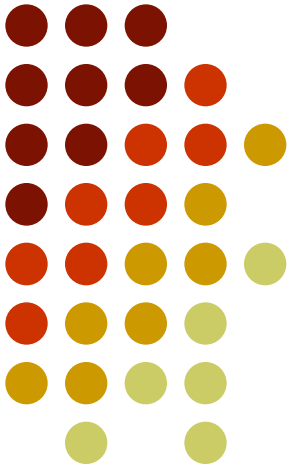
SISTEME DE CALCUL DEDICATE

Curs 3



Outline

- SystemC
  - Time
  - Concurrency
- Bibliography

Time

- three unique time measurements:
  - the simulation's wall-clock time
    - the time from the start of execution to completion, including time waiting on other system activities and applications.
  - the simulation's processor time
    - the actual time spent executing the simulation, which will always be less than the simulation's wall-clock time
  - the simulated time
    - the time being modeled by the simulation
      - it may be less than or greater than the simulation's wall-clock time

Time

- SystemC simulation performance is a combination of many factors:
  - the host system
  - system load
  - the C++ compiler
  - the SystemC simulator
  - the model being

Time

- data type sc_time – used by the simulation kernel
  - to track simulated time
  - to specify delays and timeouts
- sc_time is represented by a minimum of a 64-bit unsigned integer

```
sc_time name…; // no initialization
sc_time name(double, sc_time_unit)…;
sc_time name(const sc_time&)…;
```

Time

- time units
  - are defined by the enumeration **sc_time_unit**

enum	Units	Magnitude
SC_FS	femtoseconds	10^{-15}
SC_PS	picoseconds	10^{-12}
SC_NS	nanoseconds	10^{-9}
SC_US	microseconds	10^{-6}
SC_MS	milliseconds	10^{-3}
SC_SEC	seconds	10^{0}

Time

- all objects of **sc_time** use a single (global) time resolution
  - that has a default of 1 picosecond
  - the **sc_time** class provides get and set methods
  - **sc_set_time_resolution**()
    - may be used to change time resolution once and only once in a simulation
    - the change must occur before both creating objects of **sc_time** and starting the simulation.

```
//positive power of ten for resolution
sc_set_time_resolution(double, sc_time_unit);
```

Time

- objects of sc_time
  - may be used as operands for assignment, arithmetic, and comparison operations
  - provides conversion methods to convert sc_time to a double (to_double()) or to a double scaled to seconds (to_seconds())

```
sc_time t_PERIOD(5, SC_NS);
sc_time t_TIMEOUT(100, SC_MS);
sc_time t_MEASURE, t_CURRENT, t_LAST_CLOCK;
t_MEASURE = (t_CURRENT-t_LAST_CLOCK);
if (t_MEASURE > t_HOLD) { error("Setup violated") }
```

Time

- SystemC simulation kernel tracks simulated time using an sc_time object
  - **sc_time_stamp**() can be used to obtain the current simulated **time_value**

```
sc_time current_time = sc_time_stamp();
```

```
cout << "  The time is now "
          << sc_time_stamp()
          << "!" << endl;
```

Time

- method **sc_start**() is used to start simulation

```
//sim "forever"
sc_start();
//sim no more than max_sc_time
sc_start(const sc_time& max_sc_time);
//sim no more than max_time time_unit's
sc_start(double max_time, sc_time_unit time_unit);
```

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

Time

- simulations use delays in simulated time to model
  - real world behaviors
  - mechanical actions
  - chemical reaction times
  - signal propagation
- **wait**() method provides a syntax to allow this delay in **SC_THREAD** processes

```
wait(delay_sc_time); // wait specified amount of
                     // time
```

Concurrency

- concurrency is fundamental to simulating with SystemC
- SystemC uses **simulation processes** to model concurrency
  - event-driven simulator
  - concurrency is not true concurrent execution
  - simulated concurrency works like cooperative multitasking
    - a simulation process runs: it is expected to execute a small segment of code and then return control to the simulation kernel

Concurrency

- SystemC simulation processe **(SC_THREAD)**
  - simply C++ function
    - designated by the programmer to be used as processes (**process registration**)
  - it can be used only within a SystemC module
    - the function must be a member function of the module class
  - must be used only during the elaboration stage
  - the member function must exist and the function can take no arguments and return no values

```
SC_THREAD(MEMBER_FUNCTION);
```

Concurrency

- processes must voluntarily yield control
  - executing a **return**
  - calling SystemC's **wait()** function
- processes typically begin execution at the start of simulation and continue in an endless loop until the simulation ends

```
//FILE: two_processes.h
SC_MODULE(two_processes) {
  void wiper_thread(void); // process
  void blinker_thread(void); // process
  SC_CTOR(two_processes) {
    SC_THREAD(wiper_thread); // register process
    SC_THREAD(blinker_thread); // register process
  }
};
```

Concurrency

```
//FILE: two_processes.cpp
void two_processes::wiper_thread(void) {
  while (true) {
    wipe_left();
    wait(500,SC_MS);
    wipe_right();
    wait(500,SC_MS);
  }//endwhile
}

void two_processes::blinker_thread(void) {
  while (true) {
    blinker = true;
    cout << "Blink ON" << endl;
    wait(300,SC_MS);
    cout << "Blink OFF" << endl;
    blinker = false;
    wait(300,SC_MS);
  }//endwhile
}
```

Concurrency

- **DEFINITION**: a SystemC event is the occurrence of an **sc_event** notification and happens at a single instant in time
- an event has no duration or value
- **RULE**: to observe an event, the observer must be watching for the event prior to its notification

```
sc_event event_name1[,event_name2]…;
```

```
event_name.notify(void);          // Immediate
event_name.notify(SC_ZERO_TIME);// Delayed
event_name.notify(sc_time);       // Timed (time>0)
event_name.notify(double,units);// Convenience
```

Concurrency

- events are explicitly caused using the notify() method of an sc_event object
- invoking an immediate **notify(void)** causes any processes waiting for the event to be immediately moved from the waiting set into the runnable set for execution

```
sc_event A_event;
A_event.notify(10,SC_NS);
A_event.notify( 5,SC_NS); // only this one stays
A_event.notify(15,SC_NS);
```

Concurrency

- thread processes rely on the **wait**() method to suspend their execution

```
wait(time);                          // timeout is the event
wait(double,time_unit);              // convenience
wait(event);                         // single event
wait(event1 | event2,...);           // any of these
wait(event1 & event2,...);           // all of these
wait(time,event);                    // event or timeout
wait(time,event1 | event2,...);// any event or timeout
wait(time,event1 & event2,...);// all events or timeout
wait(); // static sensitivity - discussed later
```

Concurrency

```
…
sc_event ack_event, bus_error_event;
…
sc_time start_time(sc_time_stamp());
wait(t_MAX_DELAY, ack_event | bus_error_event);
if (sc_time_stamp()-start_time == t_MAX_DELAY) {
  break; // path for a time out
  …
```

Concurrency

- SystemC has more than one type of process
  - The **SC_METHOD** process is in some ways simpler than the **SC_THREAD**
  - **SC_METHOD** processes never suspend internally (i.e., they can never invoke **wait**()
  - **SC_METHOD** processes run completely and return

```
SC_METHOD(process_name);//Located INSIDE constructor
```

Concurrency

- **SC_METHOD** processes dynamically specify their sensitivity by means of the **next_trigger**() method

```
next_trigger(time);
next_trigger(timeout,time_unit); //convenience
next_trigger(event);
next_trigger(event1 | eventi…); //any of these
next_trigger(event1 & eventi…); //all of these
                                //required
next_trigger(timeout,event); //event with timeout
next_trigger(timeout,event1 | eventi…);//any + timeout
next_trigger(timeout,event1 & eventi…);//all + timeout
next_trigger(void); //re-establish static sensitivity
```

Concurrency

- The concept of actively determining what will cause a process to resume is often called **dynamic sensitivity**
- **Static sensitivity** establishes the parameters for resumption during elaboration (i.e., before simulation begins).
- Once established, static sensitivity parameters cannot be changed (i.e., they're static).

```
// IMPORTANT: Must follow process registration
sensitive << event [<< event]…; // streaming style
```

Concurrency

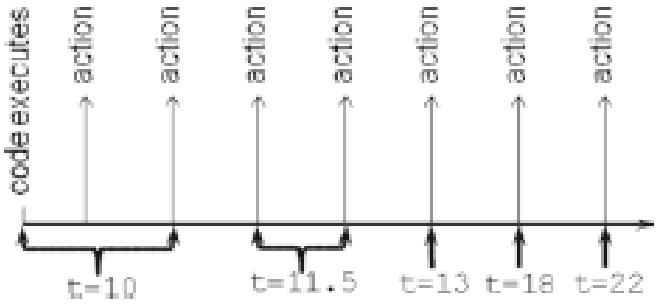
- the simulation engine description specifies that processes are executed at least once initially by placing processes in the runnable set during the initialization stage
- it may be necessary to specify that some processes should not be made runnable at initialization
  - SystemC provides the **dont_initialize**() method

```
…
SC_METHOD(attendant_method);
  sensitive(fillup_request);
  dont_initialize();
…
```

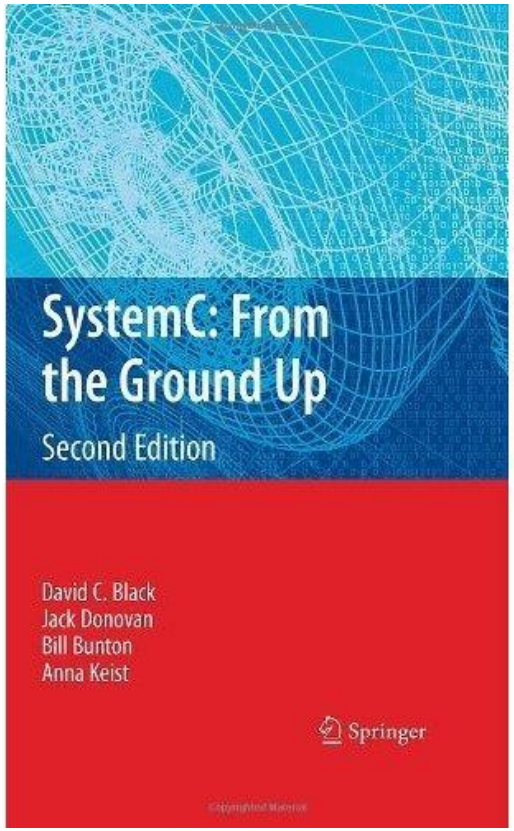
Concurrency

- **sc_event_queue**
  - allows a single event to be scheduled multiple times even for the same instant in time

```
sc_event_queue action;
wait(10,SC_NS)//assert time=10ns
sc_time now1(sc_time_stamp());//observe current time
action.notify(20,SC_NS);//schedule for 20ns from now
action.notify(10,SC_NS);//schedule for 20ns from now
action.cancel_all();//cancel all actions entirely
action.notify(8,SC_NS);//schedule for 8 ns from now
action.notify(1.5,SC_NS);// 1.5 ns from now
action.notify(1.5,SC_NS);// another identical action
action.notify(3.0,SC_NS);// 3.0 ns from now
action.notify(SC_ZERO_TIME);//after all runnable
action.notify(SC_ZERO_TIME);//and yet another
action.notify(12,SC_NS);// 12 ns from now
sc_time now2(sc_time_stamp());//observe current time
```



Bibliography



- David C. Black, Jack Donovan, Bill Bunton, Anna Keist, ***SystemC:From the Ground Up,*** Springer Science+Business Media, LLC 2010
  - *"*The authors designed this book primarily for the student or engineer new to SystemC. This book's structure is best appreciated by reading sequentially from beginning to end."