FUNCTIONAL SPECIFICATION FOR SYSTEMC 2 . 0

General terminology

Method	A C++ method, i.e., a member function of a class.
Module	A structural entity, which can contain processes, ports, channels, and other modules. Modules allow expressing structural hierarchy.
Interface	provides a set of method declarations. Provides no method implem. and no data fields.
Channel	A channel implements 1+ interfaces. Container for communication functionality.
Port	Object through which a module can access a channel's interface. But modules can also access a channel's interface directly.
Primitive Chnl	Atomic: it doesn't contain processes or modules. Cannot directly access other channels
Hierarch. Chnl	It is a module: can contain processes and modules, can directly access other channels.
Event	A process can suspend/sensitive to, 1+ events. Allow resuming/activating processes.
Sensitivity	Defines when this process will be resumed or activated. A process can be sensitive to a set of events. Whenever one of the corresponding events is triggered, the process is resumed or activated.
Static Sensitivity	The sensitivity is declared statically; i.e., it is declared during elaboration and cannot be changed once simulation has started. A so-called sensitivity list is used to define the static set of events.
Dynamic Sensit.	Can be altered during simulation.
IMC	Interface Method Call
RPC	Remote Procedure Call

Process terminology

Thread	A SystemC thread has its own thread of execution, but is not preemptive.
Automatically activated	Certain module methods (processes) are activated automatically when events occur that the processes are sensitive to.
Explicitly activated	Certain module methods must be called explicitly by other code in order to be activated.
wait()	A method that suspends execution of a thread. The arguments passed to wait() determine when execution of the thread is resumed.
Ok to call wait()	*NO*: SC_METHODs - because they don't have their own thread of execution. *YES*: SC_THREADs and the code that they call can call wait().
SC_THREAD	A module method which has its own thread of execution. Can call code that calls wait(). Automatically activated. *aka* thread process.
SC_METHOD	A module method which does not have its own thread of execution, and which cannot call code that calls wait(). SC_METHODs are automatically activated. *aka* method process.
SC_CTHREAD	A module method which has its own thread of execution. Sensitive to a +/-ive clock edge. Can call wait() with a restricted argument list. Automatically activated. *aka* clocked thread processes.

Process intialization

```
SC_MODULE( my_module )
{
    // port(s)
    sc_in_clk clk;

    // process(es)
    void proc_a();
    void proc_b();

    // constructor
    SC_CTOR( my_module )
    {
        SC_THREAD( proc_a );
        sensitive << clk.pos();
        dont_initialize();  // don't initialize proc_a

        SC_METHOD( proc_b );
        sensitive << clk.neg();
        dont_initialize();  // don't initialize proc_b
    }
};
```

Prevent the scheduler from executing a thread process or method process during the initialization phase of the simulation

MODEL OF TIME

- Underlying data type for time is a 64 bit unsigned integer
- The default *time resolution* is 1 picosecond
- change the time resolution : **sc_set_time_resolution(10, SC_PS)**.
- The time resolution must be a power of ten.
- time resolution to be specified before the start of simulation.
- The time resolution can only be specified once.
- The time resolution can only be specified before any non-zero sc_time declaration.

sc_set_time_resolution(10, SC_PS);
...
wait(3.456, SC_NS);

- rounds the time to wait to 3460 ps.

sc_get_default_time_unit()
sc_clock clk1(“clk1”, 15, SC_NS);
sc_start(1000, SC_NS);

SC_FS – femtoseconds
SC_PS – picoseconds
SC_NS – nanoseconds
SC_US – microseconds
SC_MS – milliseconds
SC_SEC – seconds

Static sensitivity

```
SC_MODULE( my_module )
{
    // ports
    sc_in<int> input;
    sc_in_clk  clock;

    // processes
    void proc_a();
    void proc_b();

    // constructor
    SC_CTOR( my_module )
    {
        SC_THREAD( proc_a );
        sensitive_pos << clock;

        SC_THREAD( proc_b );
        sensitive << input;
        sensitive_neg << clock;
    }
};
```

Dynamic sensitivity with the wait() method

```
...
// wait until event e1 has been notified
wait( e1 );
...
// wait until event e1 or event e2 has been notified
wait( e1 | e2 );
...
```

Dynamic sensitivity

- wait() can be called anywhere in the thread of execution of a thread process.
- When called,
  - specified events temporarily overrule the sensitivity list
  - the calling thread process suspends
  - When one (or all) of the specified events is notified, the waiting thread process is resumed
  - The calling process is again sensitive to the sensitivity list.
- When the wait() method is called without arguments
  - the calling thread process will suspend.
  - When one of the events in the sensitivity list is notified, the waiting thread process isresumed.
- The static sensitivity of the calling thread process doesn't change.

Forms of wait()

```
// wait on events in sensitivity list (SystemC 1.0).
wait();

// wait on event e1.
wait( e1 );

// wait on events e1, e2, or e3.
wait( e1 | e2 | e3 );

// wait on events e1, e2, and e3.
wait( e1 & e2 & e3 );

// wait for 200 ns.
wait( 200, SC_NS );

// wait on event e1, timeout after 200 ns.
wait( 200, SC_NS, e1 );

// wait on events e1, e2, or e3, timeout after 200 ns.
wait( 200, SC_NS, e1 | e2 | e3 );

// wait on events e1, e2, and e3, timeout after 200 ns.
wait( 200, SC_NS, e1 & e2 & e3 );

sc_time t( 200, SC_NS );
```

Forms of wait()

```
// wait for 200 ns.
wait( t );

// wait on event e1, timeout after 200 ns.
wait( t, e1 );

// wait on events e1, e2, or e3, timeout after 200 ns.
wait( t, e1 | e2 | e3 );

// wait on events e1, e2, and e3, timeout after 200 ns.
wait( t, e1 & e2 & e3 );

// wait for 200 clock cycles, SC_CTHREAD only (SystemC 1.0).
wait( 200 );

// wait one delta cycle.
wait( 0, SC_NS );

// wait one delta cycle.
wait( SC_ZERO_TIME );
```

NEXT_TRIGGER() METHOD

- The wait() method only used with SC_THREADs.
- **next_trigger()** method ≡ for SC_METHOD
- same arguments as **wait()**
- ≠:
  - **next_trigger()** returns immediately, without passing control to another process
  - Multiple **next_trigger()** calls allowed in one activation of an SC_METHOD process
  - The last **next_trigger()** call determines the (dynamic) sensitivity for the next activation

- Constructor – An event object can be created by calling the constructor without any arguments. For example,

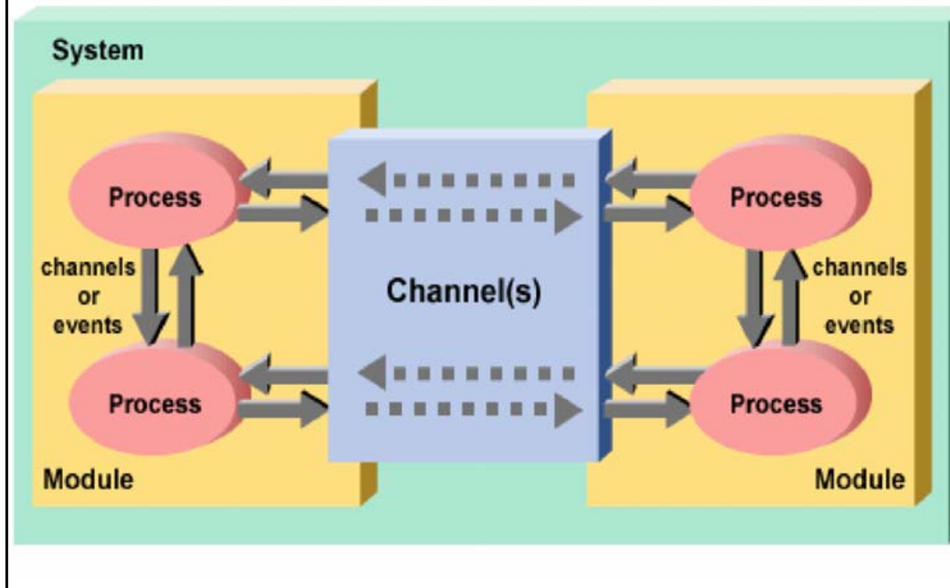
```
sc_event my_event;
```

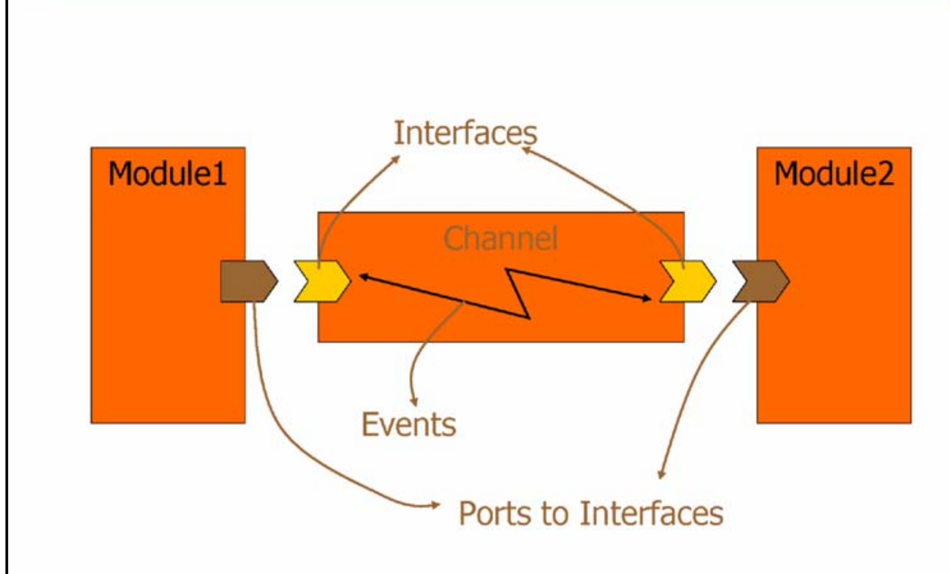
- Notify – An event can be notified by calling the (non-const) `notify()` method of the event object. For example,

```
my_event.notify();                 // notify immediately
my_event.notify( SC_ZERO_TIME );   // notify next delta cycle
my_event.notify( 10, SC_NS );      // notify in 10 ns
sc_time t( 10, SC_NS );
my_event.notify( t );              // same
```

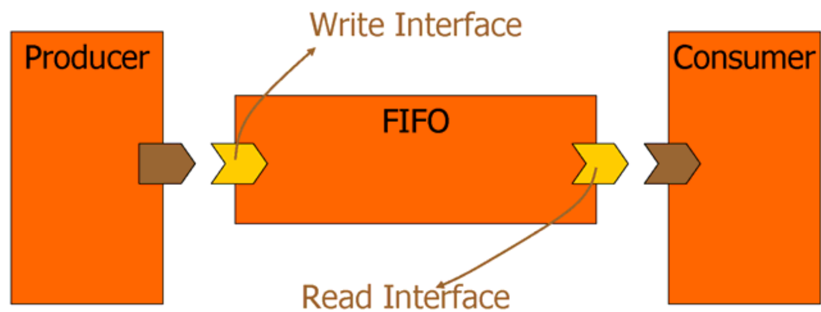
In addition, functions are provided allowing a functional notation for notifying events. For example,

```
notify( my_event );                 // notify immediately
notify( SC_ZERO_TIME, my_event );   // notify next delta cycle
notify( 10, SC_NS, my_event );      // notify in 10 ns
sc_time t( 10, SC_NS );
notify( t, my_event );              // same
```


A system in SystemC
System
Process
Process
channels
or
events
Process
Process
Channel(s)
channels
or
events
Module
Module


Communication and
Synchronization (cont'd)
Interfaces
Module1
Module2
Channel
Events
Ports to Interfaces

A Communication Modeling Example: FIFO



FIFO Example: Declaration of Interfaces

```
class write_if : public sc_interface
{
   public:
        virtual void write(char) = 0;
        virtual void reset() = 0;
};

class read_if : public sc_interface
{
   public:
        virtual void read(char&) = 0;
        virtual int num_available() = 0;
};
```

Declaration of *FIFO channel*

```
class fifo: public sc_channel,
   public write_if,
   public read_if
{
   private:
        enum e {max_elements=10};
        char data[max_elements];
        int num_elements, first;
        sc_event write_event,
                 read_event;
        bool fifo_empty() {…};
        bool fifo_full() {…};

   public:
        fifo() : num_elements(0),
                 first(0);
```

```
void write(char c) {
   if (fifo_full())
        wait(read_event);
   data[ <you calculate> ] = c;
   ++num_elements;
   write_event.notify();
}

void read(char &c) {
   if (fifo_empty())
        wait(write_event);
   c = data[first];
   --num_elements;
   first =  …;
   read_event.notify();
}
```

Declaration of *FIFO channel* (cont'd)

```
   void reset() {
        num_elements = first = 0;
   }

   int num_available() {
        return num_elements;
   }
}; // end of class declarations
```

FIFO Example (cont'd)

- Any *channel* must
  - be derived from *sc_channel* class
  - be derived from one (or more) classes derived from *sc_interface*
  - provide implementations for all pure virtual functions defined in its parent *interfaces*

FIFO Example (cont'd)

- Note the following wait() call
    - wait(*sc_event*) => dynamic sensitivity
    - wait(*time*)
    - wait(*time_out, sc_event*)
  - Events
    - are the fundamental synchronization primitive
    - have no type, no value
    - always cause sensitive processes to be resumed
    - can be specified to occur:
      - immediately/ one delta-step later/ some specific time later

Completing the Comm. Modeling Example

```
SC_MODULE(producer) {
   public:
        sc_port<write_if> out;

   SC_CTOR(producer) {
        SC_THREAD(main);
   }

   void main() {
        char c;
        while (true) {
           out.write(c);
           if(…)
                out.reset();
        }
   }
};
```

```
SC_MODULE(consumer) {
   public:
        sc_port<read_if> in;

   SC_CTOR(consumer) {
        SC_THREAD(main);
   }

   void main() {
        char c;
        while (true) {
           in.read(c);
           cout<<

   in.num_available(); }
   }
};
```

Completing the Comm. Modeling Example (cont'd)

```
SC_MODULE(top) {
   public:
        fifo afifo;
        producer *pproducer;
        consumer *pconsumer;

   SC_CTOR(top) {
        pproducer=new producer("Producer");
        pproducer->out(afifo);

        pconsumer=new consumer("Consumer");
        pconsumer->in(afifo);
   };
```

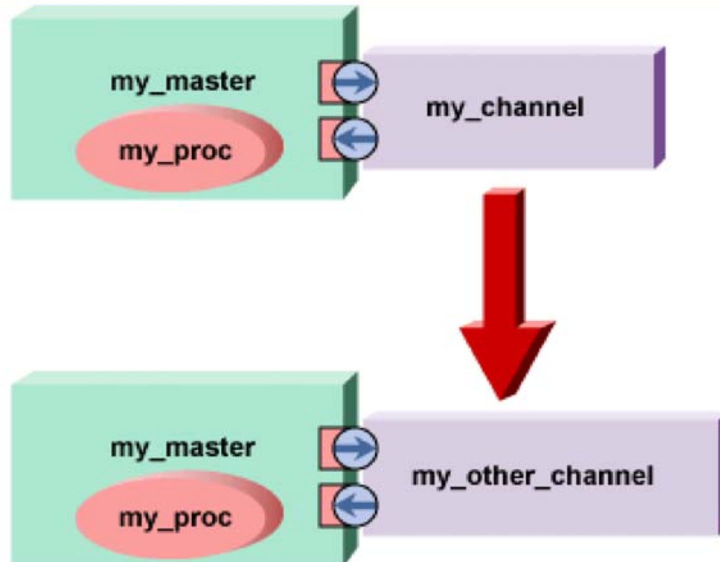
Completing the Comm. Modeling Example (cont'd)

- Note:
  - Producer module
    - sc_port<*write_if*> out;
      - Producer can only call member functions of *write_if* interface
  - Consumer module
    - sc_port<*read_if*> in;
      - Consumer can only call member functions of *read_if* interface
  - Producer and consumer are
    - unaware of how the channel works
    - just aware of their respective *interfaces*
  - Channel implementation is hidden from communicating modules

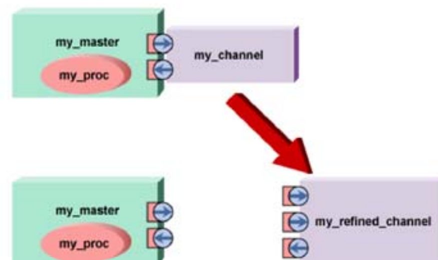
Completing the Comm. Modeling Example (cont'd)

- Advantages of separating communication from functionality
  - Trying different communication modules
  - Refine the FIFO into a software implementation
    - Using queuing mechanisms of the underlying RTOS
  - Refine the FIFO into a hardware implementation
    - Channels can contain other channels and modules
      - Instantiate the hw FIFO module within FIFO channel
      - Implement read and write interface methods to properly work with the hw FIFO
      - Refine read and write interface methods by inlining them into producer and consumer codes

SystemC refinement

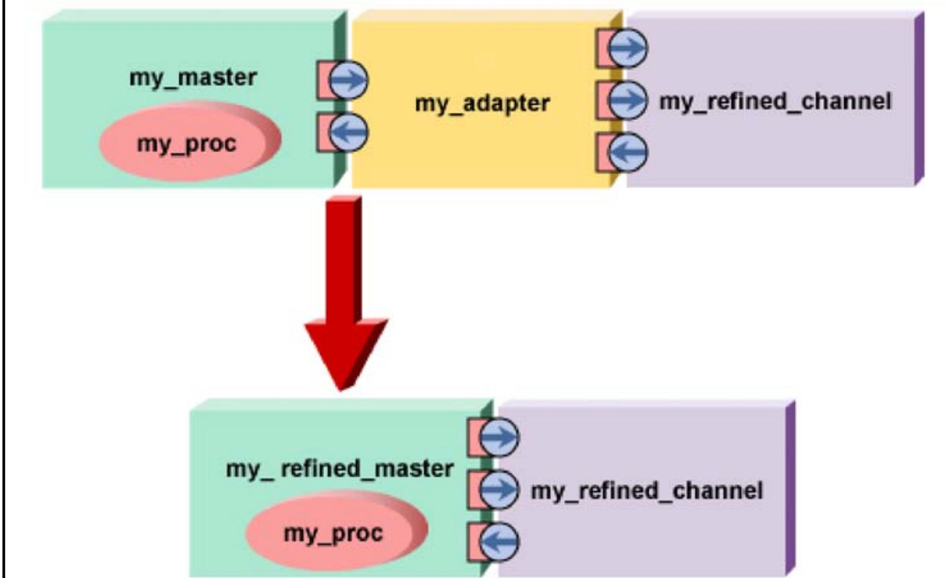


SystemC Channel Replacement



SystemC Adapter Insertion

SystemC Adapter Merge



SystemC scheduler

- Like most modeling languages SystemC has a simulation kernel, too
- Event Based Simulation for modeling concurrency
- Processes executed & their outputs updated based on events
- Processes are scheduled based on their sensitivity to events
- Similarity with key VHDL simulation kernel aspects is evident

SystemC & VHDL Similarities

```
SC_THREAD(proc_1);
sensitive << Trig.pos( );
SC_THREAD(proc_2);
sensitive << Trig.pos( );
```

- Which process should go first?
- Does it actually matter?
- On sc_signals follows VHDL paradigm
  - Process execution and signal update done in 2 phases, order of processes does not matter
  - Concept of delta cycles
  - Simulation is deterministic
- But SystemC can model concurrency, time & communication in other ways as well

SystemC & non Determinism

- Delta Cycle = Evaluation Phase + Update Phase
  - Presence of 2 phases guarantees determinism
- But for modeling S/W we need non-determinism
  - Employ the notify( ) method of an sc_event (see previous producer/consumer example)

SystemC Scheduler & Events

- notify() with no arguments
  - Called Immediate Notification
  - Processes sensitive to this event will run in current evaluation phase
- notify(0)
  - Processes sensitive to this event will run in evaluation phase of next delta cycle
- notify(t) with t>0
  - Processes sensitive to this event will run during the evaluation phase of some future simulator time

SystemC Simulator Kernel

1. Init: execute all processes in unspecified order
2. Evaluate: Select a ready to run process & resume its execution. May result in more processes ready for execution due to Immediate Notification
3. Repeat 2 until no more processes to run
4. Update Phase
5. If 2 or 4 resulted in delta event notifications, go back to 2
6. No more events, simulation is finished for current time
7. Advance to next simulation time that has pending events. If none, exit
8. Go back to step 2

Metropolis vs. SystemC

- Metro more general model of computation
- Different operational & denotational semantics
- Metro Formal Semantics & tools
- Metro Quantity Managers
  - For performance analysis
  - For modeling of Operating Systems

INTERFACE EXAMPLES

```
// -----------------------------------------
//  An example read interface: sc_read_if
//  this interface provides a 'read' method
// -----------------------------------------

template <class T>
class sc_read_if
: virtual public sc_interface
{
public:

    // interface methods
    virtual const T& read() const = 0;
};


// ------------------------------------------
//  An example write interface: sc_write_if
//  this interface provides a 'write' method
// ------------------------------------------

template <class T>
class sc_write_if
: virtual public sc_interface
{
public:

    // interface methods
    virtual void write( const T& ) = 0;
};
```

Read/write interface

```
//  An example read/write interface: sc_read_write_if
// -------------------------------------------------

template <class T>
class sc_read_write_if
: public sc_read_if<T>,
  public sc_write_if<T>
{};
```

INTERFACE BASE CLASS

All interfaces are (directly or indirectly) derived from base class sc_interface.

```
class sc_interface
{
public:

    // register a port with this interface (does nothing by default)
    virtual void register_port( sc_port_base&, const char* ) {}

    // get the default event
    virtual const sc_event& default_event() const;

    // destructor (does nothing)
    virtual ~sc_interface() {}
};
```

SIMPLE PORTS

- object through which a module, and hence its processes, can access a channel's interface.
- Modules can also access a channel's interface directly
- In SystemC 1.0, we have three basic port types:
  - sc_in<T>, sc_out<T>, and sc_inout<T>
  - They are all derived from the base class sc_port
  - Each of these port types provides
    - a set of interface methods, such as read() and write().
    - what these methods basically do is to call the corresponding interface method of the attached channel.
- With other channel types, this simple scheme has to be extended
  - the interfaces assumed by sc_in<T>, sc_out<T>, and sc_inout<T>, are not sufficient for all channel types.

General Ports

- Some channel types may require additional or altogether different interface methods.
- ≠ interfaces can be created by
  - refining predefined interface types
  - or by inheriting directly from sc_interface
- Separate function and communication
  - to increase the reusability of components
- A good design style
  - is to always select the “minimal” port type that offers the required interface methods

ATTACHING MULTIPLE INTERFACES

- SystemC 2.0 allows for connecting a port to multiple channels *implementing the same interface*. ⇒ the *multi-port* capability.

```
class simple_bus_if
: virtual public sc_interface
{
public:

    // methods return false if address out of range
    virtual bool read_data( unsigned address, int& data ) = 0;
    virtual bool write_data( unsigned address, int data ) = 0;
};
```

```
class simple_mem_if
: public simple_bus_if
{
public:

    // methods to determine address range
    virtual unsigned start_address() const = 0;
    virtual unsigned end_address() const = 0;
};
```

```
// --------------------------
//  CHANNEL : simple_bus
// --------------------------

class simple_bus
: public sc_module,
  public simple_bus_if
{
public:

    // a port to connect memories to (maximum 10 in this case)
    sc_port<simple_mem_if,10> mem_port;

    // interface methods

    virtual bool read_data( unsigned address, int& data )
    {
        // inefficient, but illustrates the use model
        for( int i = 0; i < mem_port.size(); i ++ )
            if( ( address >= mem_port[i]->start_address() ) &&
                ( address <= mem_port[i]->end_address() ) )
                return mem_port[i]->read_data( address, data );
        return false;
    }

    virtual bool write_data( unsigned address, int data );

    ...
};
```

sc_mutex_if

```
class sc_mutex_if
: virtual public sc_interface
{
    public:
        virtual int lock() = 0;
        virtual int trylock() = 0;
        virtual int unlock() = 0;

    protected:
        sc_mutex_if();

    private:
        // Disabled
        sc_mutex_if( const sc_mutex_if& );
        sc_mutex_if& operator= ( const sc_mutex_if& );
};
```

sc_mutex channel

```
class sc_mutex
: public sc_mutex_if, public sc_prim_channel
{
    public:
        sc_mutex();
        explicit sc_mutex( const char* );

        virtual int lock();
        virtual int trylock();
        virtual int unlock();

        virtual const char* kind() const;

    private:
        // Disabled
        sc_mutex( const sc_mutex& );
        sc_mutex& operator= ( const sc_mutex& );
};
```

sc_mutex channel: lock()

- virtual int **lock**();
  - If the mutex is unlocked
    - member function **lock** shall lock the mutex and return
  - If the mutex is locked
    - suspend until the mutex is unlocked (by another process)
    - attempt to lock the mutex.
  - Member function **lock** shall unconditionally return the value **0**.
- If multiple processes attempt to lock the mutex in the same delta cycle
  - which process instance is given the lock in that delta cycle
    - non-deterministic
  - relies on the order in which processes are resumed within the evaluation phase.

sc_mutex channel: trylock(), kind()

- virtual int **trylock**();
  - If the mutex is unlocked
    - member function **trylock** shall lock the mutex
    - shall return the value **0**
  - If the mutex is locked
    - member function **trylock** shall immediately return the value **-1**.
    - The mutex shall remain locked

- virtual const char* **kind**() const;
  - Member function **kind** shall return the string **"sc_mutex"**.

sc_mutex channel: unlock();

- virtual int **unlock**();
  - mutex unlocked,
    - member function **unlock** shall return the value **-1**
    - The mutex shall remain unlocked
  - mutex locked by ≠ process
    - member function**unlock** shall return the value **-1**
    - The mutex shall remain locked
  - mutex locked by the calling process
    - member function **unlock** shall unlock the mutex
    - shall return the value **0**
    - If processes suspended and waiting for the mutex
      - lock shall be given (nondeterministic) to exactly one of these processes
      - remaining processes shall suspend again
      - This shall be accomplished within a single evaluation phase
        - ⇒ use immediate notification to signal the act of unlocking a mutex

Port-less access for intra-module level communication

```
// pseudo code

SC_MODULE( mod1 )
{
    // port(s)
    sc_in<int>  in;
    sc_out<int> out;

    sc_mutex mutex; // channel; used to protect common resource 'in'

    int f( int );
    int g( int );
```

Port-less access (cont'd)

```
void thread1()
{
    while( true ) {
        wait( 10, SC_NS );
        mutex.lock();                // direct access to channel
        wait( in.value_changed_event() );
        out = f( in );
        mutex.unlock();              // direct access to channel
    }
}
```

Port-less access (cont'd)

```
void thread2()
{
    while( true ) {
        wait( 10, SC_NS );
        mutex.lock();               // direct access to channel
        wait( in.value_changed_event() );
        out = g( in );
        mutex.unlock();             // direct access to channel
    }
}
```


Port-less access (cont'd)

```
void thread2()
{
    while( true ) {
        wait( 10, SC_NS );
        mutex.lock();               // direct access to channel
        wait( in.value_changed_event() );
        out = g( in );
        mutex.unlock();             // direct access to channel
    }
}
```

Port-less access (cont'd)

```
SC_MODULE( mod1 )
{
    ...
    // constructor
    SC_CTOR( mod1 )
    {
        SC_THREAD( thread1 );
        SC_THREAD( thread2 );
    }

    ...
};
```