Bookmark file

Formal Semantics of Synchronous SystemC

Ashraf Salem
Computer & Systems Eng. Dept.
Ain Shams University
Cairo, Egypt

Abstract

In this article, a denotational definition of synchronous subset of SystemC is proposed. The subset treated includes modules, processes, threads, wait statement, ports and signals. We propose formal model for System C delta delay. Also, we give a complete semantic definition for the language's two-phase scheduler. The proposed semantic can constitute a base for validating the equivalence of synchronous HDL subsets.

1. Introduction

SystemC is a set of C++ classes that allows modeling hardware at different levels of abstraction from system behavioral to register transfer level. The SystemC Class library provides the constructs needed to model hardware timing, concurrency and reactivity [1]. This Class library supports Modules, ports, Signals, Processes, Hardware data types, Clocks, Waiting and watching.

Synthesis and verification tools are using subsets of hardware description languages [2][3], as these languages are defined mainly for simulation [4][5]. The same approach starts to be applied for SystemC by defining synthesisable subsets and description styles for synchronous designs [6]. The semantics of these subsets are given usually in terms of equivalent synchronous hardware elements; efforts are also done to define formal semantics for these subsets. Tassel [7] defined the VHDL simulation cycle in terms of higher order logic. Breuer et al.[8] proposed a denotational semantics for Unit-Delay VHDL. Olcoz et al. [9] used Petri-Nets to give operational definitions for the complete VHDL language. A number of different approaches for VHDL semantic definitions, including temporal logic, Boyer-Moore logic and process algebra have been reported in [10].

Mueller et al. [11] presented formal definition of the simulation semantic of SystemC in terms of abstract state machine; this semantic covered watching statements, signal assignments and wait statements.

In this article, we propose a denotational semantic for a synchronous subset of SystemC. The technique used follows the approach proposed in [12] for the semantic definition of a Synchronous VHDL subset. The originality of our work lies in the definition of a formal model for SystemC's two-phase scheduler [13]. We formulate the *update* and the *evaluate* phases using two function domains which express delta delay semantics.

The subset covered by our semantic definition includes: modules, method processes, thread process, clocked thread process, wait, ports, signals assignment and sensitive statement. For reason of space, we limit the data type treated to sc_bit, and the operators to &, |, ^ and ~. We leave out also the semantic definition of programming language constructs such as case statement and loops; such definition can be found in [14].

The rest of the paper is organized as follows. In section 2, a brief introduction to the denotational semantics is given. In section 3, formal model for delta delay is presented. The semantics of SystemC module is given in section 4. In section 5, the different types of SystemC processes are formulated. The semantic of the language scheduler is given in section 6, and finally our conclusions are presented in section 7.

2. Denotational Semantics

The denotational semantics [15][16] of a language consists of three parts: abstract syntax, semantic domains and semantic functions that relate the syntactic domain to the semantic domain. The abstract syntax is composed of a set of syntactic domains and a set of syntactic rules (BNF-like production rules). The syntactic domain is the set of language sentences whose structure is described by the production rules for a non-terminal.

For example, the syntactic domain ScModule represents the non-terminal SC_MODULE of SystemC. Module is composed of input/output ports declaration, signals declaration and a constructor. M is a typical module

1530-1591/03 $17.00 © 2003 IEEE

belonging to ScModule domain. The structure of this module is described by a syntactic rule.

$M \in$ ScModule
M ::= SC_MODULE(I) {IO; D; C}

Semantic domains describe the sets of value spaces of a language. For example the semantic domain *Signal* is a union domain representing sc_in, sc_out, sc_signal and sc_clock objects

$sg \in$ Signal = ScInput + ScOutput + ScSignal + ScClock

The test operator ?, decides if its first operand belongs to the domain indicated by its second operand. The predicate *i? ScInput* is true if i is sc_input port in a SystemC module.
Semantic functions, one for each syntactic domain, define the mapping between syntactic domain and semantic domain. For example the syntactic domain ScBit described by the rule: SC ::= '0' | '1' is mapped to semantic domain Boolean by the semantic function SC:

SC: ScBit -> Bool
SC['0'] = false
SC['1'] = true

3. Formal Model of Delta Delay

SystemC uses two-dimension time model: physical time and delta cycle [13]. Physical time is represented by 64 bit-unsigned integers. Delta delay is used to order the execution within the simulation cycle. The simulation cycles of SystemC consists of two phases *evaluate* and *update*. More than one delta cycle may occur at a particular physical time. Delta delay is used to separate the *evaluate phase* and the *update phase*.
To express the semantics of the delta delay, two values of each signal are needed in a specific physical time. One to store the value calculated in the evaluate phase. The second is used to hold the value of the signal after the update phase. To represent these two values we propose two function domains:

c ∈ Current = Signal -> Value
n ∈ New = Signal -> Value

Current domain contains the functions returning the values in evaluate phase and *New domain* contains functions returns the values of the signals after the update phase. Where *c(sg)* is a function belonging to the domain *Current* that returns the value of signal *sg* belongs to the domain *Signal*.

To change the current or the new value of the signal, the update [14] operator $\Leftarrow$ is used. It is a generic over function domains. For a given function f1, an element x of the domain f1, and an element y of its co-domain, f1[x $\Leftarrow$ y] produces a new function f2 identical to f1 except that f2(x) = y.
For instance, the update operator on the domain Current is identified as follows:

$\Leftarrow$: Current X Signal X Value -> Current

Let *sg* be a signal, c[sg $\Leftarrow$ true] is the same function as c but with *c(sg)* equal to true. To illustrate the usage of the *new* and *current* domains in the definition of SystemC semantics, consider the AND gate described by the following SystemC module.

```
SC_MODULE(andgate) {
    sc_in<bool> a1;
    sc_in<bool> a2;
    sc_out<bool> b;

    void andfunc{b = a1 & a2;};

    SC_CTOR (andgate){
            SC_METHOD (andfunc);
            Sensitive << a1 << a2;}
};
```

The module *andgate* has two input ports and one output port. The constructor *andgate* execute the SystemC method *andfunc* to calculate the gate output, the constructor is sensitive to the two inputs a_1, a_2. Each of the three signals a_1, a_2 and b has two values separated by a delta delay. During the *evaluation phase*, only the *new* domain is changed, by assigning the value of the expression c(a1) & c(a2) to n(b).

$$n[b \Leftarrow (c(a1) \,\&\, c(a2))]$$

In the *update phase*, i.e. after a delta delay, the new value of *b* becomes its current value.

$$c[b \Leftarrow n(b)]$$

4. SystemC Module Semantics

Module is the basic building block in SystemC. It is composed of ports, signals, variables, constructor, functions and processes. The module semantic can be viewed as a set of functions, each function models one output. The parameters of these functions are the module's inputs. These functions change the values of the current

simulation environment, i.e. signals and variables, as a result of the execution of non-declarative statement of the module, namely the module constructor.

Formally, The environment can be expressed using the domain *SigStore*, which is a union of two domains: the product domain *(New x Current*), which represents the signal's value pairs and the function domain *Store*.

$$ss \in SigStore = New \times Current + Store$$

Store is a domain of functions associating variables with their values.

$$s \in Store = Variable \rightarrow Value$$

The following syntactic domain and syntactic rules describe SC_MODULE syntax.

M ∈ ScModule
D ∈ Declaration
I ∈ Identifier
IO ∈ InputOutput
C ∈ ScCtor
P ∈ ProcessStatement
CST ∈ Cstatement

M ::= SC_MODULE(I) {IO; D; C}
IO ::= sc_in<bool> I | IO1; IO2
D ::= sc_signal<bool> I | sc_clock <bool> I ;
C ::= SC_CTOR (I) {P; CST}

The semantic function M expresses the semantic of SC_MODULE. The function modifies the value of the pair (new, current) of the signals and the varaible values (store) by evaluating the class constructor C. The constructor C is composed of process statement P followed by sensitive statement S. In the subset treated in this paper only sc_in and sc_out represented by the domain InputOutput are allowed.
The *constructor* semantic function C is calculated; by composing the *process function* P and the *C statements function* CST against the SigStore ss using the function compositions operator o.

M : ScModule -> SigStore -> SigStore
$M[[SC\_MODULE(I)\{IO; D; C\}]]_{ss} = C[[C]]_{ss}$

C : ScCtor -> SigStore -> SigStore
C[[SC_CTOR(I){PS; CST}]]ss = CST[[CST]]ss o P[[P]]ss

5. SystemC Process Semantics

Processes model the concurrent behavior of hardware in SystemC. Three types of processes exist in the language: SC_METHOD, SC_THREAD and SC_CTHREAD. Processes are activated when signals stated in the sensitive statement change. Threads can use wait statement to identify events, which are able to activate them. The main restriction in our synchronous subset is the nature of the signals allowed on the sensitive statements and wait statements.
We propose two types of processes: *Combinational Process* and *Synchronous Process*. In a combinational process, all input signals must be found on the sensitive statement of SC_METHOD. In a synchronous process, clock signal shall be found on the sensitive statement of SC_METHOD, no other signals may be found in this statement in this case.
In a SC_THREAD modeling synchronous process wait statement shall exist as the last statement in the thread and shall include a clock condition. SC_CTHREAD models synchronous process by having it triggered on the edge of the clock.
The Syntactic domains given in figure (1) and the syntactic rules figure (2) define the abstract syntax of the two types of the process. The combinational process PC is composed of the method process MP followed by combinational sensitive statement S. Synchronous Process PS is composed of a method process followed by sequential sensitive statement SS, thread process TP or a clocked thread process CP. Method process MP, Clocked thread process CP or Thread process TP are composed of C++ function CF or TF. TF shall contain wait statement modeling the rising edge of the clock.
For the sake of clarity, only C assignment statements AC are allowed inside C functions.

P ∈ ProcessStatement
PC ∈ CombinationalProcess
PS ∈ SynchronousProcess
MP ∈ MethodProcess
TP ∈ ThreadProcess
CP ∈ ClockedProcess
CF ∈ Cfunction
TF ∈ ThreadFunction
S ∈ CombinationalSensitiveStatement
SS ∈ SynchronousSensitiveStatement
SL ∈ SensitivityList
CST ∈ CStatement
TST ∈ ThreadStatement
AC ∈ AssignmentStatement

Figure 1: Process Syntactic Domains

P ::= PC | PS
PC ::= MP; S
PS ::= MP ; sensitive_pos << I; | TP | CP
MP ::= SC_METHOD(CF)
TP ::= SC_THREAD(TF)
CP ::= SC_CTHERAD(CF,I.pos())
CF ::= I () {CST}
TF ::= I () {TST}
S ::= sensitive SL
SL ::= << I | SL1 SL2
SS ::= sensitive_pos << I
TST ::= CST; wait_until (I.event() && I.delayed() == true)
CST :: = AC | CST1; CST2
AC :: = I = E
E ::= '0' | '1' | E1 & E2 | E1 | E2 | ~ E1 | E1 ^ E2 | I | (E1) |

Figure 2 : Process Syntactic Rules

The following SC_MODULE [1] models D Flip-flop using the proposed subset. The *Synchronous Process dff* is sensitive to the signal clock of type *sc_clk* and the method *df* contains only assignment statements.

```
SC_MODULE(dff)
{
   sc_in<bool> din;
   sc_in_clk  clock;
   sc_out<bool> dout;

   void df(){dout = din;};

   SC_CTOR(dff)
   {
       SC_METHOD(df);
       sensistive_pos << clock;
   }
}
```

The semantic definition of the two types of processes in the proposed subset is given by two semantic functions: PC & PS. PC describes the semantic of the combinational process. It checks the signals on the sensitivity list SL and compares it with the input port list. In case of identical lists, the function MP is executed.

PC : CombinationalProcess -> SigStore -> SigStore
$PC\ [[MP; S\ ]]_{ss} = S[[S]] \rightarrow MP[[SC\_METHOD(CF)]]_{ss}, \perp$

S : CombinationalSensitiveStatement-> Bool
$S[[sensitive\ SL]] = (\forall\ in \in Sc\_Input, (in \in SL)) \rightarrow True, False$

The function MP is executed by calling the semantic function of the sequence of C statements.

MP : MethodProcess -> SigStore -> SigStore
$MP[[SC\_METHOD(CF)]]_{ss} = CF[[CF]]_{ss}$

CF gives the semantic of C statements. For sake of clarity, we limit our self here to assignment statement. Two types of objects can be found on the LHS of the assignment, signals and variables. The semantic function AC checks the nature of the object and assigns the value of the expression of the RHS to the new value of the object in the case of output or internal signal, or to store in the case of the variables.

CF : Cfunction -> SigStore -> SigStore
$CF[[CST]]_{ss} = CST[[CST]]_{ss}$

CST : Cstatement -> SigStore -> SigStore
$CST[[CST]]_{ss} = AC[[AC]]_{ss}$

AC : AssignmentStatement -> SigStore -> SigStore

$AC[[I = E]]_{ss} = i?$ (ScOutput + ScSignal) *and* $e?Bool \rightarrow n[i \Leftarrow e]$, $i?Variable\ and\ e?Bool \rightarrow s[i \Leftarrow e], \perp$ *where* $e = E[[E]]_{ss}$

The following function E defines the semantics of the identifier and '&' operator, as an example of the semantic definition of the expressions.

E: Expression -> SigStore -> Bool
$E[[I]]_{ss} = i?Signal \rightarrow c(i), i?Variable \rightarrow s(i), \perp$
$E[[E1\ \&\ E2]]_{ss} = e_1?\ Bool\ and\ e_2?Bool \rightarrow e_1\ and\ e_2, \perp$

The semantics of the three types of the synchronous process are given by the semantic function PS. The subset treats only single clock design with positive edge activation. The semantic function *PS* calculates the signals value pairs and the variables values when the current value of the clock signal is equal to true. Also the function checks the nature of signal on the sensitive statement in the method process and on the clocked thread parameter. The init variable is used to differentiate between the initialization phase and the simulation phase. It is set to 1 when the SystemC scheduler enters the initialization phase to stop the execution of the clocked threads and assignments.

PS : SynchronousProcess -> SigStore ->SigStore

$PS[[MP\ ;\ sensitive\_pos << I]] =$
$i \in ScClock \rightarrow ((c(i)\ and\ (init=0)) \rightarrow MP[[SC\_METHOD(CF)]]_{ss}, n), \perp$

The semantic function CP defines the clocked thread process; it checks the value of and the type of the signal I and then execute the C function CF, if the clock signal value is equal to 1 else it returns the new domain n.

$CP[[SC\_CTHERAD\ (F,I.pos)]]_{ss} =$
$i \in ScClock \rightarrow\ ((c(i)\ and\ (init=0)) \rightarrow\ CF[[CF]]_{ss}\ ,\ n),\ \perp$

The function TF gives the semantic of the thread process, it checks the nature of the object of the wait_until statement and then evaluates the thread statements of the current value of the clock signal is true and the init variable is equal to zero.

TF : ThreadFunction -> SigStore -> SigStore
$TF[[IO\ \{TST\}]]_{ss} = TST[[TST]]_{ss}$

TST: ThreadStatement -> SigStore -> SigStore
$TST[[CST;\ wait\_until\ (I.event()\ \&\&\ I.delayed()\ ==\ true)]]_{ss} =$
$i \in ScClock \rightarrow ((c(i)\ and\ (init=0)) \rightarrow\ CST[[CST]]_{ss}\ ,\ n),\ \perp$

6. **SystemC Scheduler Semantics**

SystemC uses an evaluate-update scheduler. *Notify* method causes the event to be notified in the evaluate phase of the next delta cycle. *Request_Update* causes *Update* method to be called on the update phase of the current delta cycle. Notify is used for both timed and immediate notifications. Initialization phase is performed before the main simulation loop. All processes except the clocked thread are executed in this phase. Physical time is advanced only if there are timed notifications. Simulation ends when no more timed notifications exist [13].
We formulate the *evaluate phase* by calculating the module function M.

Evaluate: Module -> SigStore -> SigStore
$Evaluate\ (ss) = M[[\ SC\_MODULE(I)\ \{IO;\ D;\ C\}]]_{ss}$

Update phase assigns the signals' *current* and *new* values to their new values calculated in the *evaluate phase*.

Update: SigStore -> SigStore
$Update\ (ss) = \forall sg \in Signal,\ sg \Leftarrow (n(sg),\ n(sg))$

Scheduler *initialization phase* is modeled by function similar to update. The difference is assigning the variable *init* '1' to disable the execution of the Clocked threads during this phase and then turning it to '0'.

Initialize: Module -> SigStore -> SigStore
$Initialize\ (ss) = init \leftarrow 1;\ M[[\ SC\_MODULE(I)\ \{IO;\ D;\ C\}\ ]]_{ss};$
$init \leftarrow 0$

Figure (3) gives the semantic definition the scheduler. The function *Initialize* is called first to execute all process except processes calculating the memory elements. Then, *Evaluate* function is called to execute all processes including the clocked ones. This will change the new component in each signal. The update of the current component will take place on the *Update function*; the delta_count will be incremented by one after the update phase. If the SystemC module contains a clock signal, the scheduler will then set the current value of the clock to true; then it calls the function evaluate to compute the module, then it resets the clock to false, and it increments the time by one clock period. The multiple delta cycle in the same physical time instant is modeled by new simulation cycle if the signal's current value is not equal to the signals' new value. This is expressed by:

$while\ not\ (\forall sg \in Signal,\ n(sg) = c(sg)))$

The simulation ends when the sc_now reaches the simulation_end_time.

```
1.  Scheduler: Module -> SigStore -> SigStore
2.
3.  // Intialization Phase
4.  ss = Initialize(ss);
5.  do
6.  {
7.      // Evaluate Phase
8.      ss = Evaluate(ss);
9.
10.     // Update Phase
11.     ss = Update(ss);
12.
13.     // Next delta Cycle
14.     delta_count++;
15.     do
16.     {
17.       if (∃k ∈ Signal and k? Clock)
18.       {
19.        // Delayed events (clocked assignment)
20.        ss[ k ⇐ (true, false)];
21.        ss = Evaluate(ss);
22.        ss[k ⇐ (false, false)];
23.
24.       // Simulation time advances
25.        sc_now = sc_now + clock_period;
26.       }
27.     }
28.     // New Simulation Cycle
29.     while not (∀sg ∈ Signal, n(sg) = c(sg));
30. }
31.
32. // If no more timed notifications, simulation
33. // is finished
34.
35. while (sc_now < simulation_end_time);
```

Figure 3: Scheduler Semantic Definition

7. Conclusions

In this paper, formal semantic of a synchronous subset of SystemC is proposed. The *delta cycle* has been formulated using *current* and *new* function domains. Physical time is modeled on the clock period level. A description style based on defining two types of processes: *Synchronous* and *Combinational* is proposed. The semantics of the SystemC methods and threads limited to this description style are defined. The *evaluate* and *update* phases of SystemC scheduler have been formulated for both timed and immediate notifications. We believe that our formalism can establish a common foundation for expressing the semantics of basic primitives of Synchronous HDL, mainly signal assignment, clock, process, delta cycle and the distinction between signals and variables.

8. References

[1] Open SystemC Initiative, SystemC Version 2.0, User's Guide, www.systemc.org, 2001.

[2] IEEE, IEEE standard for VHDL Register Transfer Level (RTL) synthesis, IEEE Std. 1076.6**,** 2000.

[3] D. Borrione, L. Pierre, A. Salem, "Formal verification of VHDL Descriptions in the PREVAIL environment", IEEE Design and Test of Computers, June 1992.

[4] IEEE, IEEE Standard VHDL Language Reference Manual, IEEE Press, 1993.

[5] P. Moorby, D. Thomas, "The Verilog Hardware Description language", Kluwer Academic Publishers, 2002.

[6] Synopsys, "Describing Synthesisable RTL in SystemC", Version 1.1, Synopsys, January 2002.

[7] J. Van Tassel, "A Formalization of VHDL Simulation Cycle", Technical Report 249, University of Cambridge, March 1992.

[8] P. Breuer, L. Fernandez & C. Delgado Kloos, " A simple denotational semantics, Proof theory and a validation condition generator for unit-delay VHDL", Formal Methods in System Design, Volume 7, August 96.

[9] S. Olcoz, J.M. Colon," Towards a formal semantics of IEEE Std. VHDL 1076", Proceedings EuroDac'93 with EuroVHDL'93, Hamburg, September 1993.

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

[11] W. Mueller, J. Ruf, D. Hoffmann, J. Gerlach, T. Kropf, W. Rosenstiehl, "The Simulation Semantics of SystemC", DATE 2001, Munich, 2001.

[12] D. Borrione, A. Salem, "Denotational Semantics of a Synchronous VHDL subset", Formal Methods in System Design, Volume 7, August 96.

[13] Open SystemC Initiative, Functional Specification of SystemC, Version 2.0, www.systemc.org, 2001.

[14] R. Tennent, Principles of Programming Languages, Prentice-Hall, Englewood Cliffs, 1981.

[15] D. Schmidt, Denotational Semantics, W. Brown Publishers, Dubuque, 1988.

[16] M. Gordon, The Denotational Descriptions of Programming Languages, Springer-Verlag, 1979.