Compositional concepts in SystemC
&
Exploring the expressive power of ASM

Seminar Part-2

Presented by:
Shreyas Ramakrishna
shreyas.ramakrishna@vanderbilt.edu

CS 6377-01 (2018S)
Topics in Embedded Software and System

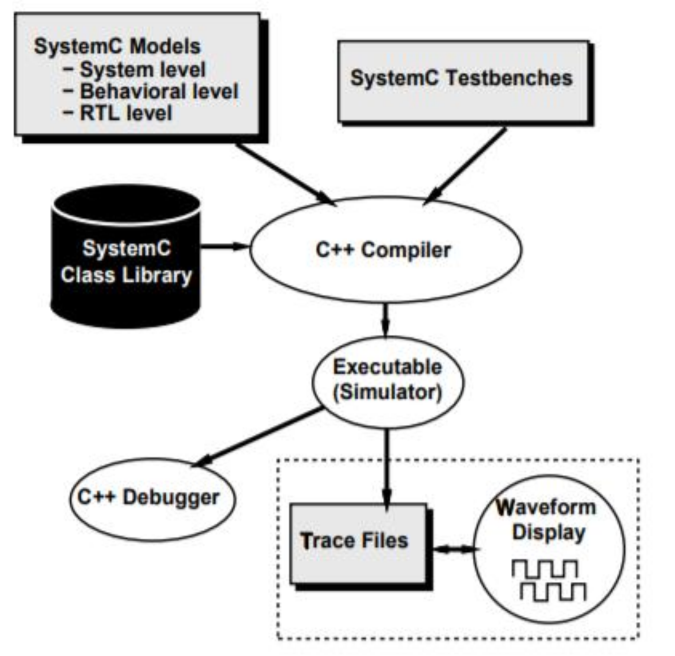
April 12th, 2018

VANDERBILT
UNIVERSITY

SYSTEMC™

Seminar Part-2 Agenda

- Review of the previous seminar
- Introduction to Abstract State Machine
- SystemC simulator semantics
- Simulation kernel
- Watching statements
- SystemC statements

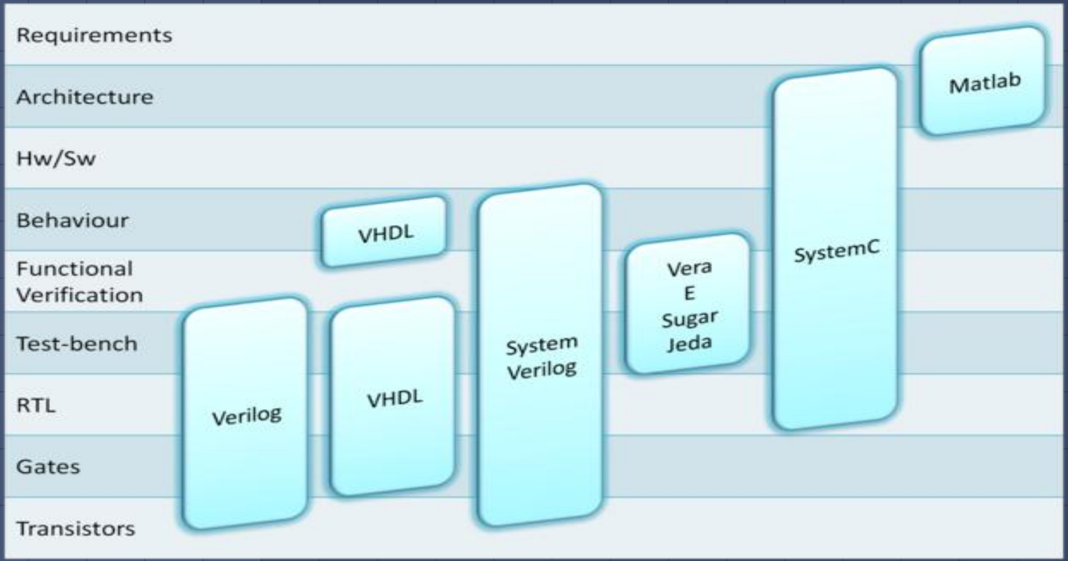


Review

Library of C++ classes which provides an event driven simulation interface.
Provides signals, events, and synchronization primitives to mimic the hardware description languages of VHDL and verilog.

What it offers:
Modules and Hierarchy
Hardware data types
Methods and threads
Events, Sensitivity
Interface and channels

What it is used for:
System-level modeling
Architectural exploration
Software development
Functional verification
High-level synthesis



Language Comparison

Review of the structure

Modules: Are the basic building blocks within SystemC to partition a design.

Ports: Used for intra-module communication.

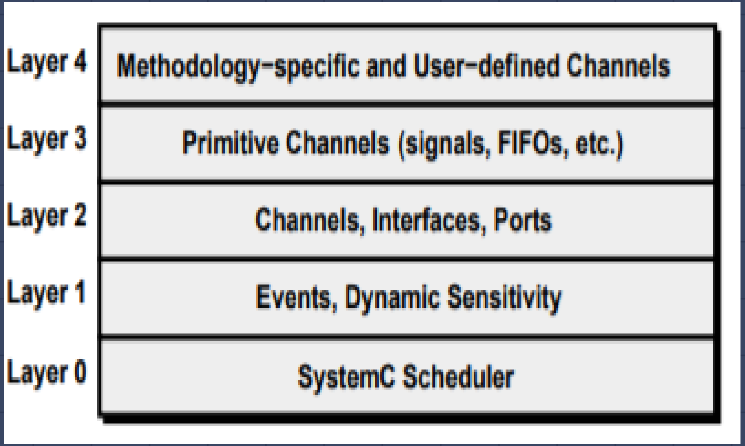
Signals: Used for inter-module communication.

Process: The functionality of a module is implemented here.

Channels: Provides communication between modules.

Time & clocks: SystemC adds the notion of time.

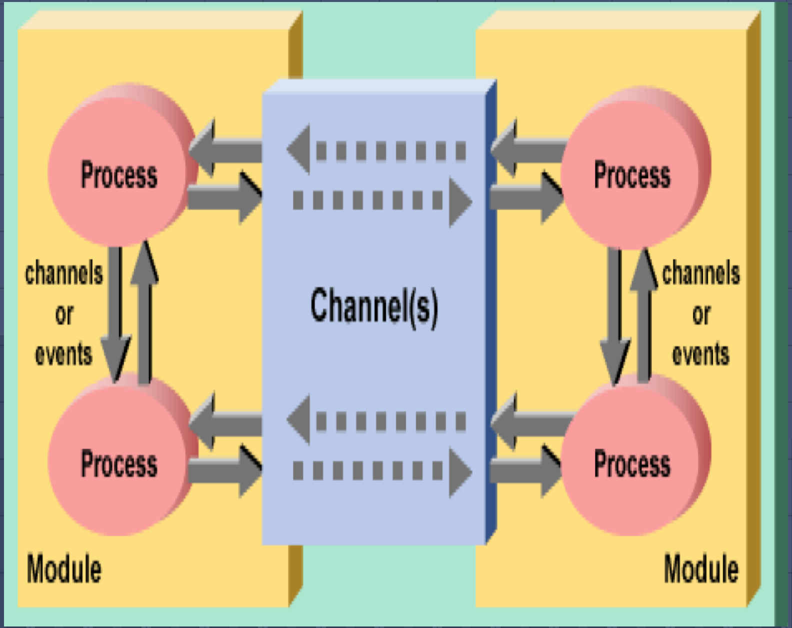
Hardware Data types.

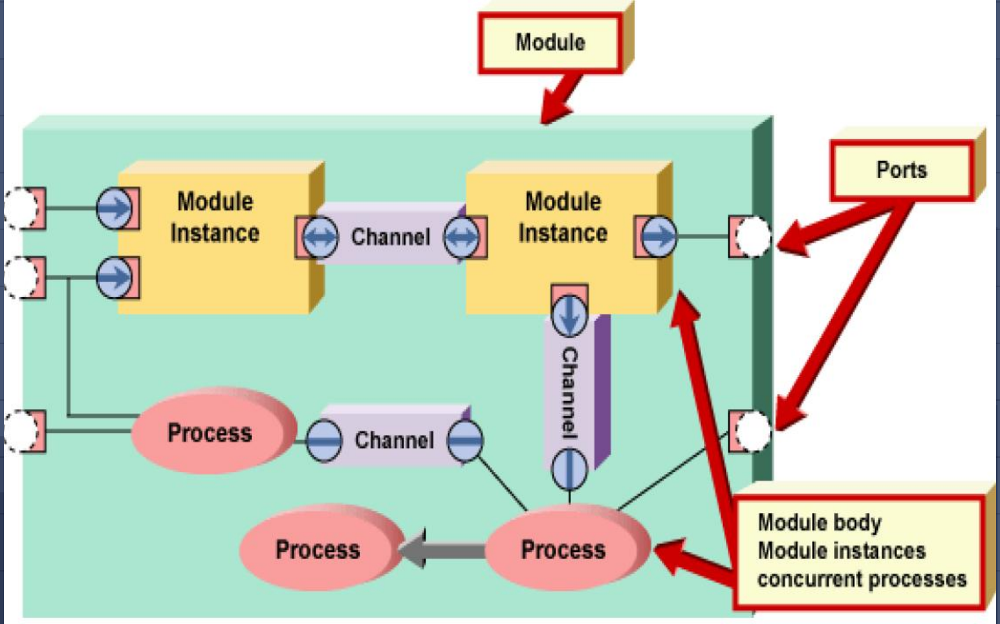


Layer 4	Methodology-specific and User-defined Channels
Layer 3	Primitive Channels (signals, FIFOs, etc.)
Layer 2	Channels, Interfaces, Ports
Layer 1	Events, Dynamic Sensitivity
Layer 0	SystemC Scheduler

SystemC Blocks

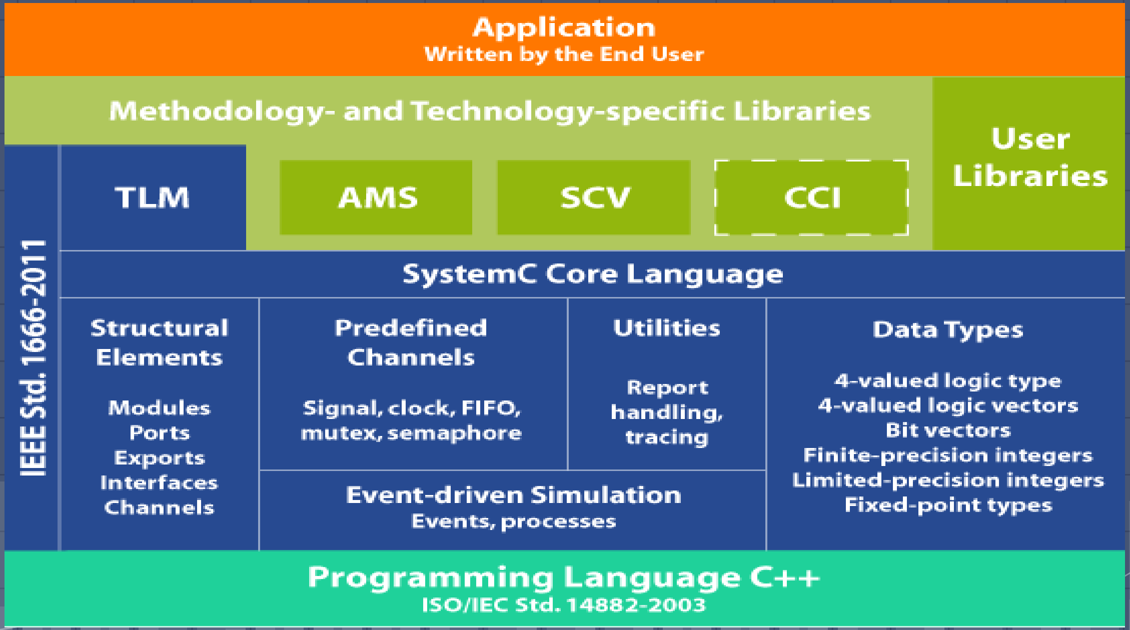
Top level overview of the SystemC blocks





Review of TLM

- TLM is an abstraction of communication among computation modules.
  - “Communication is separated from computation”



Application
Written by the End User

Methodology- and Technology-specific Libraries

User Libraries

TLM

AMS

SCV

CCI

IEEE Std. 1666-2011

SystemC Core Language

Structural Elements
Modules
Ports
Exports
Interfaces
Channels

Predefined Channels
Signal, clock, FIFO, mutex, semaphore

Utilities
Report handling, tracing

Data Types
4-valued logic type
4-valued logic vectors
Bit vectors
Finite-precision integers
Limited-precision integers
Fixed-point types

Event-driven Simulation
Events, processes

Programming Language C++
ISO/IEC Std. 14882-2003

CCI standardization effort is underway

What we concluded from the previous seminar

- The previous talk was intended to introduce SystemC, its features, and how it has been used in transaction level modeling. We saw in the last seminar, that SystemC works at a higher level of abstraction than its counterparts of HDL.
- We also saw that it emphasizes on communication rather than the implementation itself. The big advantage it provides is co-development and co-simulation of hardware and software, which in turn would increase the speed of production.
- This language (rather library of C++ classes) forms the basis for system level modeling and transaction level modeling.

Where do we take it from here

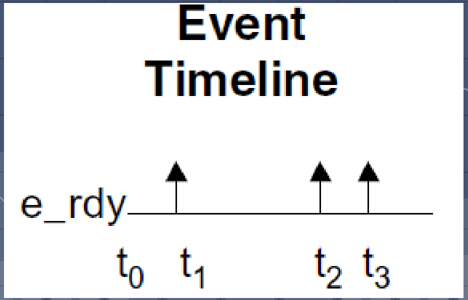
- In the last talk we went through the top level of SystemC, learnt the building blocks of the library, and some programming constructs. We also discussed about Transaction level modeling.
- In this talk, I would like to go some level below to talk about the underlying simulator.
- How does the internal things work in the simulator??
- Being a discrete event simulator, how does it handle concurrent processes.
- SystemC simulator semantics.

SystemC Building Units

- Three basic SystemC Macros:
- SC_Method, SC_Thread and SC_Cthreads.
- SC_Cthread: A special case of SC_Thread, which is sensitive to a clock edge.
- These three macros are the basic units of concurrent execution.
- Simulation kernel is responsible to invoke each of these three units, and is never done directly by the user.
- The user indirectly controls execution of the simulation processes by the kernel as a result of events, sensitivity, and notification.

Events

- Events are key for an event-driven simulator like SystemC.
- An event is something that happens at a specific point in time.
- An event has no value and no duration.
- Processes wait for an event by using a dynamic or static sensitivity.
- If an event occurs, and no processes are waiting to catch it, the event goes unnoticed.

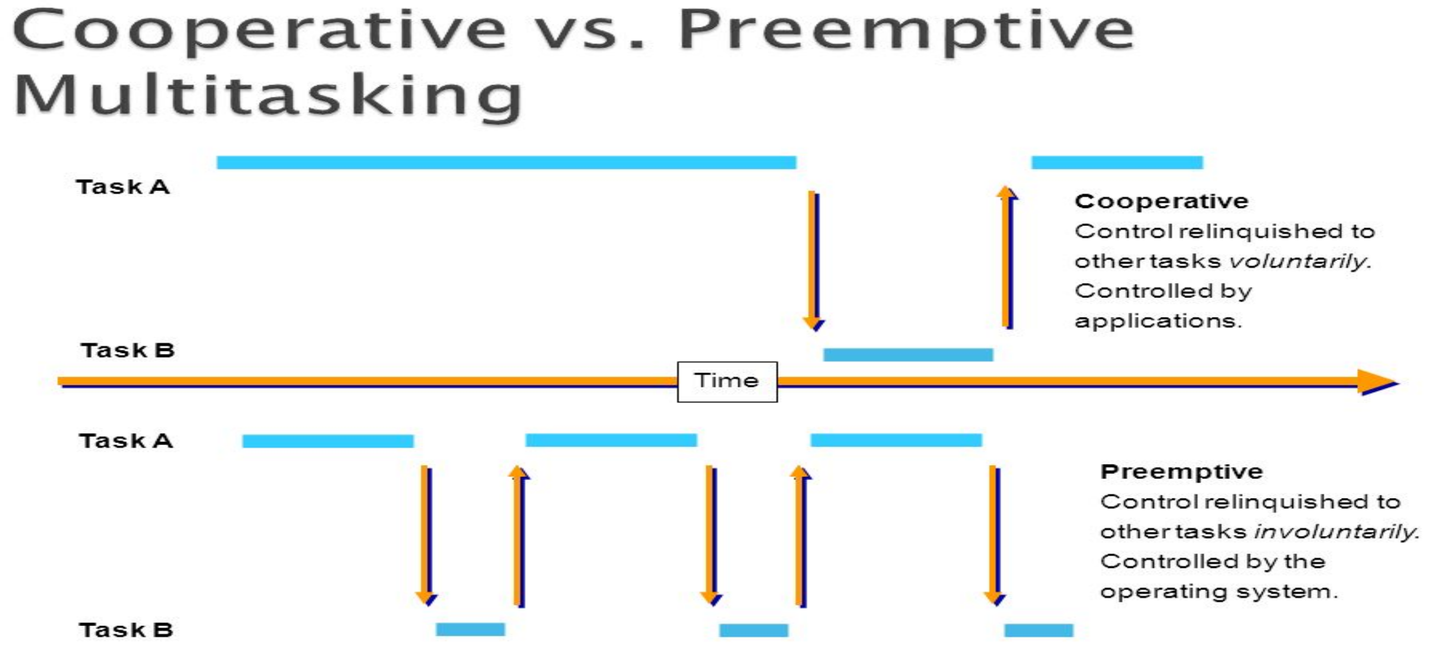


Sensitivity

- Events are implemented by the SystemC sc_event class, they are fired through the sc_event member function, notify.
- When processes are sensitive to an event, the simulation kernel schedules the process to be invoked.
- **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.
- SC_METHOD implements dynamic sensitivity with a next_trigger(arg).
- SC_THREAD implements dynamic sensitivity with a next(arg).

Co-operative v.s pre-emptive multitasking

- SystemC kernel handles the user processes in a non pre-emptive/ co-operative fashion.

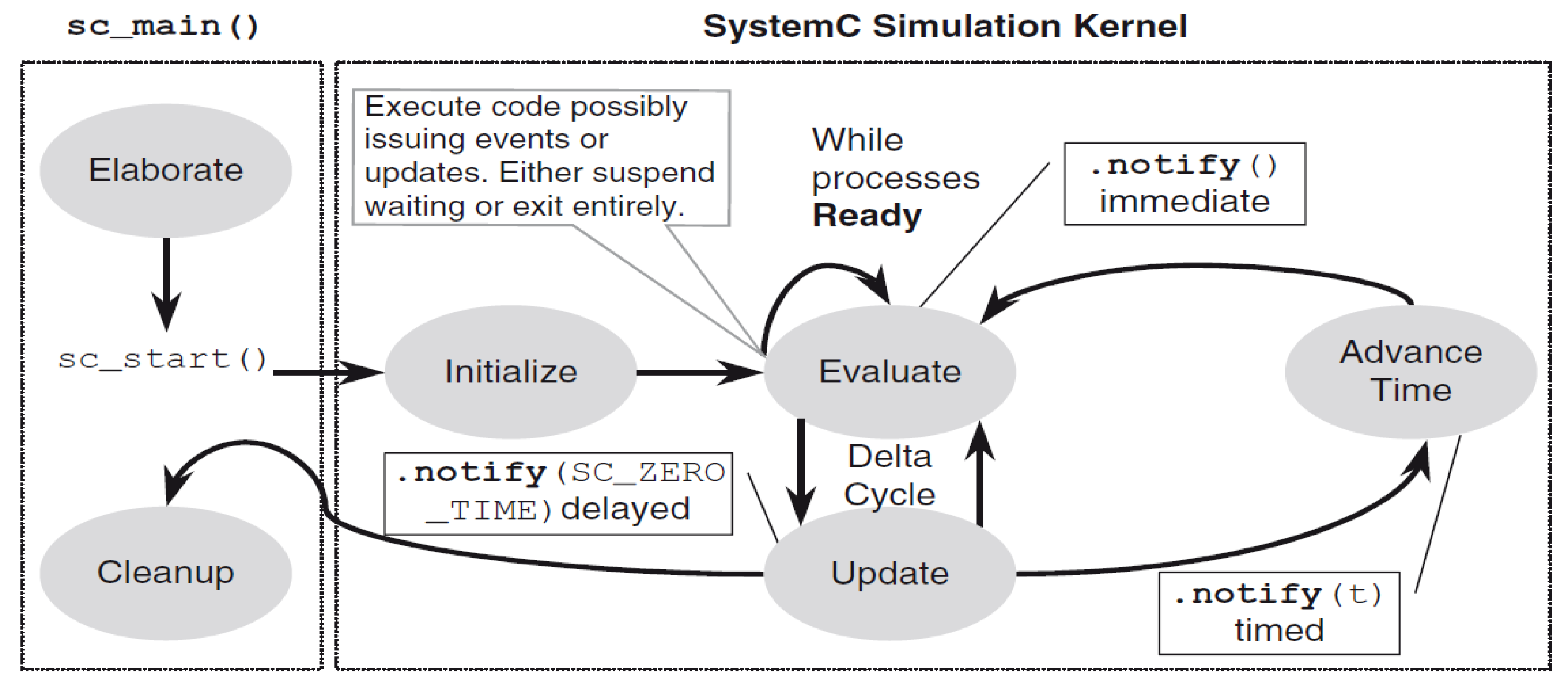


Cooperative vs. Preemptive Multitasking

Cooperative
Control relinquished to other tasks *voluntarily*. Controlled by applications.

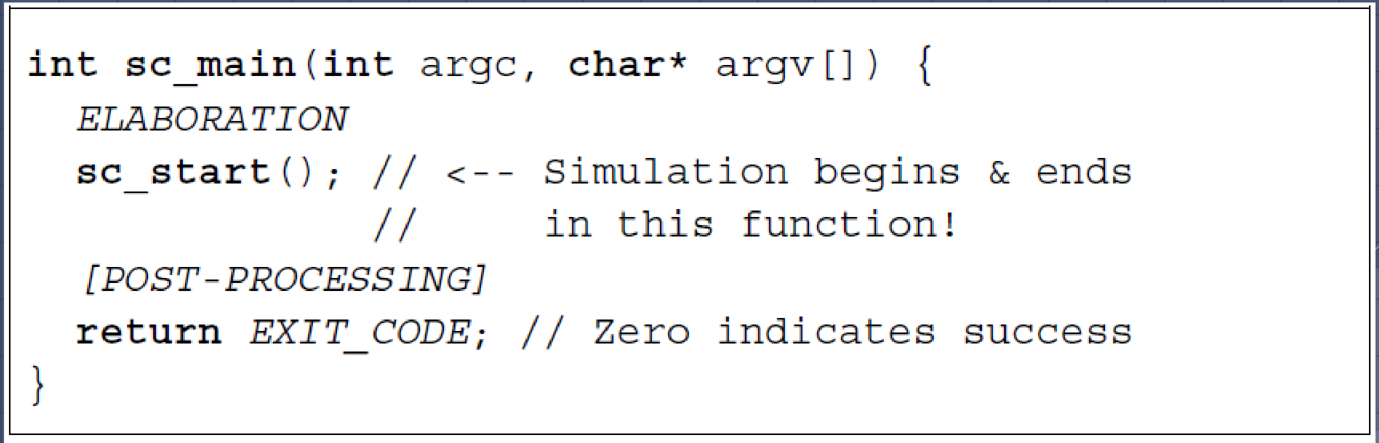
Preemptive
Control relinquished to other tasks *involuntarily*. Controlled by the operating system.

Phases of simulation kernel


sc_main()
SystemC Simulation Kernel
Elaborate
sc_start()
Cleanup
Execute code possibly issuing events or updates. Either suspend waiting or exit entirely.
While processes **Ready**
.notify() immediate
Initialize
Evaluate
Advance Time
Delta Cycle
.notify(SC_ZERO _TIME) delayed
Update
.notify(t) timed

Phases of simulation kernel

- SystemC simulator has two major phases of operation: elaboration and execution.
- **Elaboration phase**: Execution of statements prior to the sc_start() function.
- **Execution phase:** hands control to the SystemC simulation kernel, to orchestrate the execution of processes. (for concurrency)



```
int sc_main(int argc, char* argv[]) {
  ELABORATION
  sc_start(); // <-- Simulation begins & ends
              //     in this function!
  [POST-PROCESSING]
  return EXIT_CODE; // Zero indicates success
}
```

Elaboration phase

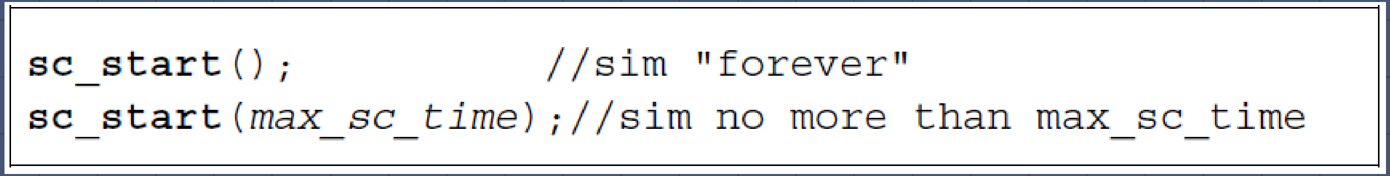
- **Elaboration phase:** Structures needed to describe the interconnections of the system are connected.
- Elaboration establishes hierarchy and initializes the data structures.
- Creates instances of clocks, design modules, and channels that interconnect designs.
- Additionally, invokes code to register processes and perform the connections between design modules.
- At the end of elaboration, sc_start() invokes the simulation phase.

Execution & Post-processing phase

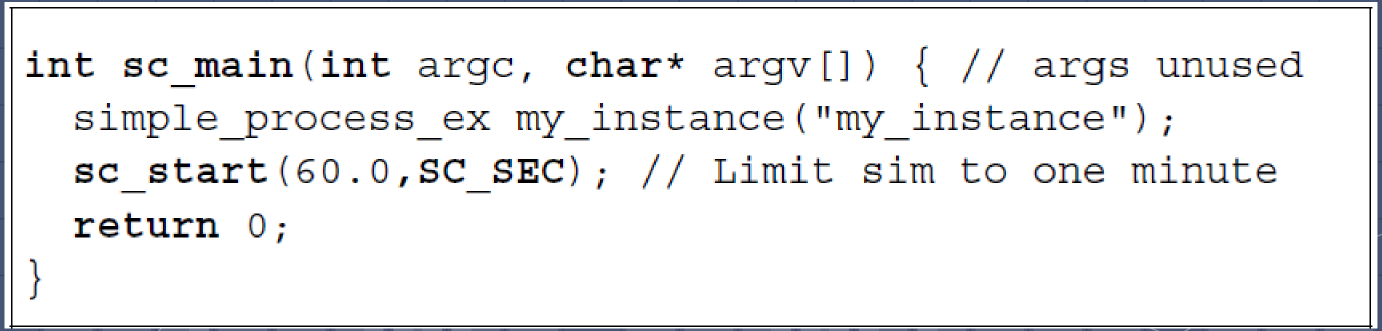
- **Execution phase:** code representing the behavior of the model executes.
- Finally, after returning from sc_start(), the post-processing phase begins.
- This may read data created during simulation and format reports or otherwise handle the results of simulation.

Simulation start

- sc_start() starts the simulation phase, taking an optional argument of sc_time().



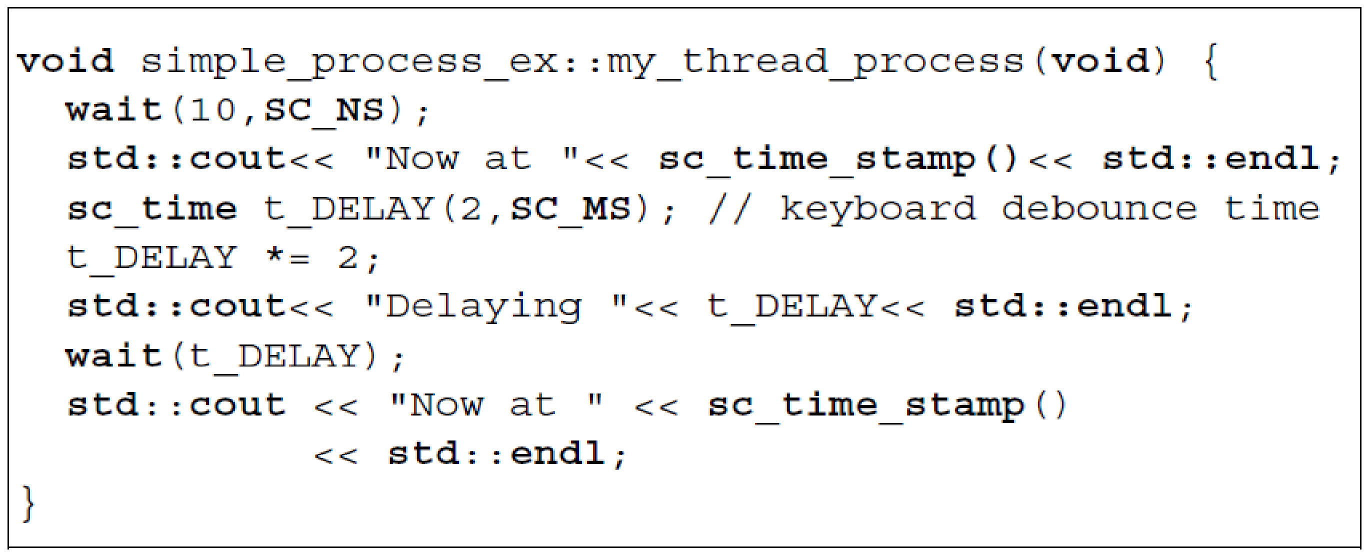
```
sc_start();          //sim "forever"
sc_start(max_sc_time);//sim no more than max_sc_time
```



```
int sc_main(int argc, char* argv[]) { // args unused
  simple_process_ex my_instance("my_instance");
  sc_start(60.0,SC_SEC); // Limit sim to one minute
  return 0;
}
```

Wait

- Provides syntax allowing delay specifications in SC_THREAD processes.
- When a wait() is invoked, the SC_THREAD process blocks itself and is resumed by the scheduler at the specified time.



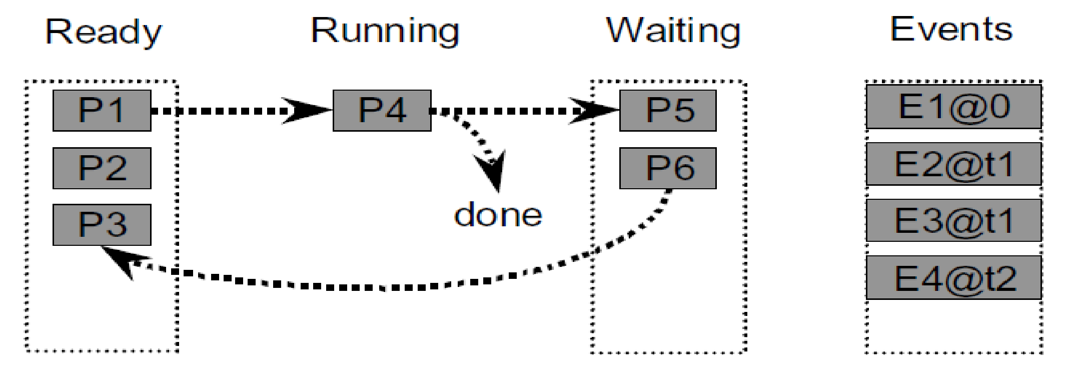
```
void simple_process_ex::my_thread_process(void) {
  wait(10,SC_NS);
  std::cout<< "Now at "<< sc_time_stamp()<< std::endl;
  sc_time t_DELAY(2,SC_MS); // keyboard debounce time
  t_DELAY *= 2;
  std::cout<< "Delaying "<< t_DELAY<< std::endl;
  wait(t_DELAY);
  std::cout << "Now at " << sc_time_stamp()
            << std::endl;
}
```

Wait

- What happens when wait is called??
- When wait executes, the state of the current thread is saved, the simulation kernel is put in control and proceeds to activate another ready process.
- When the suspended process is reactivated, the scheduler restores the calling context of the original thread, and the process resumes execution at the statement after the wait

Process & Event Pools

- Once simulation enters into evaluation phase,
- Processes are randomly taken one by one from the wait queue to running.
- Each process executes till it either completes execution or is suspended by wait.
- Completed processes are discarded.
- Suspended processes are placed into a waiting pool.
- Simulation proceeds until there are no more processes ready to run.



After Execution

- Execution exits the evaluate bubble with one of three possibilities:
- waiting processes or events that are zero time delayed, non-zero time delayed, or neither.
- 1) There may be processes or events waiting for an SC_ZERO_TIME delay.
- Waiting pool processes with zero time delays are placed back into the ready pool.
- Another round of evaluation occurs if any processes have been moved into the ready pool.

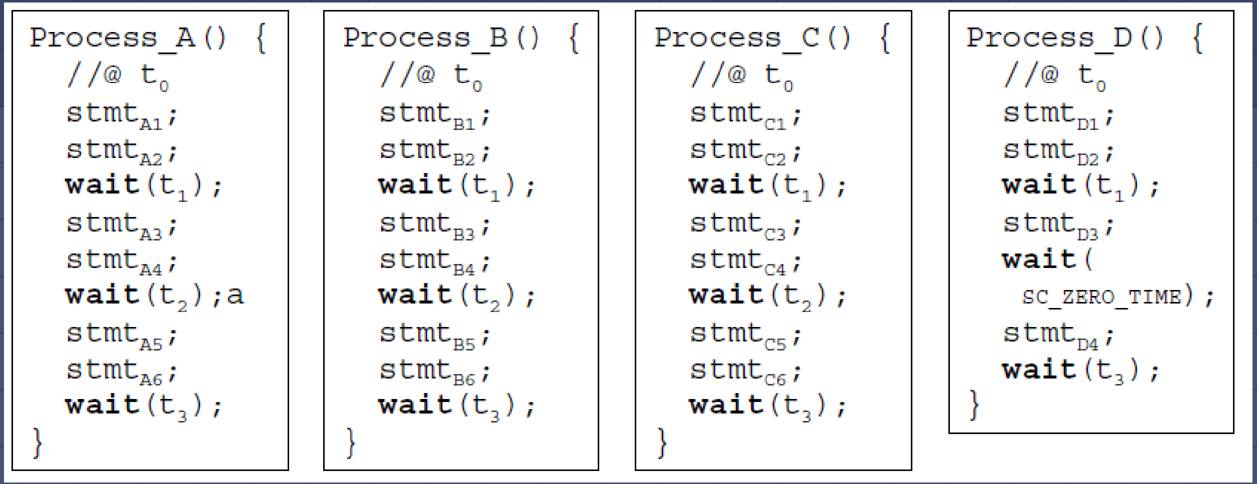
After Execution

- 2) There may be processes or events, scheduled for later, waiting for a non-zero time delay to occur.
- Processes waiting on that specific delay will be placed into the ready pool.
- If an event occurs at this new time, processes waiting on that event are placed into the ready pool.
- Another round of evaluation occurs if any processes have been moved into the ready pool.

- 3) There were no delayed processes or events.
- Since there are no processes in the ready pool, then the simulation simply ends.

Concurrency

- Lets see an example.

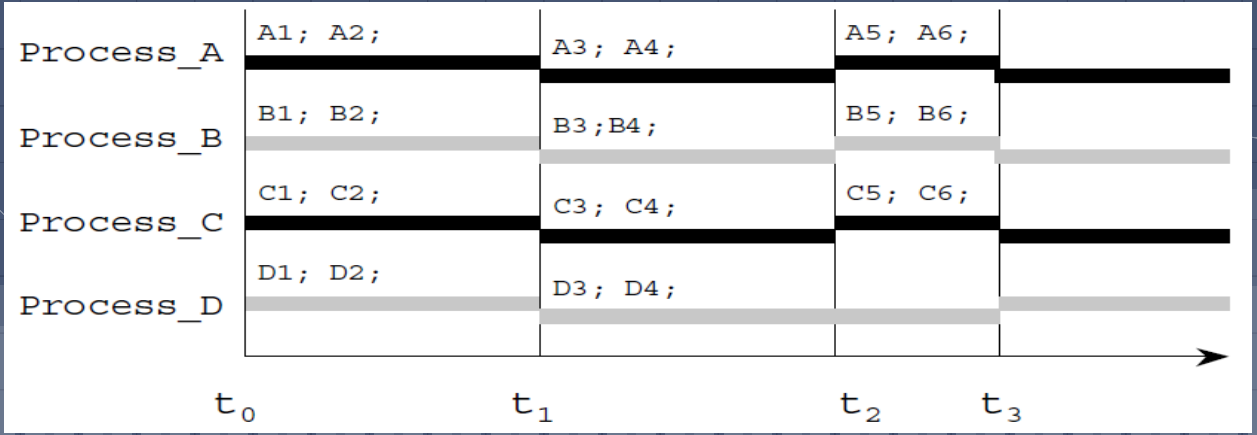


```
Process_A() {
  //@ t_0
  stmt_A1;
  stmt_A2;
  wait(t_1);
  stmt_A3;
  stmt_A4;
  wait(t_2);a
  stmt_A5;
  stmt_A6;
  wait(t_3);
}
```

```
Process_B() {
  //@ t_0
  stmt_B1;
  stmt_B2;
  wait(t_1);
  stmt_B3;
  stmt_B4;
  wait(t_2);
  stmt_B5;
  stmt_B6;
  wait(t_3);
}
```

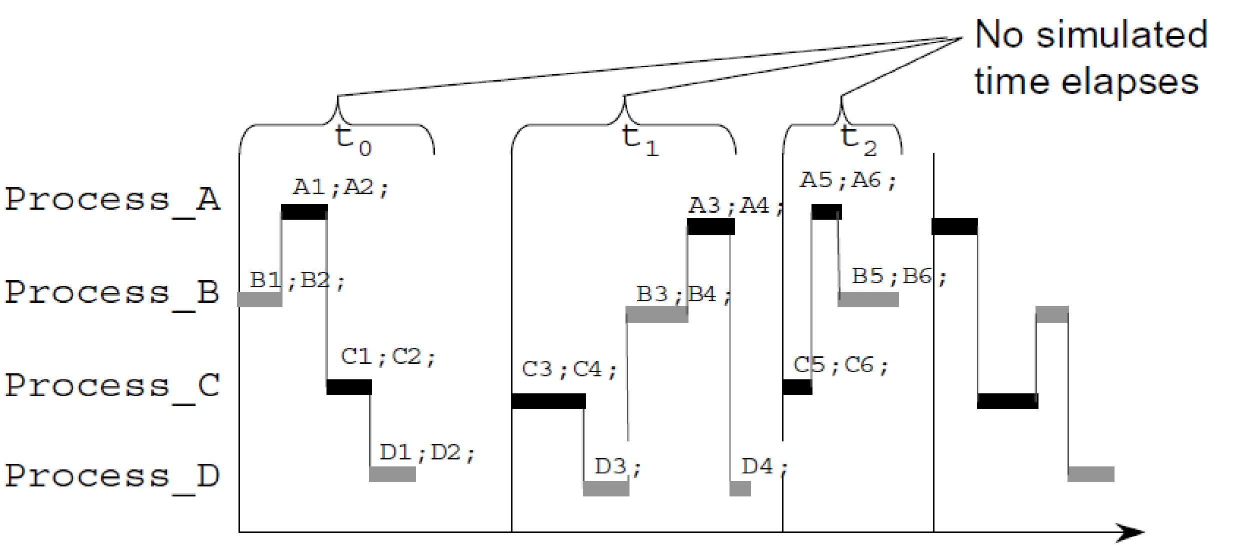
```
Process_C() {
  //@ t_0
  stmt_C1;
  stmt_C2;
  wait(t_1);
  stmt_C3;
  stmt_C4;
  wait(t_2);
  stmt_C5;
  stmt_C6;
  wait(t_3);
}
```

```
Process_D() {
  //@ t_0
  stmt_D1;
  stmt_D2;
  wait(t_1);
  stmt_D3;
  wait(
   SC_ZERO_TIME);
  stmt_D4;
  wait(t_3);
}
```

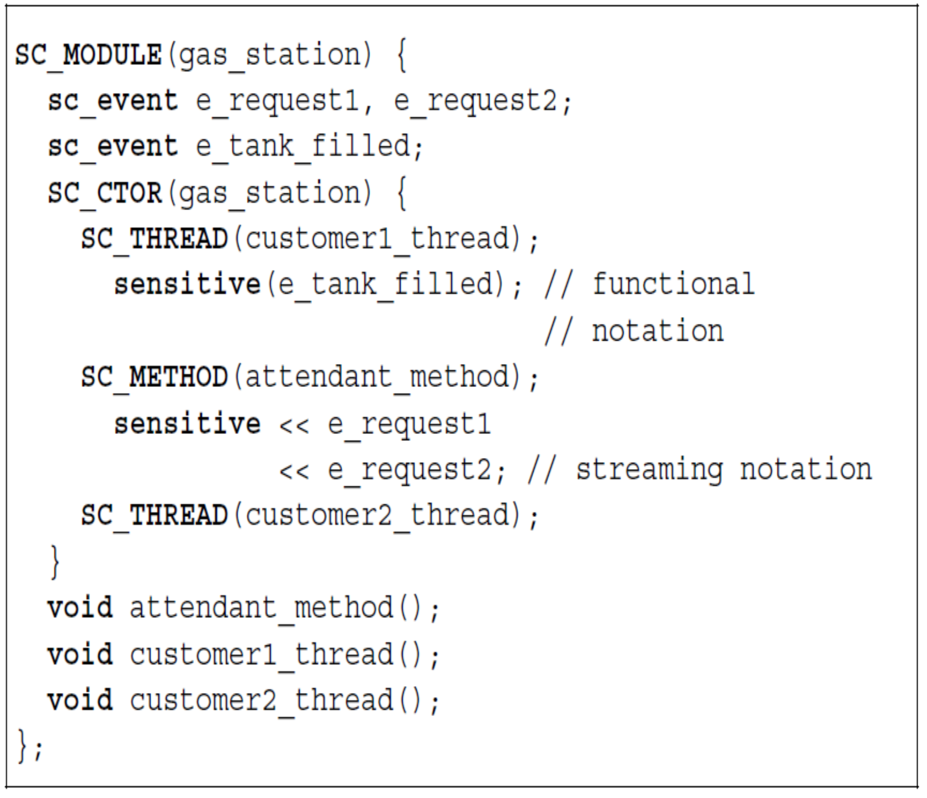


Non-deterministic SystemC

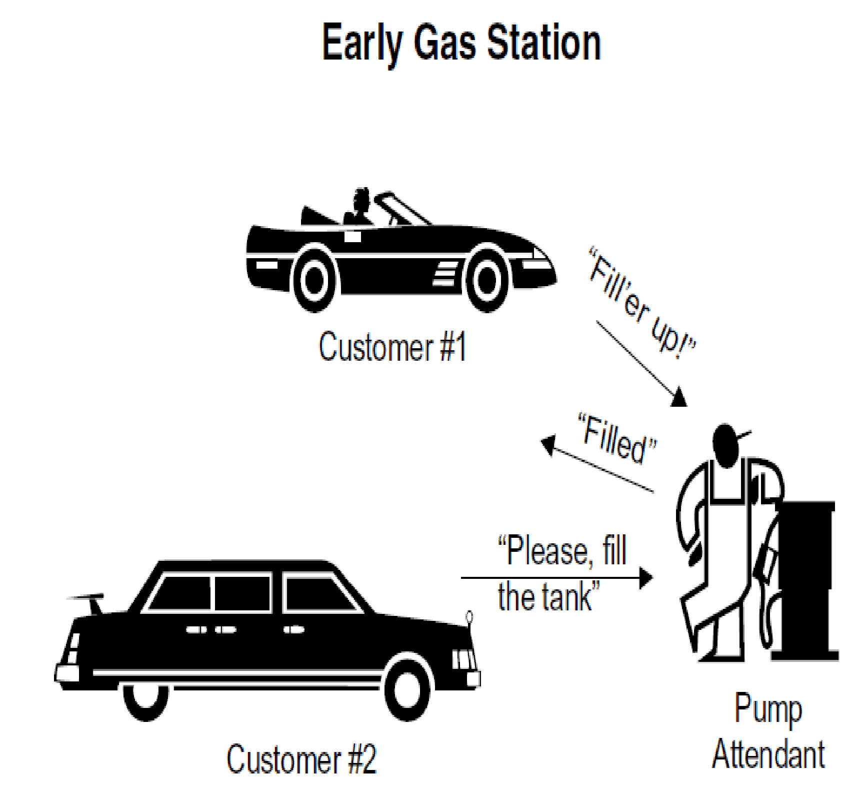
- All of the statements are execute during the same evaluate phase of a delta cycle.
- If any of the statements had been a delayed notification, then multiple delta cycles may have occurred during the same instant in time.



Example



```
SC_MODULE(gas_station) {
  sc_event e_request1, e_request2;
  sc_event e_tank_filled;
  SC_CTOR(gas_station) {
    SC_THREAD(customer1_thread);
      sensitive(e_tank_filled); // functional
                                // notation
    SC_METHOD(attendant_method);
      sensitive << e_request1
                << e_request2; // streaming notation
    SC_THREAD(customer2_thread);
  }
  void attendant_method();
  void customer1_thread();
  void customer2_thread();
};
```



Abstract State Machines

&

The Simulation semantics of SystemC

What to expect from this work

- Provides a semantics definition of SystemC that covers method, thread, clocked thread behavior and their interaction with the simulation kernel process.
- It also includes watching statements, signal assignment, and wait statements.
- Define the semantics in terms of Abstract state machines (ASM’s).
- Develops a computational model of interaction between the user defined processes and the simulation kernel process.
- Basically understand how the **whole simulation kernel is designed to work.**

Abstract State Machines (ASM’s)

- Conventional computation models assume symbolic representations of states and actions.
- Abstract-State Machine model takes a more liberal position:
- Any mathematical structure may serve as a state.
- This results in a computational model that is more powerful and more universal than standard computation models.

“The Expressive power of Abstract State Machines”

ASM (interesting read)

- ASM idea: Any algorithm can be modeled at its natural abstraction level by an appropriate ASM.
- Developed a methodology based upon 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.
- Plentiful examples exist in the literature of ASMs applied to different types of algorithms.
  - “The Expressive power of Abstract State Machines”

ASM characteristics

- Precision
- Faithfulness
- Understandability
- Executability
- Scalability
- Generality

Interesting Reads:

Evolving Algebras 1993: Lipari Guide∗, by Yuri Gurevich

http://web.eecs.umich.edu/gasm/intro.html

How it works

- A state transition is performed by firing a set of rules in one step.
- Only those rules are fired whose guards (Condition) evaluate to true.
- It just works like a simple if, else statement. Once the condition is satisfied, the statements in if are executed, or else control transfers to else.

if Condition then <Updates > else < Updates > endif

- At each step the guards evaluate to a set of function updates (block) each of form f (tl, t2 ... ti) := to where ti are terms (including functions).

Distributed ASM’s

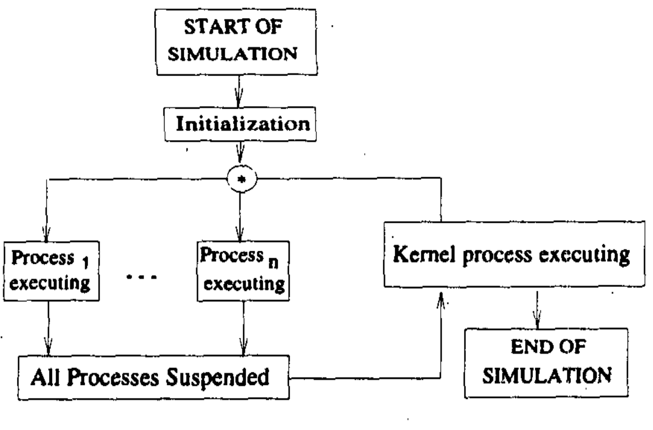
- The distributed ASMs, partitions rules into modules where each module is given by its module name v.
- A module is instantiated to execute by setting Mod(a) := v for an agent.
- The execution is defined by partially ordered state transitions where agents are asynchronously executed.
- The SystemC specification has two modules: the kernel process and the user processes.

Formal Behavioral Semantics

- SystemC establishes 3 hierarchical network of a finite number of parallel communicating processes each of methods, threads and CThreads.
- These, under the supervision of the distinguished simulation kernel process, concurrently update's the new values for given signals and variables.
- The signals do not change their values immediately. Their assignments become effective only in the next simulation cycle.(which will be shown in the further slides)

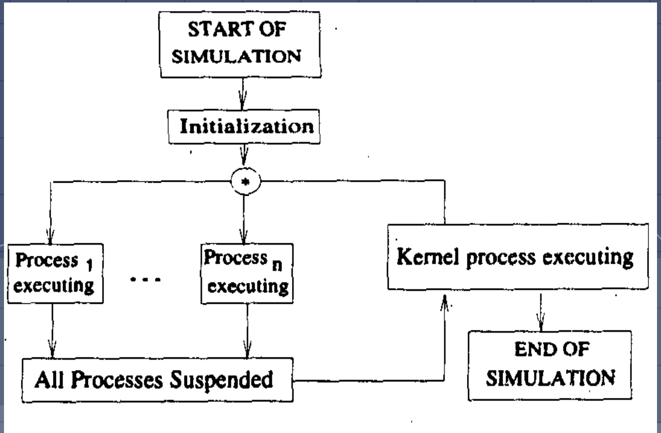
Basic Concepts of the model

- When all user defined processes are suspended, the kernel process goes through different phases and updates signals and clocks, invokes processes, and advances simulation time.
- Advancements of clocks and assignments to signals which are performed by user defined processes cause events which may trigger processes again to execute.
- SystemC processes can be classified into methods, threads and clocked threads.



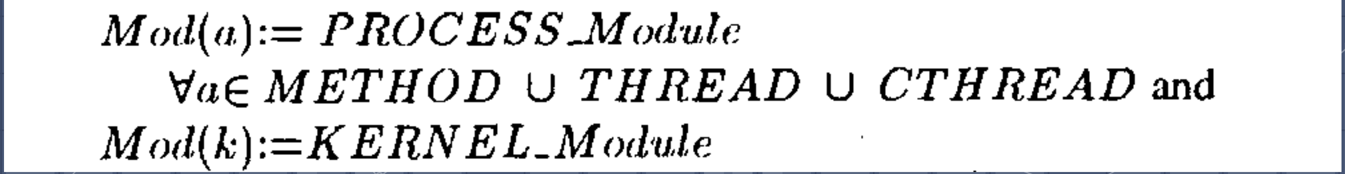
Basic Concepts of the model

- Clocked threads are executed only on request of time advancement, i.e.. after all the methods and threads have finished execution at Tc.
- So after reaching Tc, a time advancements take place.
- Then before resuming processes and executing them, the new signal values have to be assigned to current values which may lead to the generation of new events.



ASM and the model

- Modeling ASM agents, one for the simulation kernel process and one for each of the user defined process.
- We had spoken earlier of ASM agents, agents are instantiations of ASM modules. two initial steps:
- 1) First rules are defined for the Kernel module
- 2) Then define the semantics of the different statements that are executed in the process module.
- For initialization:



$$Mod(a) := PROCESS\_Module$$

$$\forall a \in METHOD \cup THREAD \cup CTHREAD \text{ and}$$

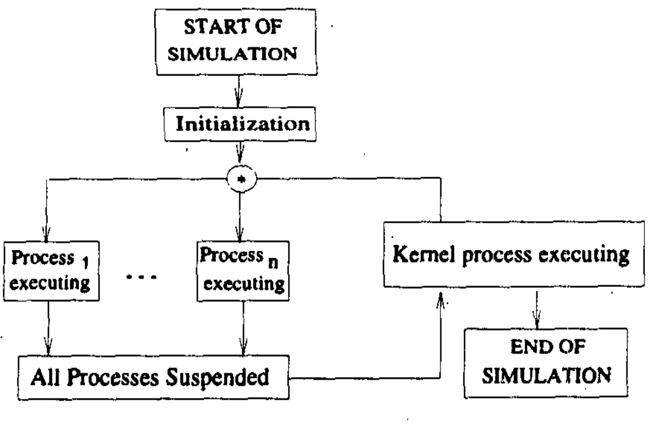
$$Mod(k) := KERNEL\_Module$$

ASM and Initialization

- Modules of threads and clocked threads are set undef after executing the last statement which disables these processes until the end of simulation (EOS).
- At the beginning, we consider the phase = executeProcesses and current time Tc=0.0
- Given the underlying discrete SystemC time model, the domain Time is linearly ordered.
- Uses Tc for current simulation time.
- Uses Tn for the next simulation time.

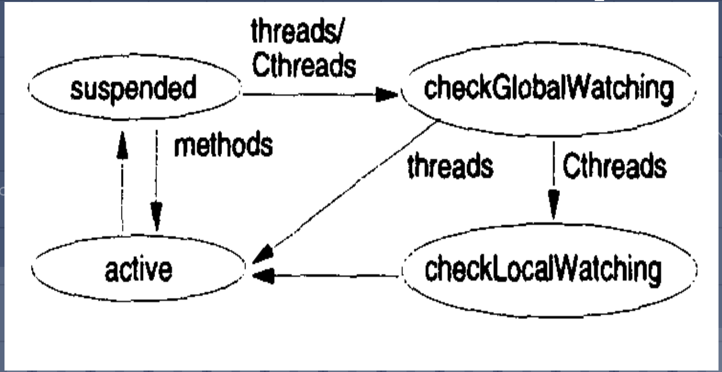
Formal Behavioral Semantics

- Simulation starts with the sc_start, and initial values are assigned to the signals.
- After the initial generation of events, there is a mutually exclusive execution of the simulation kernel process and the concurrently running (user defined) processes.
- The kernel process periodically starts its execution if all user defined processes are suspended and vice versa.
- The user defined processes end, when it thinks that there are no more updates to listen for.



States and context switching

- Each user defined process is active until it suspends upon reaching a wait statement or after executing the last process statement.
- Before getting active again, a process first checks its watching conditions and sets its program counter accordingly.
- Considering the life cycle of a process p, set's the status(p) to either the active, suspended, checkglobal watching, checklocal watching.
- After invocation, a method moves from status suspended to active.

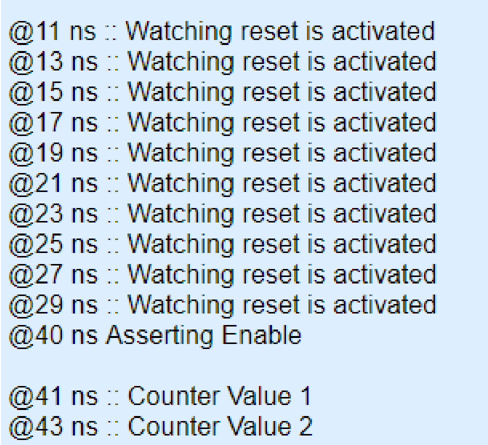


Global Watching

- SC_CThread processes typically have infinite loops that will continuously execute. A designer typically wants some way to initialize the behavior of the loop or jump out of the loop when a condition occurs.
- This is accomplished through the use of the watching construct. The watching construct will monitor a specified condition.
- When this condition occurs control is transferred from the current execution point to the beginning of the process, where the occurrence of the watched condition can be handled. The watching construct is only available for SC_CThread processes.

Global Watching

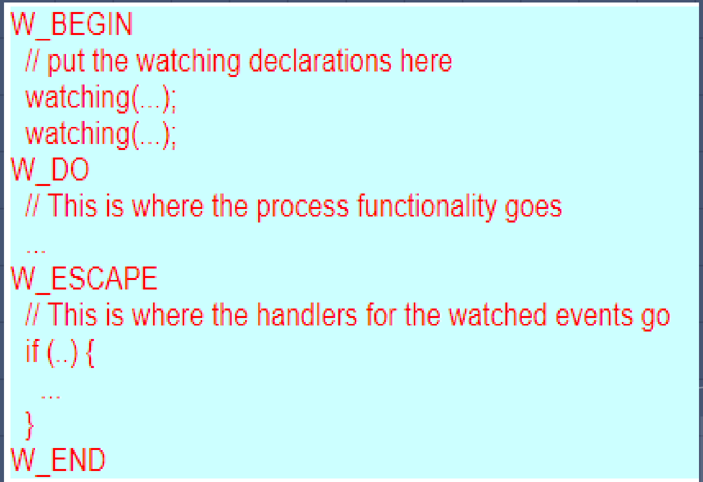
- `watching(reset.delayed() == true);`
- This statement specifies that signal reset will be watched for this process.
- If signal reset changes to true then the watching expression will be true and the SystemC scheduler will halt execution of the while loop for this process and start the execution at the first line of the process.
-



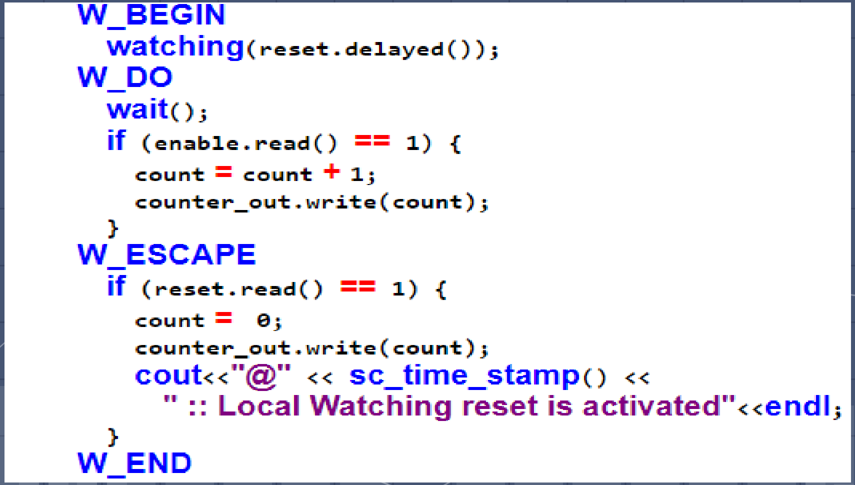
```
@11 ns :: Watching reset is activated
@13 ns :: Watching reset is activated
@15 ns :: Watching reset is activated
@17 ns :: Watching reset is activated
@19 ns :: Watching reset is activated
@21 ns :: Watching reset is activated
@23 ns :: Watching reset is activated
@25 ns :: Watching reset is activated
@27 ns :: Watching reset is activated
@29 ns :: Watching reset is activated
@40 ns Asserting Enable

@41 ns :: Counter Value 1
@43 ns :: Counter Value 2
```

Local Watching

- Local watching allows you to specify exactly which section of the process is watching which signals, and where the event handlers are located. This functionality is specified with 4 macros that define the boundaries of each of the areas.
-



```
W_BEGIN
  // put the watching declarations here
  watching(...);
  watching(...);
W_DO
  // This is where the process functionality goes
  ...
W_ESCAPE
  // This is where the handlers for the watched events go
  if (..) {
    ...
  }
W_END
```



```
W_BEGIN
  watching(reset.delayed());
W_DO
  wait();
  if (enable.read() == 1) {
    count = count + 1;
    counter_out.write(count);
  }
W_ESCAPE
  if (reset.read() == 1) {
    count =  0;
    counter_out.write(count);
    cout<<"@" << sc_time_stamp() <<
      " :: Local Watching reset is activated"<<endl;
  }
W_END
```

Local Watching

- The W_BEGIN macro marks the beginning of the local watching block.
- Between the W_BEGIN and W_DO macros are where all of the watching declarations are placed.
- These declarations look the same as the global watching events. Between the W_DO macro and the W_ESCAPE macro is where the process functionality is placed.
- This is the code that gets executed as long as none of the watching events occur. Between the W_ESCAPE and the W_END macros is where the event handlers reside.
- The event handlers will check to make sure that the relevant event has occurred and then perform the necessary action for that event.
- The W_END macro ends the local watching block.

Local and Global Watching

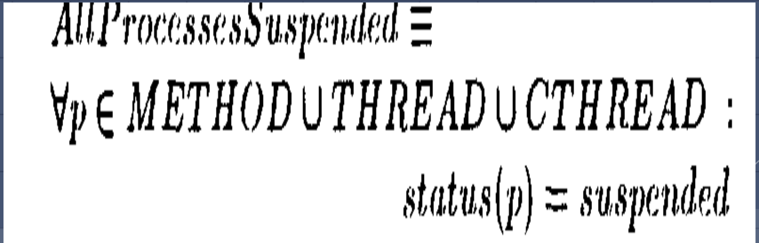
- A few interesting things about local watching:
- All of the events in the declaration block have the same priority. If a different priority is needed then local watching blocks will need to be nested.
- Local watching only works in CThreads.
- The signals in the watching expressions are sampled only on the active edges of the process.
- Globally watched events have higher priority than locally watched events.

Overview of the simulation

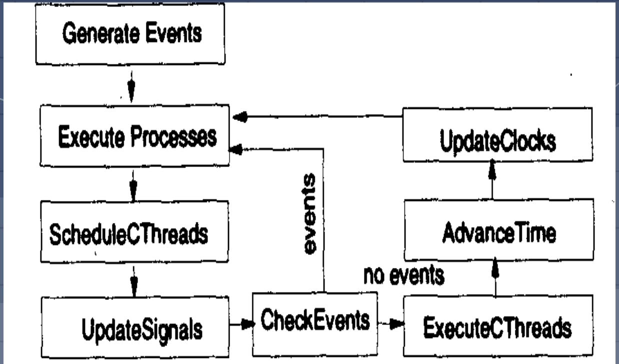
- Simulation of concurrent execution is accomplished by simulating each concurrent unit (SC_METHOD, SC_THREAD, or SC_CTHREAD).
- Each unit is allowed to execute until simulation of the other units is required to keep behaviors aligned in time.
- Events in the simulation code determines when the simulator has to make switches.
- The simulator uses a cooperative multi-tasking model.
- It provides a kernel to orchestrate the swapping of the various concurrent processes.

Simulation kernel

- Lets explore the each phase of the simulation kernel.
- The kernel is a separate process which is executed as soon as all user defined processes are suspended.
- When all processes are suspended, the kernel goes through different states by setting the function phase where Generate Events function generates initial events for all clocks which are active at Tc.

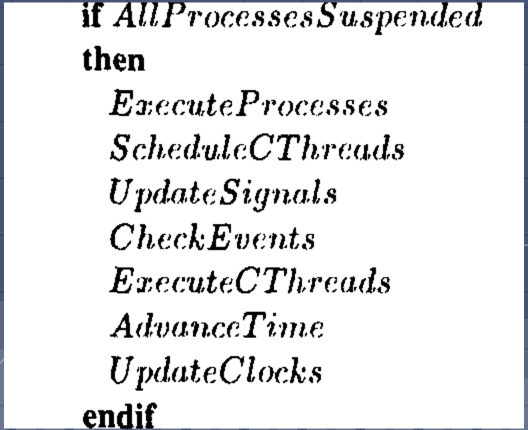


$$AllProcessesSuspended \equiv \forall p \in METHOD \cup THREAD \cup CTHREAD : status(p) = suspended$$

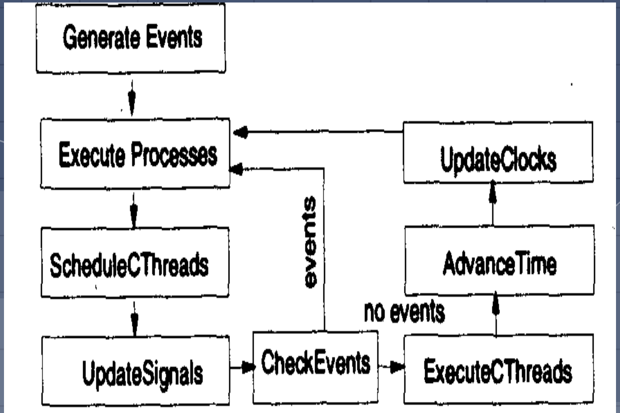


Simulation kernel

- Thereafter, the methods and threads with events are executed until they suspend.
- Threads are suspended after executing a wait statement and methods are suspended after their last statement.
- Then the simulation time is advanced, and clocks are updated w.r.t. the time of the next active clock.

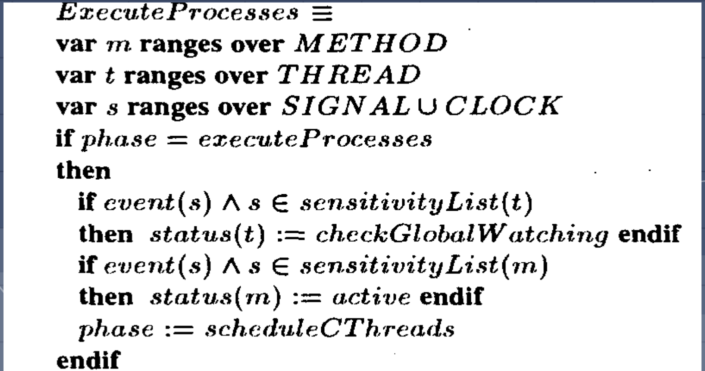


```
if AllProcessesSuspended
then
   ExecuteProcesses
   ScheduleCThreads
   UpdateSignals
   CheckEvents
   ExecuteCThreads
   AdvanceTime
   UpdateClocks
endif
```

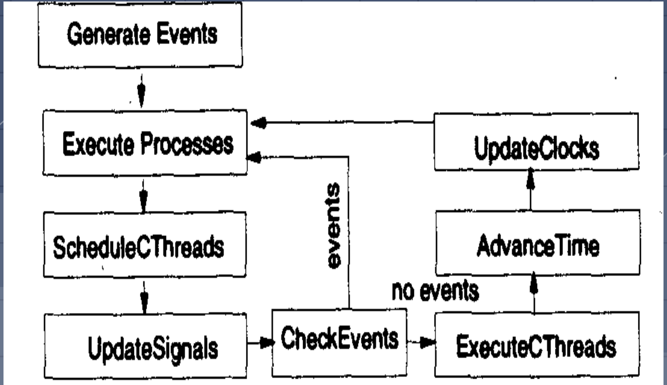


Phases of the kernel operation

- The Execute Process function checks for events of all the signals and clocks in the sensitivity lists of all method's and thread's.
- In the case of an event each thread is set to status checkGlobalWatching after which it will be activated.
- Methods are immediately set to active since no watching are required.
- The phase is then finally incremented.

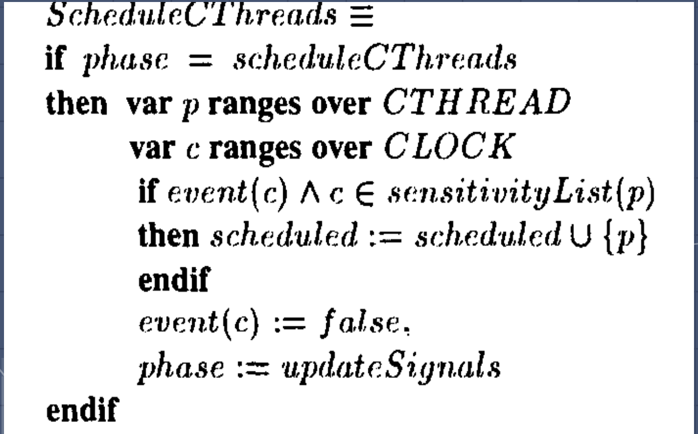


```
$ExecuteProcesses \equiv$
var $m$ ranges over $METHOD$
var $t$ ranges over $THREAD$
var $s$ ranges over $SIGNAL \cup CLOCK$
if $phase = executeProcesses$
then
  if $event(s) \wedge s \in sensitivityList(t)$
  then $status(t) := checkGlobalWatching$ endif
  if $event(s) \wedge s \in sensitivityList(m)$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

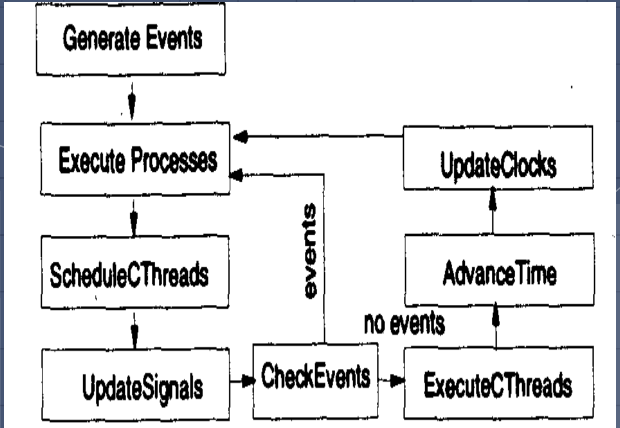


Phases of the kernel operation

- The Schedule Cthreads fuction checks for events on the clocks over all the sensitivity lists of all the cthread's.
- In the case of an event, the corresponding cthread is added to the set of scheduled cthread's, the clock events are reset, and the next phase will be assigned.

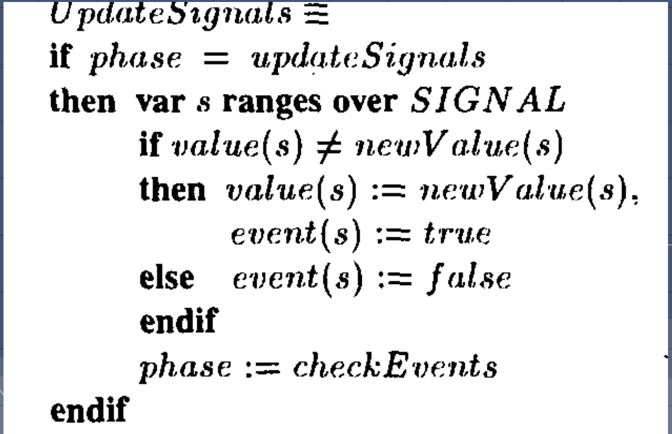


```
$ScheduleCThreads \equiv$
if $phase = scheduleCThreads$
then var $p$ ranges over $CTHREAD$
      var $c$ ranges over $CLOCK$
       if $event(c) \wedge c \in sensitivityList(p)$
       then $scheduled := scheduled \cup \{p\}$
       endif
       $event(c) := false,$
       $phase := updateSignals$
endif
```

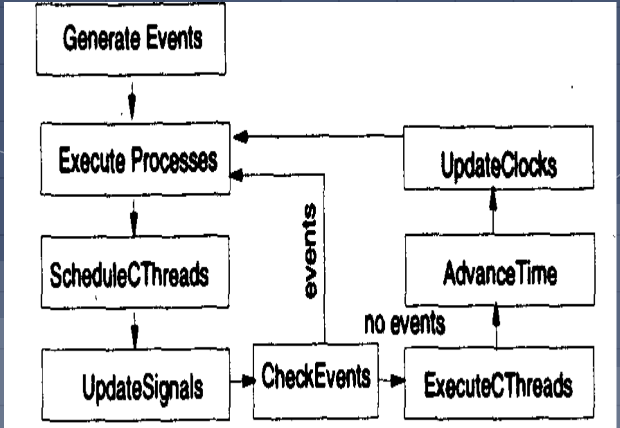


Phases of the kernel operation

- After scheduling the cthread's, their outputs are updated if the new value of the output signal does not equal the old value.
- Each signal update generates an event.
- Other events of other signals are reset to false.

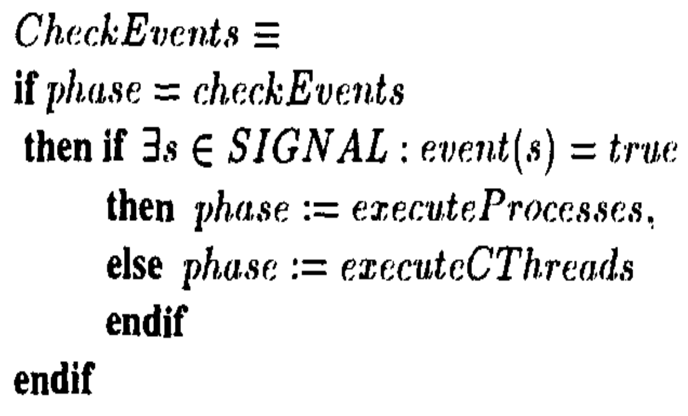


```
$UpdateSignals \equiv$
if $phase = updateSignals$
then  var $s$ ranges over $SIGNAL$
        if $value(s) \neq newValue(s)$
        then  $value(s) := newValue(s)$,
                $event(s) := true$
        else  $event(s) := false$
        endif
        $phase := checkEvents$
endif
```

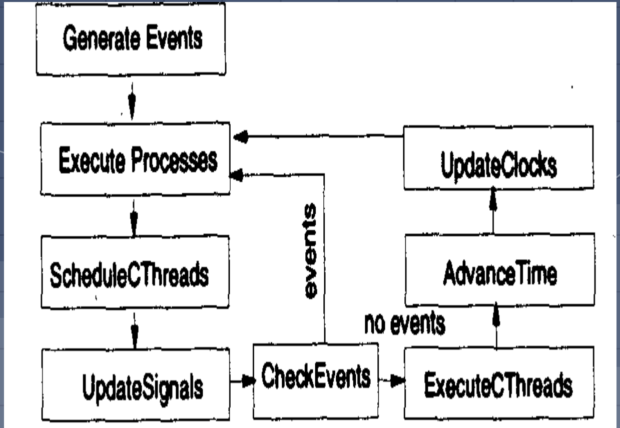


Phases of the kernel operation

- After the signals are updated, an event check has to be performed.
- The next phase checks if any events have been generated on signals and sets the next phase either to Execute Processes or to execute the cthreads.

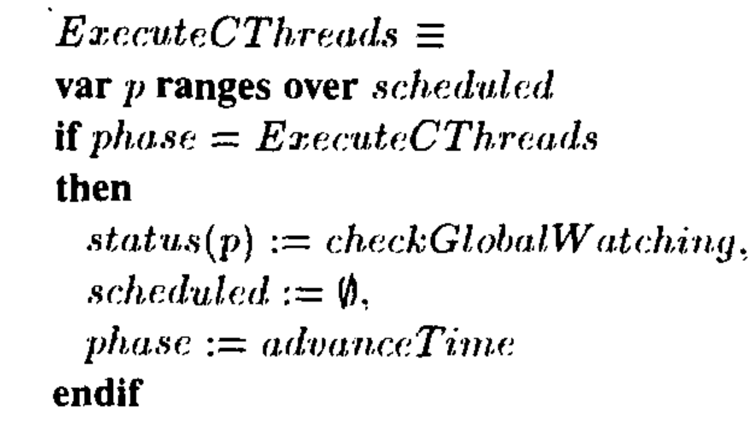


```
$CheckEvents \equiv$
if $phase = checkEvents$
 then if $\exists s \in SIGNAL : event(s) = true$
      then $phase := executeProcesses,$
      else $phase := executeCThreads$
      endif
endif
```

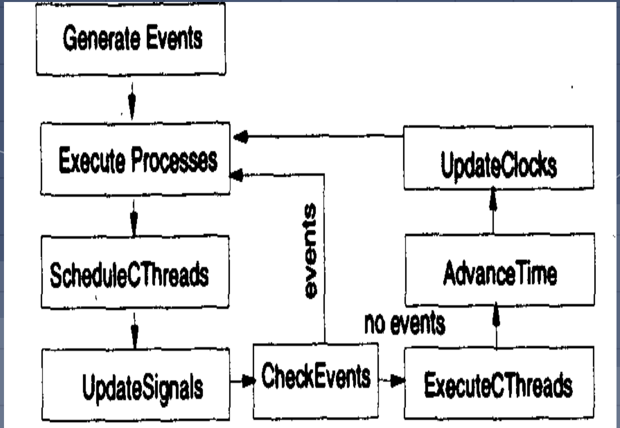


Phases of the kernel operation

- When no further events are generated at the current time T, the set of postponed cthread's are executed by setting their status to checkGlobalWatching which later on proceeds to active.
- Additionally, the set of scheduled processes has to be reset for the next cycle, and phase proceeds to advance Time.

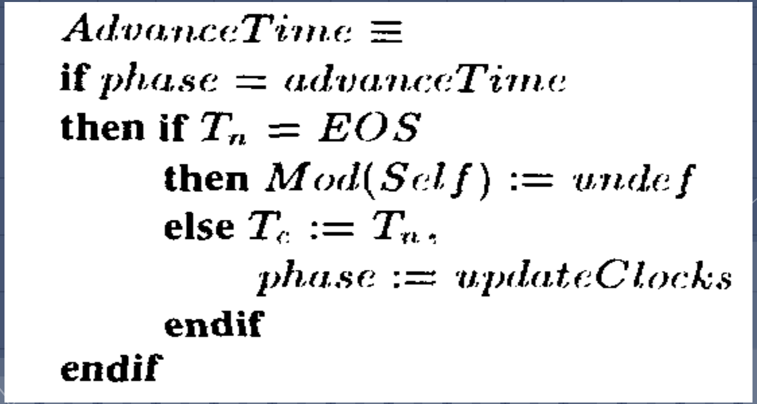


```
$ExecuteCThreads \equiv$
var $p$ ranges over $scheduled$
if $phase = ExecuteCThreads$
then
  $status(p) := checkGlobalWatching,$
  $scheduled := \emptyset,$
  $phase := advanceTime$
endif
```

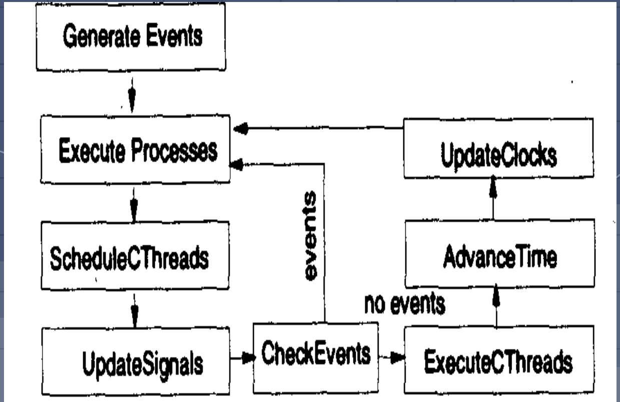


Phases of the kernel operation

- For advancing the time, we first have to check for the final end of simulation (EOS) at which the simulation kernel is deactivated.
- Otherwise, the current time Tc is advanced to the next point in time Tn, and clocks are updated accordingly.



```
$AdvanceTime \equiv$
if $phase = advanceTime$
then if $T_n = EOS$
        then $Mod(Self) := undef$
        else $T_c := T_n$,
                $phase := updateClocks$
        endif
endif
```

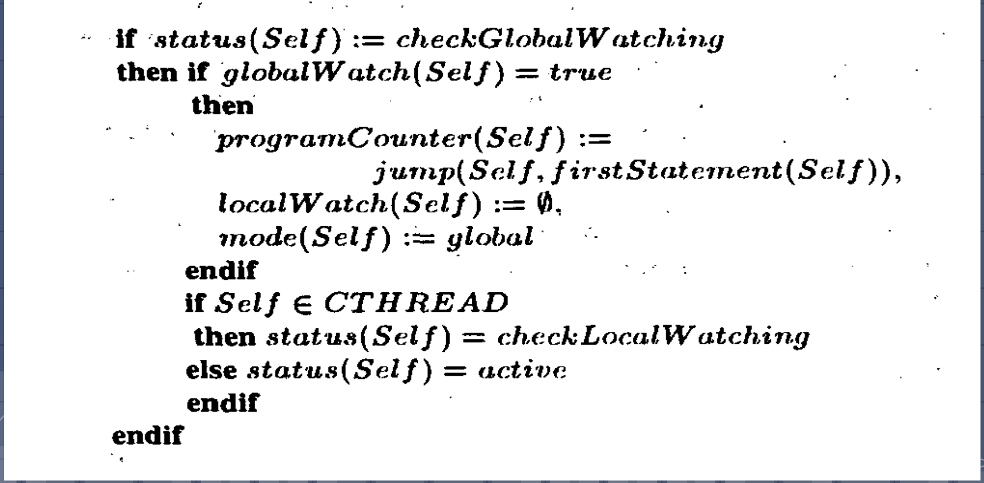


Watching conditions

- After initialization and before becoming active, each process first has to check for global and local watching conditions.
- A global watching can be defined for clocked and unclocked threads.
- A local watching are only defined for CThreads.
- If a watching condition evaluates to true then the continuation of the program, either:
- (i) Resets to the first statement of the thread Cthread in the case of a global watching or
- (ii) proceeds to the first statement within the local watching block.

Global Watching status

- After the execution of the last statement and before the beginning of the new simulation cycle, the global check has to be performed.
- The pseudo code for global watching status is shown in the figure below.



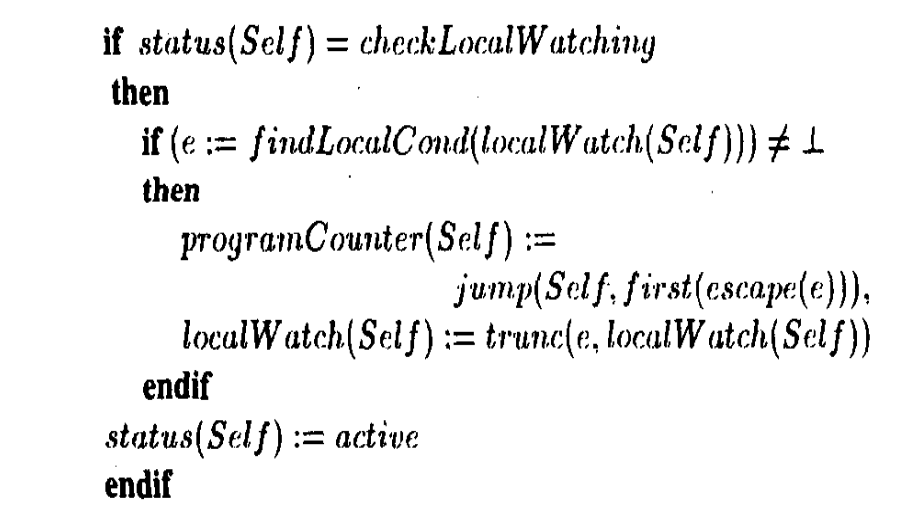
```
if $status(Self) := checkGlobalWatching$
then if $globalWatch(Self) = true$
        then
          $programCounter(Self) :=$
                         $jump(Self, firstStatement(Self)),$
          $localWatch(Self) := \emptyset,$
          $mode(Self) := global$
        endif
        if $Self \in CTHREAD$
         then $status(Self) = checkLocalWatching$
        else $status(Self) = active$
        endif
endif
```

Global Watching status

- In status checkGlobal Watching, the global watching conditions of processes, threads and CThreads are checked.
- The globalWatch becomes true when one of its sub conditions evaluate to true.
- In that case the program counter is adjusted accordingly, i.e., it is reset to the first statement of the thread.
- Thereafter, all local watchings are in activated by setting their list to Null.
- This is done, since the priority of global watching is higher than the one for the local watchings.
- Finally, CThreads are set to status checkLocal Watching, other threads directly proceed to active.

Local Watching status

- After the Global watching status is performed for the methods, threads and CThreads, only the CThreads are put in a priority list of the local watching status.
- The pseudo code for the local watch status is shown below.



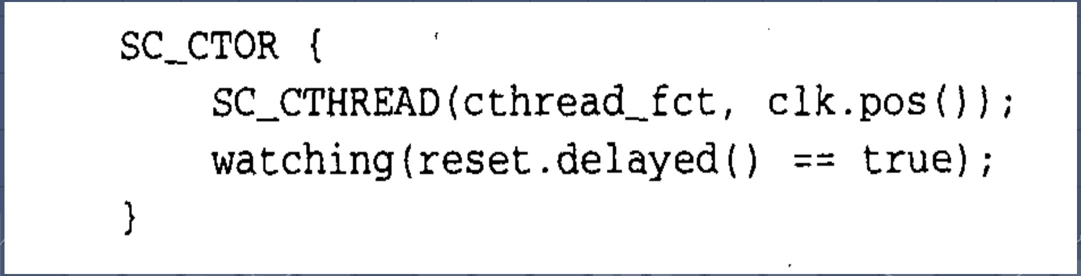
```
if $status(Self) = checkLocalWatching$
 then
   if $(e := findLocalCond(localWatch(Self))) \neq \bot$
   then
     $programCounter(Self) :=$
                         $jump(Self, first(escape(e)))$,
     $localWatch(Self) := trunc(e, localWatch(Self))$
   endif
$status(Self) := active$
endif
```

Local Watching status

- In status checkLocal Watching a CThread checks for a list element e of highest priority with true watching condition.
- If there exists a higher priority element, that element is extracted from the localwatch by the function findLocalCond.
- If it does not exist, i.e., the highest priority element is the current element, then the process Self proceeds to status active.
- Otherwise, the programCounter is reset to the first statement of the escape block of e,using first(escape(e)).
- Aditionally, trunc prunes the lower priority tail elements including e from local Watch.

SystemC watching statements

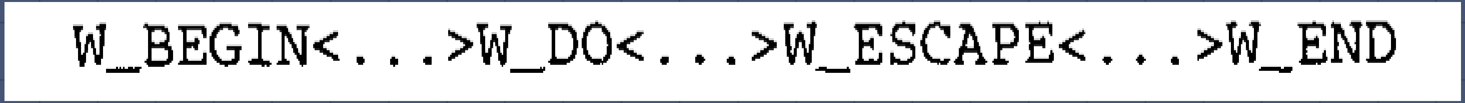
- The semantics for global and local watching statements. In order to decide if
- the current watching definition (watching(expr)) appears in the context of a global watching or of the current local one we set mode(p) belongs to either local or global.
- Global watching is usually defined in the constructor SC_CTOR of the SystemC module.



```
SC_CTOR {
     SC_CTHREAD(cthread_fct, clk.pos());
     watching(reset.delayed() == true);
}
```

SystemC watching statements

- Local watching’s are defined within the scope of a Cthread.
- Each local watching block is enclosed by W-BEGIN and W-END and has the form shown below.

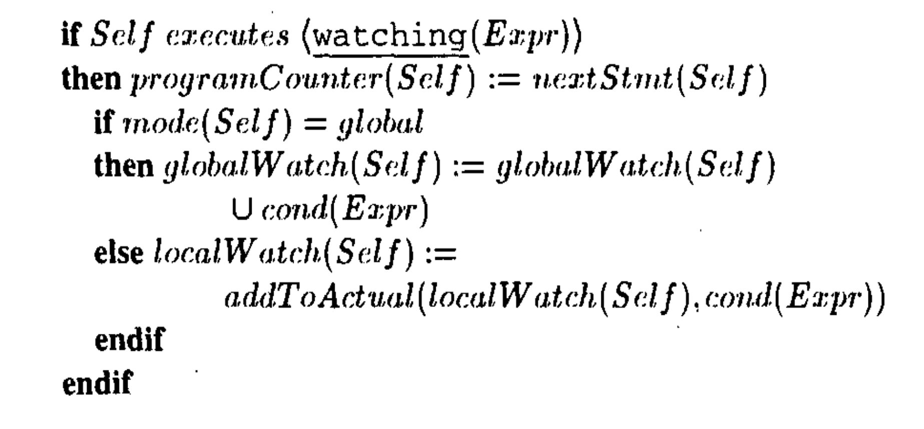


```
W_BEGIN<...>W_DO<...>W_ESCAPE<...>W_END
```

- where the list of watching conditions are specified after W-BEGIN.
- The control flow of a CThread continues after the W-DO if all watching conditions evaluate to false, then jumps to the first statement after W-ESCAPE as soon as one condition evaluates to true.
- Otherwise, that part is skipped and the program continues after W-END.

SystemC watching statements

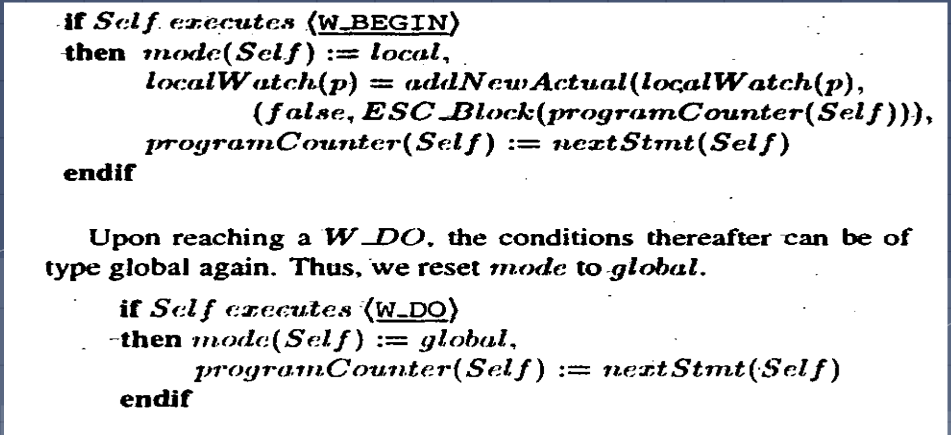
- For global watching definitions, the condition given by an individual expression is added to globalWatch.
- Otherwise, in mode local the current condition is joined with the condition of the actual
- list element of the actual local watching block.



```
if $Self$ executes $\langle$watching$(Expr)\rangle$
then $programCounter(Self) := nextStmt(Self)$
   if $mode(Self) = global$
   then $globalWatch(Self) := globalWatch(Self)$
              $\cup\ cond(Expr)$
   else $localWatch(Self) :=$
              $addToActual(localWatch(Self), cond(Expr))$
   endif
endif
```

SystemC watching statements

- Upon reaching a W-BEGIN, watching conditions are decided to be local thereafter.
- Additionally, an new actual element for the local watching list is generated through the abstract function addNewActual.
- Its initial condition is set false. All local watching conditions inside the W-BEGIN W-DO block are added to the actual localWatch element by the function addToActual.



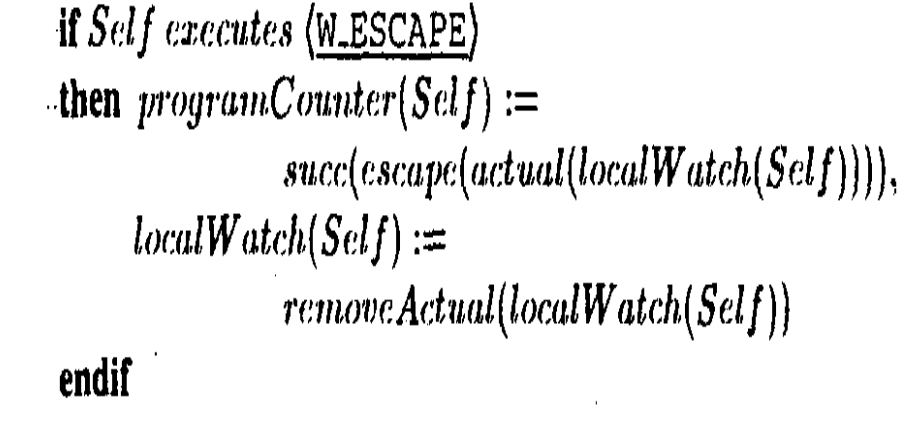
```
if Self executes ⟨W_BEGIN⟩
then mode(Self) := local,
     localWatch(p) = addNewActual(localWatch(p),
             (false, ESC_Block(programCounter(Self))),
     programCounter(Self) := nextStmt(Self)
endif
```

Upon reaching a *W_DO*, the conditions thereafter can be of type global again. Thus, we reset *mode* to *global*.

```
  if Self executes ⟨W_DO⟩
  then mode(Self) := global,
       programCounter(Self) := nextStmt(Self)
  endif
```

SystemC watching statements

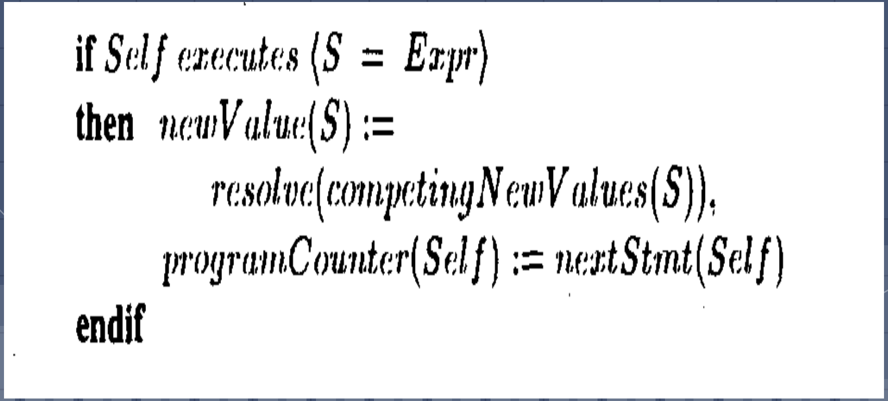
- When executing W-ESCAPE (i.e., after having completed the W-DO block) the programCounter is set to the successor of the W-END statement.
- Additionally, the currently executed local watching is removed from the priority list of local watchings by removeActual.



```
if $Self$ executes $\langle$W_ESCAPE$\rangle$
then $programCounter(Self) :=$
                $succ(escape(actual(localWatch(Self))))$,
     $localWatch(Self) :=$
                $removeActual(localWatch(Self))$
endif
```

Signal Assignment

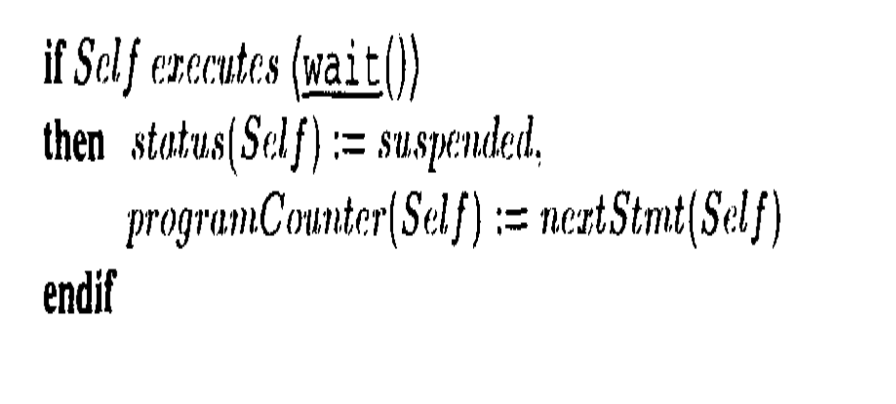
- Signal assignments are not immediately assigned to the current value of signal S but to its potential new value.
- Updating a current value with a new value generates an event if the values are different.
- The update is performed by the simulation kernel process.
- This operation is equivalent to write statements and parallel writes accesses are allowable.



```
if $Self$ executes $(S = Expr)$
then $newValue(S) :=$
          $resolve(competingNewValues(S))$,
       $programCounter(Self) := nextStmt(Self)$
endif
```

Wait Statements

- On reaching a wait statement a process simply stops execution by setting its status to suspended.



```
if Self executes (wait())
then status(Self) := suspended,
     programCounter(Self) := nextStmt(Self)
endif
```

Conclusion

- This work provides a simulation semantics for SystemC in-terms of ASM. The semantics for the two modules of kernel process and user process are explained. The entire simulation kernel phase is explained and modeled in-terms of ASM.
- This work provides a very nice insight in to the modeling of the underlying simulator semantics and the different System C statements implementation using above described ASM.

References

- SystemC: From the Ground Up by David C. Black
- W. Mueller, J.Ruf, D. Hofmann, J. Gerlach, T. Kropf, W.Rosenstiehl. "The Simulation Semantics of SystemC," in Proceedings of DATE, 2001.
- Y. Gurevich. Evolving algebra 1993: Lipari guide. In E. Borger, editor, Specification and Validation Methods. Oxford University Press, Oxford, 1994.
- Open SystemC Initiative, Synopsys Inc,-CoWare Inc. Frontier Inc. SYSTEM C Version 0.9 Users Guide, 1999.
- http://web.eecs.umich.edu/gasm/intro.html