for system specification and simulation.

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.

Due to the above-mentioned motivations, we developed the formal language $\text{SystemC}^{\text{FL}}$ [2] (a portable subset) of SystemC. Since processes are the basic units of execution within SystemC that are used to simulate the behavior of a device or a system, Process Algebra [8] was chosen as the mathematical framework for $\text{SystemC}^{\text{FL}}$. Process algebra was used, because it provides an elegant notation for transition, and allows for axiomatic reasoning. The main goal of $\text{SystemC}^{\text{FL}}$ is to provide the formal reasoning of SystemC designs and the formal analysis about the behavior of SystemC processes.

Based on the informal semantics presented in [1], the language semantics of $\text{SystemC}^{\text{FL}}$ was formally defined in a standard structured operational semantics (SOS) style [9]. Furthermore, a strong state based bisimulation on $\text{SystemC}^{\text{FL}}$ processes was defined in [2] and shown to be a congruence. Moreover, a set of useful axioms was also introduced in [2]. Recently in [6], we extended the formal language $\text{SystemC}^{\text{FL}}$ to deal with concurrency and interaction. The newly developed communication semantics of $\text{SystemC}^{\text{FL}}$ was also formally defined in SOS style. The proposed semantics can incorporate both point-to-point communication and multi-party communication.

A. Related Works

The simulation semantics (including watching statement, signal assignment, and wait statement) of SystemC in the form of distributed Abstract State Machine (ASM) specifications and the Denotational Semantics for a synchronous subset of SystemC were studied by [11] and [12] respectively.

It is general believed that the SOS style semantics is more intuitive [10], and the methods of ASM specifications and denotational semantics appear to be difficult to apply to describe the dynamic behavior of processes. Therefore, the language semantics of $\text{SystemC}^{\text{FL}}$ was formally defined in a standard structured operational semantics (SOS) style.

It should be noted that the fundamental mechanisms used to model processes in $\text{SystemC}^{\text{FL}}$ were inspired by similar constructs in the formal specification Chi language [7] for hybrid systems and the Algebra of Communicating Processes (ACP) [8].

B. Organization

The remainder of this paper is organized as follows. The next section introduces the formal language of $\text{SystemC}^{\text{FL}}$. Section III and IV present some practical applications of $\text{SystemC}^{\text{FL}}$. Various possible extensions for $\text{SystemC}^{\text{FL}}$ are shown in Section V. Section VI contains our conclusions.
II. SystemCFL Language

In this section, we introduce the formal language SystemCFL. For the syntax and the formal semantics of SystemCFL, we also refer to [2] and [6].

A. SystemCFL Data Types

In order to define the semantics of SystemCFL processes, we need to make some assumptions about the data types. Let Var denote the set of all variables \( x_0, \ldots, x_n \), and Value denote the set all possible values \( v_0, \ldots, v_n \) that contains at least \( \mathbb{B} \) (booleans) and \( \mathbb{R} \) (reals). A value is a partial function from variables to values (e.g. \( x_0 \mapsto v_0 \)). The set of all valuations is denoted by \( \Sigma \). The set \( Ch \) of all channels and the set \( S \) of all sensitivity lists with clocks may be used in SystemCFL processes that are assumed. Notice that the above proposed data types are the fundamental ones. Several extensions of data types (e.g. “sc_bit” and “sc_logic”) were already introduced in [3].

B. Syntax of the SystemCFL Language

A process term \( P \) in SystemCFL is built from atomic process terms \( AP \). SystemCFL consists of various operators that operate on process terms. The formal language SystemCFL is defined according to the following grammar:

\[
AP ::= \delta \mid \text{skip} \mid x := e \mid \Delta e_n \mid \triangleright \triangleright
\]

\[
P ::= AP \mid P \| \bullet P \mid P \triangleright P \mid P \bullet P \mid P \|_\ell P \mid P \sim P \mid P \| H P \mid \tau_I(P)
\]

The operators are listed in descending order of their binding strength as follows: \( \{ \|_\ell, \bullet, \triangleright, \tau_I \}, \{ \|, \Theta, \|_H \}, \{ \|, \tau \} \). The operators inside the braces have equal binding strength. In addition, operators of equal binding strength associate to the left, and parentheses may be used to group expressions. Below is an introduction of the formal language SystemCFL.

A constant called deadlock \( \delta \) is introduced, which represents no behavior. The \text{skip} process term performs the internal action \( \tau \). The assignment process term \( x := e \), which assigns the value of expression \( e \) to \( x \) (modeling a SystemC “assignment” statement). The delay process term \( \Delta e_n \), which is able to delay the value of numerical expression \( e_n \). The unbounded delay process term \( \triangleright \triangleright \) (modeling a SystemC “wait” statement) may delay for arbitrary long duration of time or perform the internal action \( \tau \).

Complex process terms are constructed using several operators. The conditional composition operator \( p \|_\ell b \| P \) operates as a SystemC “then-else” statement, where \( b \) denotes a boolean expression and \( p, q \in P \). The watching operator \( b \triangleright P \) is used to model a SystemC “watching” statement. The timeout process term \( p \triangleq q \) (modeling a SystemC “time out” construct) behaves as \( p \) if \( p \) performs a time transition before a duration of time \( d \in \mathbb{R} : d > 0 \). Otherwise, it behaves as \( q \). The sequential composition \( p \bullet q \) models the process term that behaves as \( p \), and upon termination of \( p \), continues to behave as process term \( q \). The alternative composition \( p \| q \) models a non-deterministic choice between process terms \( p \) and \( q \). The watchdog process term \( p \| H q \) behaves as \( p \) during a duration of time less than \( d \). At time \( d, q \) takes over the execution from \( p \) in \( p \| q \). If \( p \) performs an internal cancel \( \chi \) action, then the delay is canceled, and the subsequent behavior is that of \( p \) after \( \chi \) is executed. A repetition \( *p \) (modeling a SystemC “loop” construct) executes \( p \) zero or more times. The parallel composition \( \| \) and the communication composition \( \sim \) are used to express parallelism. The encapsulation of actions is allowed using \( \partial_I(p) \), where \( H \) represents the set of all actions to be blocked in \( p \). The abstraction \( \tau_I(p) \) behaves as the process term \( p \), except that all actions names in \( I \) are renamed to the internal action \( \tau \). Notice that we always assume that the execution of action transitions has priority over time transitions (i.e. the maximal progress operator is not defined).

C. Semantics of the SystemCFL Language

Definition 1: A SystemCFL process is a quintuple \( \langle P, \Sigma, \Theta, S, Ch \rangle \). We use the convention \( \langle p, \sigma', \sigma, s, m \rangle \) to write a SystemCFL process, where \( p \) is a process term; \( \sigma, \sigma' \) are valuations; \( s \) is a set of sensitivity lists with clocks; \( m \) is a channel.

Definition 2: The set of all actions \( A_\tau \) is defined as follows: \( A_\tau = \{ aa(x,v), s(m), r(m), com(m), \chi, \tau \} \), where \( aa(x,v) \) is the assignment action (i.e. the value of \( v \) is assigned to \( x \)), \( s(m) \) is the parameterized send action, \( r(m) \) is the parameterized receive action, \( com(m) \) is the parameterized communication action between \( s(m) \) and \( r(m) \), \( \chi \) is the internal cancel action and \( \tau \) is the internal action.

Definition 3: A formal semantics for SystemCFL processes is given in terms of a Labelled Transition System (LTS). We define the following transition relations on SystemCFL processes:

- an action transition \( (p, \sigma', \sigma, s, m) \overset{\alpha}{\rightarrow} (p', \sigma, \sigma'', s, m) \) is that the process \( (p, \sigma', \sigma, s, m) \) executes the action \( \alpha \in A_\tau \), starting with the current valuation \( \sigma \) (at the moment of the transition taking place) and by this execution \( p \) evolves into \( p' \), where \( \sigma' \) represents the previous accompanying valuation of the process, and \( \sigma'' \) represents the accompanying
valuation of the process after the action \( a \) is executed.

- a termination transition \( \langle p, \sigma', \sigma, s, m \rangle \xrightarrow{\text{\checkmark}} \langle \sigma', \sigma''', s, m \rangle \) is that the process executes the action \( a \) followed by termination, where \( \checkmark \) is used to indicate a successful termination, and \( \checkmark \) is not a process term.

- a time transition (so-called delay) \( \langle p, \sigma', \sigma, s, m \rangle \xrightarrow{d} \langle p', \sigma', \sigma'', s, m \rangle \) is that the process \( \langle p, \sigma', \sigma, s, m \rangle \) may idle for a duration of time \( d \) and then behaves like \( \langle p', \sigma', \sigma'', s, m \rangle \).

D. Deduction Rules

The above transition relations are defined through deduction rules (SOS style). These rules (of the form \( \frac{\text{premises}}{\text{conclusions}} \)) 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 conclusions hold as well. Giving the deduction rules for all atomic process terms and other operators of \( \text{SystemC}^{\text{FL}} \) is far beyond the scope of this paper, we refer those rules to [2] and [6].

III. Modeling with \( \text{SystemC}^{\text{FL}} \)

The formal language \( \text{SystemC}^{\text{FL}} \) can be reasonably efficiently used to model software, hardware and concurrency [3]. In this section, we apply \( \text{SystemC}^{\text{FL}} \) to model two nontrivial case studies. All two case studies are taken from [1], rather than devised by us.

Synchronous D Flip Flop

D flip flops are one of the most basic building blocks of RTL designs. Below is a SystemC implementation that implements a synchronous D flip flop.

```c
// dff.h
#include "systemc.h"
SC_MODULE(dff) {
    sc_in<bool> din;
    sc_in<bool> clock;
    sc_out<int> dout;
    void doit() {
        dout = din;
    }
    SC_CTOR(dff) {
        SC_METHOD(doit);
        sensitive_pos << clock;
    }
}
```

A formal \( \text{SystemC}^{\text{FL}} \) specification of the above synchronous D flip flop is given as follows:

\[
\langle \text{Cond}_{\text{clock}}(\sigma', \sigma, s) \cup (d_{\text{out}} := d_{\text{in}}), \sigma', \sigma, s, m \rangle,
\] for some \( \sigma', \sigma, s, m \).

\( \text{Cond}_{\text{clock}} \) is a function that returns true if a positive edge occurs on port clock. The formal \( \text{SystemC}^{\text{FL}} \) specification of the above synchronous D flip flop has a clock input (\( \text{clock} \)), a data input (\( \text{din} \)), and a data output (\( \text{dout} \)). When a positive edge occurs on the clock input (which means the function \( \text{Cond}_{\text{clock}} \) returns true), the input port data (\( d_{\text{in}} \)) is assigned to the output port \( d_{\text{out}} \). Notice that \( \text{clock} \), \( d_{\text{in}} \), \( d_{\text{out}} \) \in \text{dom}(\sigma'), \text{dom}(\sigma), \) and only \( \text{clock} \in s \).

Remote Procedure Call (RPC) Protocol

With RPC, a process in a module can call a function in another module, which is called the slave process. This is very similar to RPC semantics in Unix. The two processes must be connected through specialized ports to a specialized communication link. Below shows a \( \text{SystemC}^{\text{FL}} \) implementation of RPC communication.

```c
SC_MODULE(producer) {
    sc_outmaster<int> out1;
    sc_in<bool> start;
    void generate_data () {
        for (int i = 0; i < 10; i++) {
            out1 = i;
        }
    }
    SC_CTOR(producer) {
        SC_METHOD(generate_data);
        sensitive << start;
    }
}
```

```c
SC_MODULE(consumer) {
    sc_inslave<int> in1;
    int sum;
    void accumulate() {
        sum += in1;
        cout << "Sum = " << sum << endl;
    }
    SC_CTOR(consumer) {
        SC_SLAVE(accumulate, in1);
        sum = 0;
    }
}
```
A formal SystemC\textsuperscript{TL} specification of the above RPC communication is given as follows:

\[
\{(\tau_{c_n(d_a)})\{d_{s_m(d_a)}\}r_{m(d_a)}\{(\text{Cond}(\sigma^t,\sigma,\{\text{start}\})
\oplus \text{producer} \parallel \text{consumer})\}, \sigma^t, \sigma, s, m) \text{ for some } \\
\forall i < 10 \text{ and } \text{sum} := i; \text{start} := i + 1 \text{ and the process consumer} \equiv \text{sum} \equiv \text{in}_1 \text{ respectively. Notice that } \{i \mapsto 0, \text{sum} \mapsto 0\} \in \sigma, \text{ and start } \in \sigma.
\]

We model the formal SystemC\textsuperscript{TL} specification of the above RPC communication slightly different from the SystemC implementation, because we would like to show how to model communication between processes through channel (rather than multi-point link) using SystemC\textsuperscript{TL}. In the above formal SystemC\textsuperscript{TL} specification, the process producer (sensitive to start) produces a set of numbers that each number invokes the process consumer, which accumulates the numbers. These two processes execute concurrently (modeled by the $|$ operator) and communicate over channel $m$. We write $s_{m}(d_a), r_{m}(d_a)$ and $c_{m}(d_a)$ for the action of sending datum $d_a$ through channel $m$, the action of receiving datum $d_a$ through channel $m$, and the action of communicating datum $d_a$ through channel $m$. Intuitively, the process producer sends the value of out$_1$ through channel $m$, and the process consumer receives the value of in$_1$ through channel $m$. The action $c_{m}(d_a)$ is the action that is left when $s_{m}(d_a)$ and $r_{m}(d_a)$ are performed synchronously (i.e. the process producer and the process consumer communicate over channel $m$ and out$_1 = \text{in}_1$ necessarily). The encapsulation operator ($\tau$) are needed to enforce the process producer and the process consumer to communicate, and to make the communication action $c_{m}(d_a)$ internal.

IV. Verification of SystemC\textsuperscript{TL} Designs

In this section, we briefly describe how SystemC\textsuperscript{TL} design properties (e.g. safety property) can be verified using various formal methods.

\textbf{A. Analyzing SystemC\textsuperscript{TL} Designs Using Timed Automata}

A formal translation was defined in [4] from SystemC\textsuperscript{TL} to a variant (with very general settings) of timed automata [13]. The practical benefit of this translation is to enable verification of properties of SystemC\textsuperscript{TL} designs using existing verification tools for timed automata, such as Uppaal [17].

However, specifications of timed automata are not always trivial and intuitive for users not having a computer science background. In addition, variants of timed automata are used for different verification tools for timed automata. Users are required to adapt manually the settings of the variant of timed automata proposed in [4] for various verification tools.

\textbf{B. Formal Verification of SystemC\textsuperscript{TL} Designs Using the SPIN Model Checker}

In [5], an approach was introduced to use the SPIN model checker ([14] and [15]) as a verification engine for SystemC\textsuperscript{TL} designs, by translating SystemC\textsuperscript{TL} designs to PROMELA [16] that is the input language of SPIN. Among various formal verification tools, the SPIN model checker was chosen, because it is one of the most successful software tools that can be used for the formal verification of distributed software systems.

Furthermore, the input language of the SPIN model checker is PROMELA that is a popular language for building verification models. It is widely used in industrial and academic fields. Moreover, PROMELA is similar to the language C. This makes PROMELA easy to understand by verification engineers, researchers and even students.

V. Extensions for SystemC\textsuperscript{TL}

This section describes our on-going research works to develop mixed-signal system extension and to get a formal verification framework using existing verification tools.

\textbf{A. Mixed-Signal Systems}

A number of research works (e.g. [20] and [21]) has already been done to develop SystemC extensions for modeling and simulating mixed-signal systems. To our best understanding, there is still no formal semantics defined for those extensions. We are now defining the formal semantics (also in SOS style) in SystemC\textsuperscript{TL} for modeling...
mixed-signal systems. This semantics intends to support the development of system-level analog and mixed-signal specifications, and will be a conservative extension to the existing SystemC$^{\mathcal{FL}}$.

B. Formal Verification of SystemC$^{\mathcal{FL}}$ Designs Using the SMV/NuSMV Model Checker

Nowadays, formal verification of hardware plays an very important role in electronic industry. The formal verification approach proposed in subsection IV-B was not specifically used to verify hardware designs, because SPIN is a verification tool for software systems.

The SMV[18] and NuSMV[19] are well-known model checkers. They can be reliably used for the verification of industrial designs. Various successful applications of SMV and NuSMV to verify hardware designs can be easily found in the literature. We are currently defining the formal translation from SystemC$^{\mathcal{FL}}$ to the SMV language [18] that is the input language of SMV and NuSMV model checkers. This approach enables verification of properties of SystemC$^{\mathcal{FL}}$ designs using SMV and NuSMV model checkers.

VI. CONCLUSIONS

We gave the main aspects of the current status of the formal language SystemC$^{\mathcal{FL}}$, and showed some practical applications of SystemC$^{\mathcal{FL}}$. We also presented some possible extensions of SystemC$^{\mathcal{FL}}$. These extensions can be used for modeling mixed-signal systems and formal verification of hardware designs written in SystemC$^{\mathcal{FL}}$.

ACKNOWLEDGMENTS

The author would like to thank D.A. van Beek, P.J.L. Cuipers, M. Mousavi, M.A. Reniers, and R.R.H. Schifflers for many stimulating and helpful discussions. The author is grateful to J.C.M. Baeten for his support and encouragement.

REFERENCES