An overview of SystemC and its underlying Simulator

Shreyas Ramakrishna
shreyas.ramakrishna@vanderbilt.edu
May 3rd, 2018

***Abstract*— This paper intends to provide an overview of SystemC, which is a system design language that has evolved due to the lacking of a common platform for hardware and software design of a system. The introduction of this language has made improvements in the areas of system productivity, and cost reduction by introducing co-simulation of both hardware and software designs. This works provides a comprehensive review at the concepts of SystemC, by explaining the building blocks, programming structure, applications and then making a transition into some important concepts of scheduler, simulation semantics and time. SystemC is a co-operative, event driven simulator which provides an overall co-operative scheduling mechanism to the underlying simulator. The idea of internal delta cycle is also introduced to work within the frame of a simulation time. This is a review paper which is intended to provide the readers a comprehensive look at the internals of SystemC.**

keywords: **Methods, Threads, events, wait, sensitivity list, scheduler, context switching, Inheritance, Encapsulation.**

I. INTRODUCTION

SystemC is a system design language that has come into existence as a need for a language which could improve the overall productivity and time to market of electronic systems. Much before 2001, when the idea of SystemC was coined, system design had two separate and evident phases of hardware and software designs. The software design teams had to wait for the hardware team to provide them a prototype over which they could build their software programs. Having two teams were particularly disadvantageous owing to cost, communication (synchronization) and also the overall time to market and productivity were hit. To overcome all of these disadvantages, SystemC was coined. It is basically a C++ library having some additional features to make it suitable as a system design language.

Some widely used applications of SystemC design language is System-level modeling, Architectural exploration, Software development, Functional verification and High-level synthesis. Some interesting features of the simulator like co-operative multitasking and event-driven scheduling makes it highly desirable for the concurrent design of a system.

The SystemC scheduler is event driven, in which the scheduling works based on the triggering of an event. In such a system, the scheduler context switch happens when a user defined process is triggered by an event, and this action allows a running process to wait in a queue and the triggered process to enter the running state. An elaborative explanation is provided in the sections to follow. The scheduler is also managed with the help of co-operative multitasking action, which allows for a co-operation among the multiple processes running.

Understanding the complete operation of the scheduler would help the designer to manage his processes to work in a co-operative fashion. The user gets the full control of the scheduling operation within. Knowing all of the scheduling information would help the user build upon the various applications of transaction level modeling, and system level modeling.

The remaining section of this paper is organized as follows. Section II is a review of few related literatures. In Section III, we present a brief comparison of different languages used for system level design. In Section IV, a top level overview of SystemC is given. Section V discusses about the underlying simulator semantics. Section VI describes the simulation semantics in terms of ASM. In Section VII, a critical review of the tool in terms of composability is made. Following this is the concluding remarks which are presented in Section VIII.

II. RELATED WORK

There have been a number of previous works which reviews SystemC from different aspects. The report [8] provides a good overview into the different versions of the library. It reviews various features of SystemC 1.0 which had inbuilt support for various features like fixed precision data types, concurrent processes,communication channels, reusable modules, along with a primitive simulation kernel. There was however extended with the version 2.0 which further added features of generalizing communication channels, supporting model's of computation and better synchronizing support. This work has provided a clear separation on the different features implemented by the two versions. It also outlines about the different layers built by the features of version2.0.

Similar to the above cited work, the book [7] by springer provides a complete insight into SystemC. It introduces the different aspects of data types, notion of time, modules, concurrency, communication and channels. This work would provide a complete understanding into the above mentioned concepts of SystemC. However, this work lacks to provide insights into the practical programming constructs of the language.

Understanding the top level concepts would allow the user to build applications like Transaction level modeling, System level modeling, Analog/ Mixed signal, SystemC verification

and configuration, control and inspection (CCI). The work [1] builds on the transaction level modeling application. Explains the key communication concepts, TLM channels, some application programming constructs along with examples for readers to build and use their TLM applications.

However understanding only the top level architecture would not completely aid the readers in understanding the internal concurrency support that the SystemC simulator has to offer. The work [7] provides an extended overview of the concurrency supported by the simulator. It discusses the related concepts of threads, modules, events, wait and sensitivity lists, which aid the SystemC simulator in providing an event driven and co-operative scheduling. This chapter provides an extensive explanation about the system concurrency which would make readers understand the parallel structure of the programming structure in SystemC. The other interesting thing that could be concurred from this work is the composability of different processes running concurrently.

The work [3] also provides an insight into SystemC simulator semantics by explaining the underlying operation in terms of abstract state machines (ASM). This breaks down the complex internal simulator operation as simple abstract state machine transitions, having an initial state and a final state.

All of the above mentioned works have successfully provided extensive explanation into the different aspects of SystemC. This work is basically going to assimilate the ideas from some of the above works to provide the readers an understanding of the current state of SystemC library. The aim of this work is also to show how the composition works between the different actors involved.

III. LANGUAGE COMPARISON

Previously VHDL and Verilog were the two widely used hardware description languages (HDL), which worked at lower abstraction behavioral levels, but lately, there has been an increase in other languages that work at higher levels of abstraction. Problems like design complexity, and faster time to market were the driving forces behind looking for a new system design language which could reduce these drawbacks. Some of new high-level languages that were built to overcome the drawbacks are, SystemC, System-Verilog, and SpecC. These languages have some common programming syntax with powerful constructs, making modeling and simulation of systems much simpler. Among these, SystemC has been widely used in the areas of system level design, high level synthesis and architectural exploration. Figure 1 shows the different abstraction levels at which the different HDL and SystemC work at. It can be seen that SystemC works at much higher level of abstraction, which makes it easier for the users to work with.

IV. PART1: TOP LEVEL OVERVIEW OF SYSTEMC

This section provides a top level understanding into different components of SystemC. The work [11] provides a

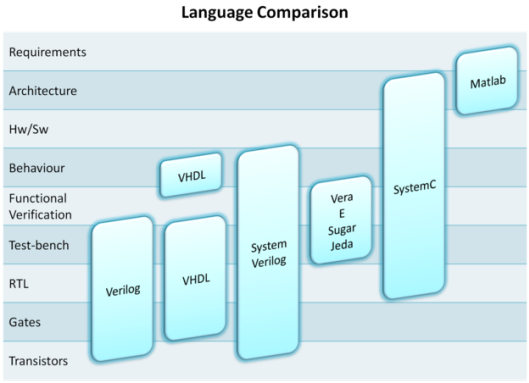

Fig. 1. Language Comparison

complete insight into the different blocks of SystemC and its syntax. Some of the important building blocks are discussed below:

A. Modules

Modules are the basic building block of SystemC. This allows the designers to split bigger and complex systems into smaller and manageable parts. In terms of programming language, modules are just like class. So it could be either public or private. This means, modules can hide their data from other modules by making it private. However using public interfaces to other modules, could make the entire system easier to change and easier to maintain. Similarly using modules provides the advantage of hierarchy. This block is similar to module in Verilog and Entity in VHDL. Below is list of the different parts of a module:

Ports, Internal Variables, Constructor and Internal Methods.

B. Ports

Ports are responsible for passing the data to and from the processes of a module to processes in other modules. Three types of port declarations are:
in : Input Ports
out : Output Ports
inout : Bi-direction Ports

C. Signals

As discussed above, ports are used for communicating between different modules. But Signals are used for communicating within the same modules.
Syntax : sc_signal type variable;

D. Process

All of the SystemC functionalities are implemented in the Processes. Processes are pieces of codes which run concurrently with other processes. This is similar to the always block of Verilog.

E. Threads

Threads are also one different kind of process, which keeps executing until it is stopped by an event. Threads can be controlled by the user by sending events. They can also be suspended and reactivated by the use of wait () function. So once a thread is triggered by an event, it starts executing and it can be suspended by using a wait(). Upon calling wait, the thread moves from the running queue to the waiting queue.

A special class of threads is called as clock threads CThreads. Original threads are triggered by events, with no notion of it being executed at particular clock edges. However, clocked threads are triggered at particular clock edges, either positive edge or negative edge.

F. Channels

SystemC uses channels to communicate among modules. There are two types of channels, namely

(a)**Primitive:** Is derived from from sc_prim_channel, and can thus have access to the scheduler (evaluate update). sc_mutex, sc_fifo and sc_semaphore.

(b)**Hierarchical:** Is derived from sc_module, and can thus have all the features of an sc_module (processes, hierarchy etc).

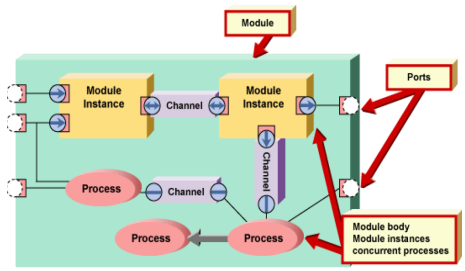

Fig. 2. SystemC blocks

The figure above from [12] shows all the components discussed earlier. Shows how channel is formed between modules for communication.

G. Hardware Data type

Along with the C++ data types, SystemC also supports hardware data types, which makes it such a powerful system design language. Some of the supported hardware data types are: Bit Type, Logic Type, Arbitrary Width Bit Type, Arbitrary Width Logic Type, Signed Integer Type, Unsigned Integer Type, Arbitrary Precision Signed Integer Type and Arbitrary Precision Unsigned Integer Type.

SystemC data types are represented as: sc_lv, sc_logic which provides variables to be assigned with 0,1,X and Z. Here X represents unknown state value and Z represents high impedance.

syntax:

sc_bv <100>x,y represents bit and bit vectors.

sc_int<48>x represents fixed and arbitrary precision.

H. Time & Clocks

Time is also a new feature added by SystemC into the basis C++ programming construct. Timing of signals plays a very important aspect when dealing with hardware. Both VHDL and Verilog supports time to monitor the change in signals. This feature of time is extended from Verilog into the SystemC construct.

syntax:

sc_clock clock(my_clock, 10, 0.5, 1)

sc_clock clk(clk, 10, SC_NS)

I. Programming Structure

SystemC follows the C++ programming construct. It uses features of inheritance, encapsulation to perform code re-usability.

Every program starts with a class SC_MODULE. Every module must have an actor SC_CTOR with the same module name. As discussed earlier, Ports are used for external communication. Also every module should have one process to give functionality to the module.

```
SC_MODULE(<module name>)
{
sc_in<data type> port1; sc_out<data type> port2; ........//multiple port declaration

<return type> <process name> (<argument type>); ......//can accommodate multiple processes
.....................................................................................//any other codes

SC_CTOR(<module name>)
{
SC_METHOD(<process name>);//immediate call to the process 'process name'
sensitive << <sensitivity list>;
...........//any other code
}
};
```

Fig. 3. General program structure

The figure above shows the simple SystemC program structure. The different components of modules, constructors, methods and process are shown. Using this the figure below implements a simple and gate.

```
//A simple example of and gate in SystemC
#include "systemc.h"

SC_MODULE(and_gate)
{
sc_in<sc_bit> x;
sc_in<sc_bit> y;
sc_out<sc_bit> z;

void prc_and_gate() {z=x & y;}//process which gives functionality

SC_CTOR(and_gate)
{
SC_METHOD(prc_and_gate); //process is called here
sensitive << x << y;
}
};
```

Fig. 4. And gate program structure

J. Transaction Level Modeling

As defined in [1], Transaction-level modeling (TLM) defined as a high-level approach to modeling digital systems where details of communication among modules are separated from the details of the implementation of functional units. It is also seen as an abstraction of communication among computation modules. In simple words, TLM could

be thought of as Communication is separated from computation.

TLM builds on top of the SystemC construct where, the communication mechanisms such as FIFO are modeled as channels, and are presented to modules using SystemC interface classes. Here, the transaction requests take place by calling interface functions of these channel models, which encapsulate low-level details of the information exchange. At the transaction level, the emphasis is more on the functionality of the data transfers and less on their actual implementation.

The TLM channels of tlm_fifo and tlm_rep_resp_channels are built on the underlying SystemC channels, with additional features of unidirectional or bidirectional reads and writes, which could either be blocking or non-blocking.

Motivation behind the use of TLM are: To provide an early platform for software development along with the hardware development, System Level Design Exploration and Verification and, the need to use System Level Models in Block Level Verification.

This section introduced the various building blocks of the language along with explaining the simple SystemC program structure. The next sections would explain the internal simulator semantics.

V. PART2: SYSTEMC SIMULATION SEMANTICS

This section would provide a complete insight into the underlying simulator and its related concepts.

A. SystemC Macros

Methods, Threads and CThreads (clocked threads) are the three building block for concurrent process execution in SystemC. SC_Methods are repeatedly called within the sc_modules, they allow for code re-usability. Threads are also processes which are used to perform user defined functions. The only difference between the two is, threads can be interrupted externally by the users using events, waits but methods cannot be interrupted. There is also a third actor called CThread, which are similar to threads but are sensitive to a clock. These three are the processes defined by the users and are run by the system scheduler.

B. Events

Events are key for an event-driven simulator like SystemC. An event is something that happens at a specific point in time, it has no value and no duration. Processes wait for an event to change its state from running to wait or vice versa. However, if no processes are waiting to catch it, the event goes unnoticed. The figure 5 shows how simple event triggering works.

C. Sensitivity list & notify

Events, sensitivity lists and notify are important concepts for concurrency in the simulator. Sensitivity list controls when the input signals in the system are going to be evaluated, they are similar to the one in Verilog. There are

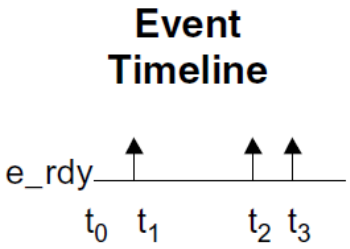

Fig. 5. Events Triggering

two types of sensitivity list namely,
Static sensitivity: implemented by applying the SystemC sensitive command to processes at elaboration time.
Dynamic sensitivity: lets a simulation process change its sensitivity on the fly.
Notify is an event function which is used to create immediate notification or a notification based on events.
The simulator will use events and notify in the context switch of events, which will be discussed in the future sections.

D. Event driven simulator

Event driven simulators are ones which are triggered by happening of an event. SystemC simulator is also system driven, in which the threads or methods are triggered by happening of some events. In these simulators, concurrency is not real concurrent process execution, however they are simulated concurrency, which rather work like co-operative multitasking.

E. Co-operative multitasking

As we know there are two types of multitasking, co-operative and preemptive. In co-operative multitasking, the processes/ threads executes for sometime and then willingly co-operates for the execution of another process. However, in preemptive multitasking a process will not willingly handover to the execution of another process, but the scheduler takes control of managing the scheduling for the two processes.

In case of SystemC simulator co-operative multitasking has been used by the scheduler. So, this allows some flexibility and control to the users to switch between the various user defined processes. In this case the processes transfer the execution control with the help of predefined events. Figure below shows a simple co-operative and preemptive multitasking.

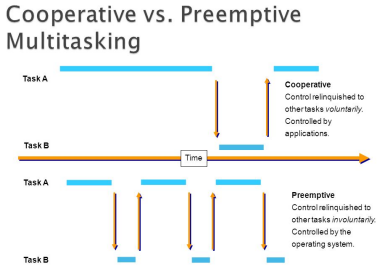

Fig. 6. Co-operative vs preemptive multitasking

F. The Simulation Kernel

As discussed earlier, the SystemC simulation kernel is event driven which works in a co-operative multitasking fashion. It has two major phases of operation called elaboration and execution. However there also exists another phase called post-processing. The figure below from [7] shows the different simulation phases.

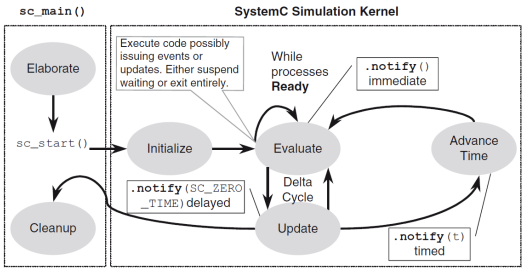

Fig. 7. Simulation phases

Elaboration phase: In this phase, the following initializations are performed.

The structures needed to describe the interconnections of the system are connected. It also establishes hierarchy and initializes the data structures. Along with this it also creates instances of clocks, design modules, and channels that interconnect designs. Additionally, it also invokes the code to register processes and perform the connections between design modules. After all of these initialization, the sc_start() invokes the simulation phase.

Execution Phase: Once the elaboration phase has been performed and the simulation has started, the control is transferred to the simulation kernel, which orchestrate the execution of the different process in a simulated concurrency fashion.

As described in [7] after the simulation starts,all the processes are randomly invoked during initialization. After initialization, a process is executed when it is triggered by events it is sensitive to. Several processes created by the user may begin at the same instant in simulator time.

Since there may be multiple user processes waiting for the execution, all the processes are evaluated and then only their outputs are updated. The evaluate- update cycle is referred to as a delta-cycle. There may be multiple delta cycles due to the occurrence of multiple processes in one simulation cycle. After all the processes are executed and if no processes need to be executed at that instant of time, then the simulation time is advanced. Thereafter When no additional simulation processes need to run, the simulation ends.

Post Processing Phase: After the simulation has finished, the control returns and the post-processing phase begins. In this phase data created during the simulation may be read or otherwise handle the results of simulation.

As seen from the above mentioned phases, the execution phase is the most important phase where the simulation kernel takes the control and performs all the context switches. Let us look at some of the important concepts which lead to context switches and then discuss to the important concept of delta cycle.

1) ***Wait****:* As we have seen throughout this work about the co-operative nature of the simulation kernel. The processes voluntarily gives up the execution upon receiving a wait from another process which needs to execute. So, if a process is currently executing and if another process is triggered by an event and wants to execute, then the process requiring the schedule will send a wait to the current executing process. Upon receiving the wait, the current process will stop executing and wait for its execution turn.

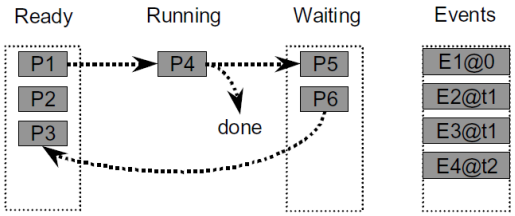

Fig. 8. Multiple Process Execution

The figure above from [7] shows that different queues are maintained for the different processor states. So, the processes have three different states of ready, running and waiting. So depending on the different processor states, they will be place in the respective queues. Similarly there is also an event queue which holds all the events which will trigger the process to change its state. The significance of this figure is it explains the co-operative nature with which the user defined processes hands over the control to the other process. Similarly it explains the event driven nature of the process execution.

2) ***Delta Cycle****:* There are two important concept relating to the simulation kernel. The first is the simulation time. This is the time that the user can see, when he runs the simulation. Within this individual simulation time is the concept of delta-cycle. As discussed earlier an execution-update cycle is called as a delta cycle. So, there can be multiple delta cycles depending on the number of processes running and the number of events, wait calls interrupting the process execution. So, only when all the processes are done with their execution, then only the delta-cycle gets completed and the simulator time is updated. The figure below from [7] shows the concept of delta cycles.

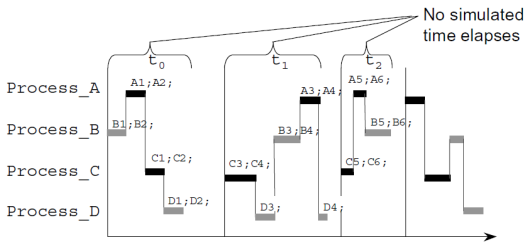

Fig. 9. Multiple Delta Cycles

It can be seen from the figure that the ordering of the process execution may differ, this is completely in the control of the scheduler and it cannot be predicted as to which process will be executed and in which order. This figure also shows that even though there are multiple delta cycles have been executed, the simulation time is not updated but remains the same.

This section explains the different concepts involved with the simulation kernel. The important lessons of simulation time and delta cycles are discussed along with their co-relation. There is this smooth synchronizing relationship between the delta cycles and simulation time. So, user can be very sure that there can never be a miss of any process execution before the simulation time has been updated.

VI. PART3: SYSTEMC SIMULATION SEMANTICS USING ASM'S

The previous section provided a comprehensive explanation of the simulator, this section reviews the simulator semantics in terms of abstract state machines (ASM). Inspired by the powerful nature of ASM's, this section reviews the work from Mueller et al., which provides an excellent overview into the SystemC simulator semantics.

A. Abstract State Machine

As described in [3], conventional computation models assume symbolic representations of states and actions, but ASM has the power to take a more liberal position. The expressive power of ASM allows any mathematical structure to be represented as a state. This results in a computational model that is more powerful and more universal than the standard computation models.

The underlying idea behind ASM is that, any algorithm can be modeled at its natural abstraction level by an appropriate ASM.This was developed as a methodology based on mathematics which allows algorithms to be modeled at their natural abstraction levels. The result is a simple methodology for describing simple abstract machines which correspond to algorithms. So the power of ASM allows any complicated mathematical model to be written in the form of simple state machines.

An ASM in terms of programming construct is a simple if-else loop.

if some condition is true **then**
  do some processing
else
  do some other processing
end if

So, this represents a simple state transition, where the if section represents the initial state and the else section represents the final state after transition. So every mathematical system can be represented by ASM. So, this powerful expressive nature of ASM could be used to represent the different phases of the simulator.

B. Simulation phases in terms of ASM

Similar to the above discussed section where we had different phases of the simulator, the figure below shows the execution phase of the simulator, which explains the different agents involved. This shows that every user process is represented as an agent and the simulator kernel is also represented as another agent. This also explains that only once all the user processes finish execution and suspends, then the kernel process is executed to update the simulation time and then the simulation cycle ends.

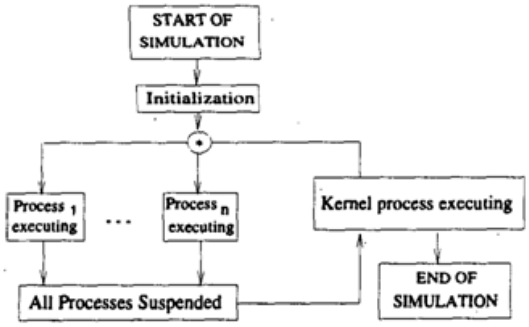

Fig. 10. Phases of Simulator

Similar to the above figure which explains the executing phase, the figure below expresses in depth the different states in the executing phases. Each of these states will be represented as ASM states in this section.

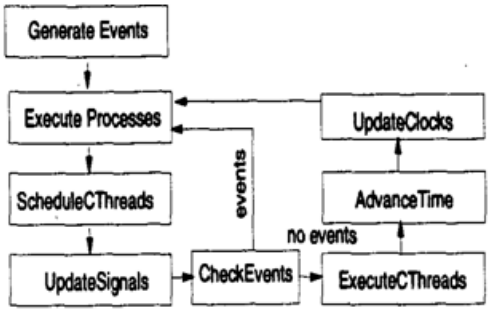

Fig. 11. Different states in Simulator Execution phase

The figure above is also representative of the well synchronized delta cycle and the simulation cycle. The inner loop which repeatedly checks for events represents the delta

cycle and the outer loop which updates the clock represents the simulation cycle. As discussed before the two loops are so well synchronized that the outer loop does not execute until the inner loop has completed executing all the events.

1) States and context switching: Each of the user process remains active until it is suspended by a wait statement or after all the processes have finished execution. Before getting active again, the process(threads, CThreads) first has to check for its watching conditions. However, the CThreads have to also check for the local watching. Methods which are also default process do not have any watching on them as they cannot be interrupted by the users. So, considering the life cycle of a process, the status of the process can be either active, suspended, global watching, and local watching. The figure below shows the different status of the process.

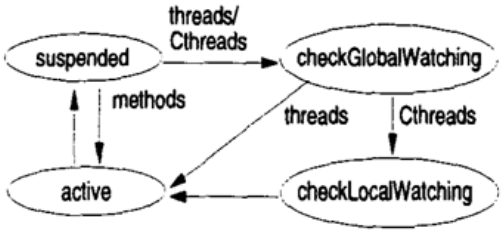

Fig. 12. Different Process status

2) Global Watching: Threads and CThreads generally have infinite loops and a designer would typically want a way to initialize the behavior of the loop or jump out of the loop when an event occurs. This requirement is accomplished with the help of a watching construct. The watching construct will monitor a specified condition as required by the user. So, When the specified condition occurs the control is transferred from the current execution point to the beginning of the process.

3) Local Watching: Local watching is applicable to CThreads only. This is used by the designer when there is a fine grain watching is required.

C. ASM of different simulation states

After understanding ASM's and their usage, we could apply them to explain the different execution states of the simulator. Some of the ASM representations are discussed further.

The generate state shown in figure 11, could be represented as an ASM as shown in the figure below.

$$AllProcessesSuspended \equiv$$
$$\forall p \in METHOD \cup THREAD \cup CTHREAD :$$
$$status(p) = suspended$$

Fig. 13. ASM for Generate State

The above figure referenced from [7] shows generate state as a simple state, which checks for the status of the processes and assigns it to suspended. Similarly the next state of execute process can also be represented as in the figure shown below.

if $AllProcessesSuspended$
then
  $ExecuteProcesses$
  $ScheduleCThreads$
  $UpdateSignals$
  $CheckEvents$
  $ExecuteCThreads$
  $AdvanceTime$
  $UpdateClocks$
endif

Fig. 14. ASM for Execute Process state

From the above figure, if all the user processes are suspended then it iterates through all the simulation states as shown in figure 11. Similar to this, the next state of ScheduleCThread can be represented as in the figure below.

$ExecuteProcesses \equiv$
var m **ranges over** $METHOD$
var t **ranges over** $THREAD$
var s **ranges over** $SIGNAL \cup CLOCK$
if $phase = executeProcesses$
then
  if $event(s) \wedge s \in sensitivityList(t)$
  then $status(t) := checkGlobalWatching$ **endif**
  if $event(s) \wedge s \in sensitivityList(m)$
  then $status(m) := active$ **endif**
  $phase := scheduleCThreads$
endif

Fig. 15. ASM for Execute Schedule CThreads

So, the ASM here shows the transition from the initial state of ExecuteProcess to the final state scheduleCThread.So, for all the running methods, threads and CThreads, if the threads are triggered by events, it moves to the check global state. However if the methods are triggered it moves directly to the active state. The next state of UpdateSignal are shown as in figure below.

$UpdateSignals \equiv$
if $phase = updateSignals$
then var s **ranges over** $SIGNAL$
  if $value(s) \neq newValue(s)$
  then $value(s) := newValue(s)$,
    $event(s) := true$
  else $event(s) := false$
  endif
  $phase := checkEvents$
endif

Fig. 16. ASM for Execute Schedule CThreads

The ASM has an initial state of updateSignal and final state of checkEvents. If there is a new value generated, it updates it and checks for pending events. If there are no pending events, then it moves to the next state.

This section explains the simulator semantics in terms of ASM's. Also, the figures described above shows the simple way of explaining the complex simulator state transitions in form of simple distributed state machines, which could be explained with simple if-else programming constructs.

VII. Part4: Critical Review about SystemC

Performing literature survey on the different aspects of SystemC and understanding the top level programming structure along with the robust underlying scheduler, has provided a comfortable understanding of the subject to make some critical points:

A. Deceptive underlying communication channels

The underlying channels between the modules are so deceptively established that the user defined modules does not know about it. The mind blowing concept of communication between the modules take place even without them knowing of an underlying channel.

The whole concept of communication through channels take place even without the user having to think about the transportation layer. The programming structure only have to define the different socket types and the protocol of reads, writes, blocking or unblocking. The library will create an underlying channel for the communication between the two modules.

B. Composability of multiple processes

Reviewing the sections on concurrency has provided an excellent insight into the smooth operation of the scheduler. The co-operative multitasking nature of the scheduler provides all control to the user to manage the smooth transition between the different processes. Critically putting thoughts into how multiple processes work with complete co-ordination makes it a fascinating library to work with.

The tool is so well built that the designer has complete control of the transition between the different process he defines. Most of simulators available are mainly preemption based and are in complete control of the underlying scheduler. It uses one of the underlying scheduling mechanisms of round robin (RR), First come First serve (FCFS), earliest deadline first (EDF).

If SystemC was built on a preemption based mechanism, then the whole composition of having multiple processes having simulated concurrency would have never worked. So it fascinates me to think about the underlying scheduler, which works in complete control of the designer. Using this beautiful composability and co-operative nature the designer could build a system on top.

C. Beautiful Synchronization between delta cycles and simulation time

SystemC simulator kernel has once again fascinated me in the context of its time concepts and synchronization. This concept of synchronization between delta cycles and simulation time is the most critical and well designed part of the simulator. This beautiful composability between the inner loop (ref figure 11), which indefinitely looks for different events to completes the process execution (delta cycle) and the outer loop which indefinitely waits for the inner loop to finish completely before updating time.The figure below represents a programmatic understanding of this composability.

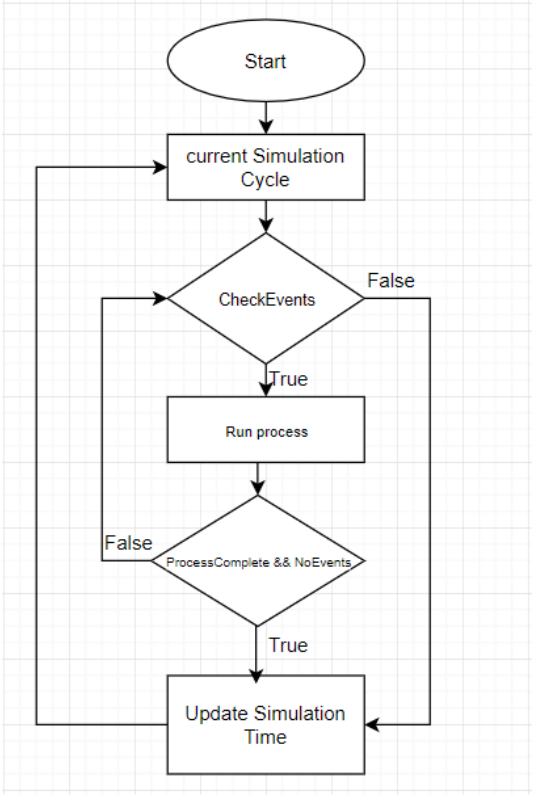

Fig. 17. Programmatic view of delta cycle and simulation time

The figure explains the composability of the two different cycles discussed above. This critical composition works indefinitely, so that every process can satisfy its requirements before it moves to the new simulation cycle.

The inner cycle works in an event driven fashion, where the delta cycles are executed upon different incoming events, however the outer loop works in discrete fashion.

D. Relating the underlying simulator with HLA

As taken from [13] [14] HLA is a general purpose architecture for distributed computer simulation systems. Individual distributed computer simulations can interact with other computer simulations regardless of their underlying

platforms. The whole interaction between the different simulations is controller by the run-time infrastructure (RTI). Here every single simulation entity is called a federate. So one whole simulation cycle of SystemC acts like an individual federate.

The problem that stems from having distributed simulations happening parallelly is that, the coordinated simulations may not correctly reproduce temporal aspects of the real world that is being modeled. This leads to complete asynchrony between the different messages being received from the simulation world. In order to avoid this asynchrony, time management between different federates is extremely critical. (Since, assuming this distributed simulators work for different war field application).

As discussed in our discussion of the simulation cycle and delta cycle, they are synchronized by the underlying scheduler which makes sure both remained in synchronization with each other. Simulation time will never be updated until every single delta cycle is complete. This looks a little simple when we consider it to the HLA, which has multiple simulators which work in parallel. The question that leads to this comparison is, how can all the federates be synchronized to provide a completely synchronized temporal relationships by eliminating anomalies from the simulation world.

Similar to our well built scheduler in SystemC, HLA relies on RTI for its time management, to remove asynchrony between its federates. RTI uses two important constraints, they are: a) Time stamps: b) Federates will not receive events in the past. This means, the federate will not have to care about arrival of events from the past time, once it has passed that time of execution. To elaborate, if the current simulation time is t=10, and a federate receives an event from the past which was t=7, in this case the RTI makes sure that the federate will not receive that event.

So, both of these tools look different at the top level, one works for the system level design and the other a generalized architecture for distributed simulation. But, there lies a similar underlying internal architecture of the scheduler. As discussed above, the SystemC simulator could be compared to a single federate in the HLA architecture.

VIII. Conclusion

This work assimilates different aspects of SystemC, to provide the readers a comprehensive understanding of the language. It explains the top level programming constructs and then makes a transition to the lower level simulator semantics. Reading this work, the user will be able to get a comprehensive look at different aspects of the language. The goal of this work is to explain the composability of the different actors involved in the system. Understanding the composability of different actors involved, would make it easier for the readers to use SystemC and build on its applications.

A critical review is provided on the different composability aspects of SystemC, to show the synchronization of simulation cycles with the internal delta cycles, co-ordination and composition of multiple processes to simulate its concurrent operation. Also HLA, a generalized architecture for distributed simulation is compared with the underlying SystemC scheduler to look for some similarity. So, internally a SystemC simulator could be thought of an individual federate of the HLA

References

[1] Adam Rose, Stuart Swan, John Pierce, Jean-Michel Fernandez, Transaction Level Modeling in SystemC, Cadence Design Systems, Inc., included in the TLM SystemC package, November 2004.

[2] P.R. Panda. SystemC: A modeling platform supporting multiple design abstractions. In Proceedings of the International Symposium on Systems Synthesis (ISSS), pages 7580. ACM, 2001.

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

[4] Y. Gurevich. Evolving algebra 1993: Lipari guide. In E. Borger, editor, Specification and Validation Methods. Oxford University Press, Oxford, 1994.

[5] Open SystemC Initiative, Synopsys Inc,-CoWare Inc. Frontier Inc. SYSTEM C Version 0.9 Users Guide, 1999.

[6] http://web.eecs.umich.edu/gasm/intro.html

[7] SystemC: From the Ground Up by David C. Black

[8] An Introduction to System level modeling in SystemC 2.0, January 2001.

[9] OSCI TLM-2.0 language reference manual, July 2009

[10] John Moondanos, SystemC Tutorial.

[11] http://www.asic-world.com/verilog/intro1.html

[12] John Moondanos, "SystemC Tutorial", UC Berkeley EE249 Lecture Notes.

[13] Janos sztipanovits, "High level Architecture: Basic concept", Vanderbilt University CS6377-01 Lecture Notes.

[14] Richard M. Fujimoto, "Time Management in the High Level Architecture", Georgia Institute of Technology.