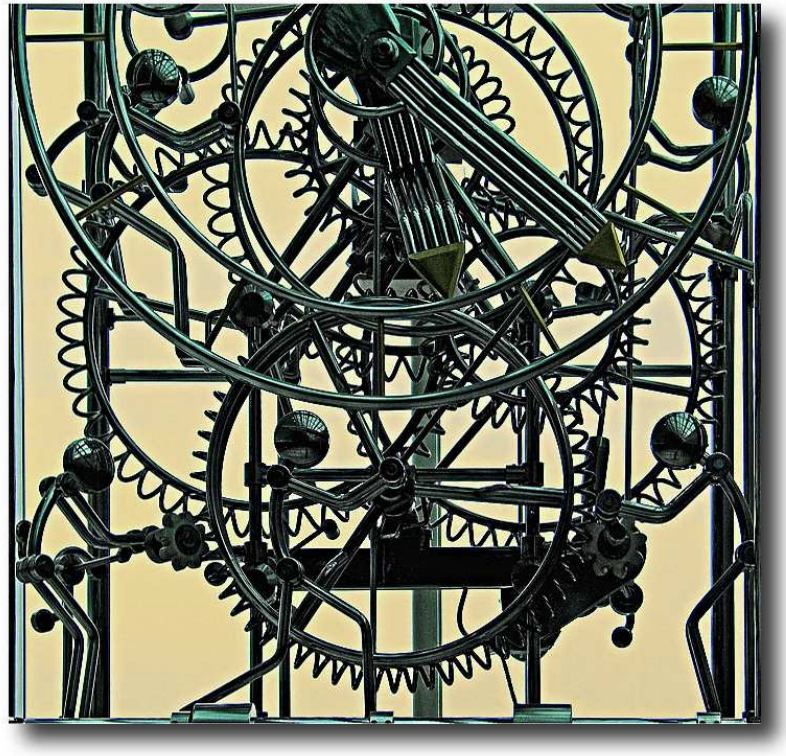
Introduction to SystemC

Deian Tabakov

Rice University
Houston, TX

September 10, 2007



Introduction

The brave new world

Time-to-market : This year's gadgets must replace last year's

- Shorter design cycles
- Find and fix most bugs early
- Early prototyping and design exploration
- Design software and hardware in parallel

Complexity : Designing in the age of the iPhone

- Moore's Law
- SoC: Hardware + software on a single die
- Concurrent systems
- IP development, reuse, exchange
- Specifying systems

Introduction

Toward a new language: desiderata

- Specification and design at various levels of abstraction
- Executable specification
- Executable platform models
- Fast simulation speed
- Separating communication from computation
- Easy to learn
- Object-oriented.

Toward a new language: desiderata

- Specification and design at various levels of abstraction
- Executable specification
- Executable platform models
- Fast simulation speed
- Separating communication from computation
- Easy to learn
- Object-oriented.

SystemC in a nutshell

- A "system-level" modeling language
  - Several levels of abstraction (from purely functional to cycle accurate pin-accurate)
  - Special attention to systems with embedded software

Introduction

SystemC in a nutshell

- A "system-level" modeling language
  - Several levels of abstraction (from purely functional to cycle accurate pin-accurate)
  - Special attention to systems with embedded software
- A library of C++ templates and classes for modeling concurrent systems
  - Hardware-oriented data types
  - Communication mechanism
  - Concurrency model

SystemC in a nutshell

- A "system-level" modeling language
  - Several levels of abstraction (from purely functional to cycle accurate pin-accurate)
  - Special attention to systems with embedded software
- A library of C++ templates and classes for modeling concurrent systems
  - Hardware-oriented data types
  - Communication mechanism
  - Concurrency model
- An event-driven simulation kernel for executing models
- Available for free (Windows + Linux)

Modeling an EXOR gate

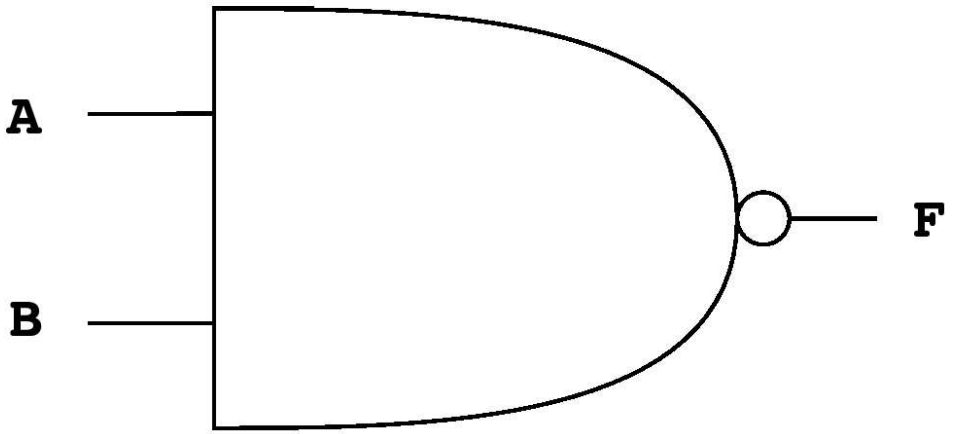


Figure: NAND gate

Modeling an EXOR gate

Example (nand.h)

```
#include "systemc.h"

SC_MODULE(nand) {           // declare a NAND sc_module
  sc_in<bool> A, B;         // input signal ports
  sc_out<bool> F;           // output signal ports

  void do_it() {            // a C++ function
    F.write( !(A.read() && B.read()) );
  }

  SC_CTOR(nand) {           // constructor for the module
    SC_METHOD(do_it);       // register do_it() w/ kernel
    sensitive << A << B;    // sensitivity list
  }
};
```

Modeling an EXOR gate

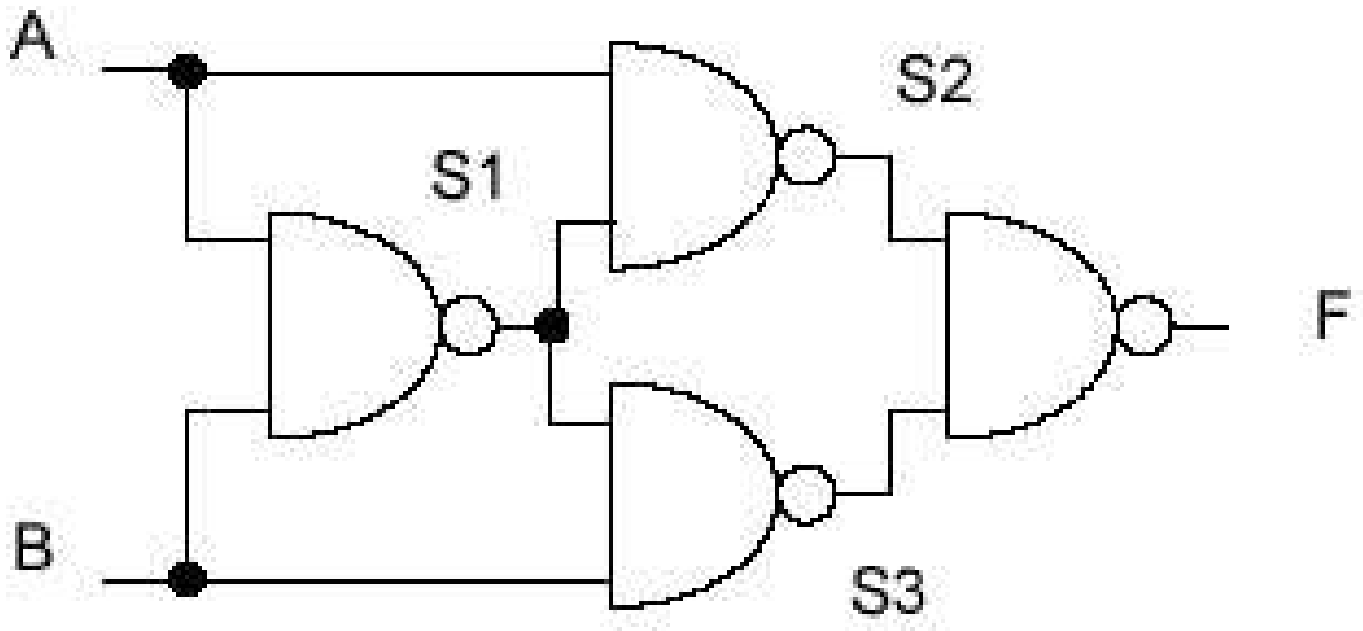


Figure: EXOR gate

Modeling an EXOR gate

Example (exor.h)

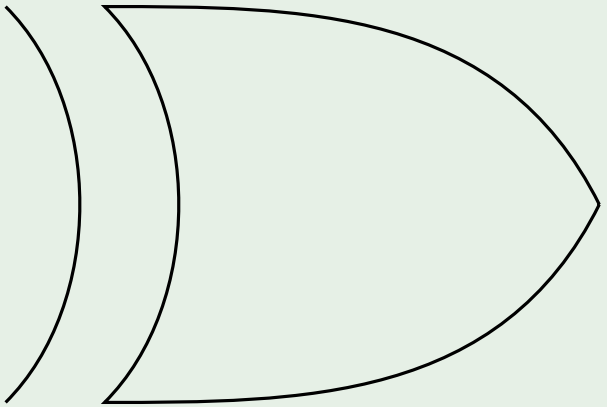
```
#include "nand.h"
SC_MODULE(exor) {
  sc_in<bool> A, B;
  sc_out<bool> F;
  nand n1, n2, n3, n4;
  sc_signal<bool> S1, S2, S3;

  SC_CTOR(exor) : n1("N1"), n2("N2"), n3("N3"), n4("N4") {
    n1.A(A);
    n1.B(B);
    n1.F(S1);

    n2 << A << S1 << S2;

    n3(S1);
    n3(B);
    n3(S3);

    n4 << S2 << S3 << F;
  }
};
```



Modeling an EXOR gate

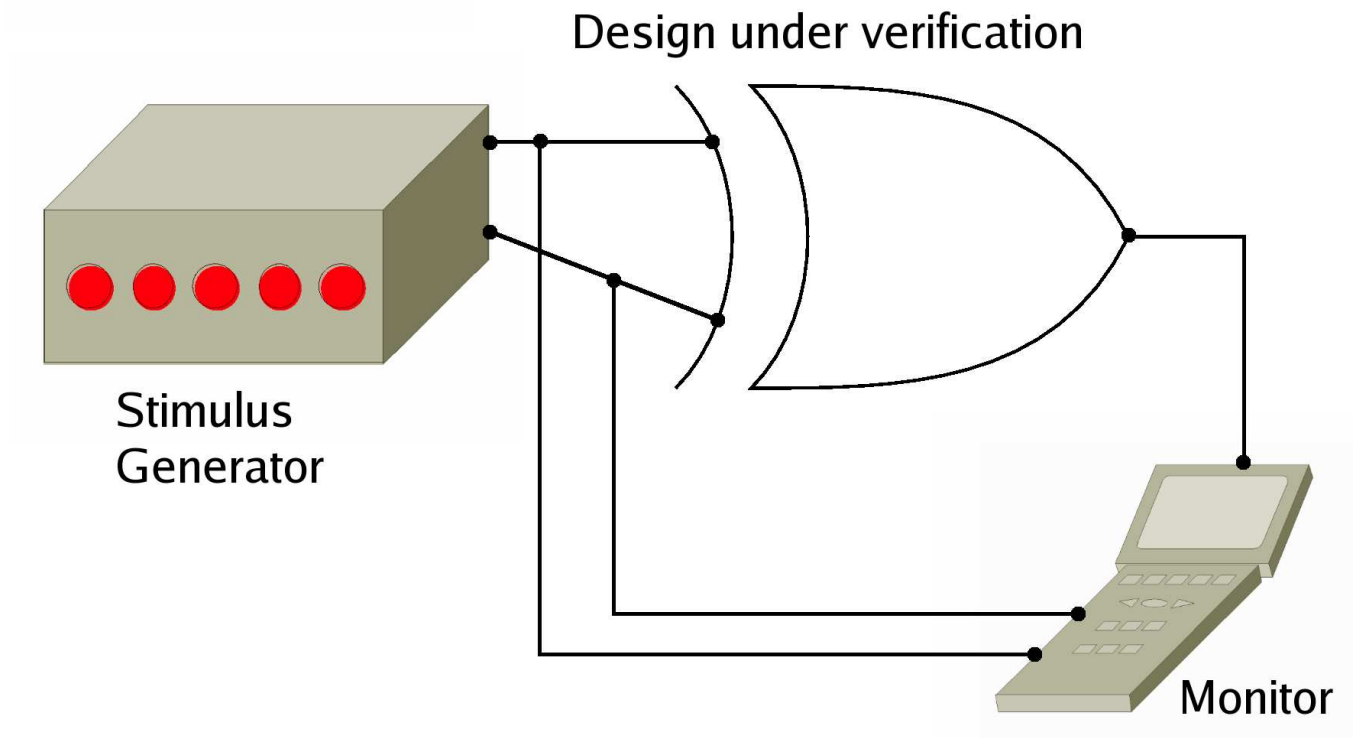


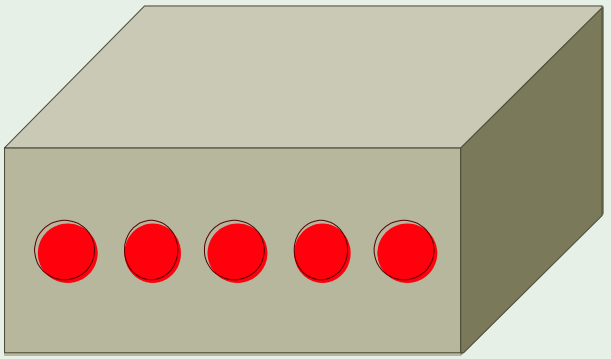
Figure: Test Bench

Modeling an EXOR gate

Example (stim.h)

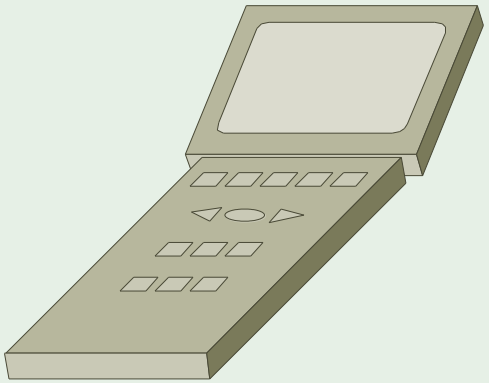
```
#include "systemc.h"
SC_MODULE(stim) {
  sc_out<bool> A, B;
  sc_in<bool> Clk;

  void StimGen()  {
    A.write(false);
    B.write(false);
    wait();                  // wait for the next clock tick
    A.write(false);
    B.write(true);
    wait();                  // wait for the next clock tick
        ...
    sc_stop();               // notify kernel to stop simulation
  }
  SC_CTOR(stim)  {
    SC_THREAD(StimGen);
    sensitive << Clk.pos();
  }
};
```



Modeling an EXOR gate

Example (mon.h)



```
#include "systemc.h"
SC_MODULE(mon) {
  sc_in<bool> A, B, F;
  sc_in_clk Clk;

  void Monitor()  {
    while(1) {
      wait();
      cout << sc_time_stamp() << "\t" << A.read()
           << " " << B.read() << " " << F.read() << endl;
    }
  }
  SC_CTOR(mon)  {
    SC_THREAD(Monitor);
    sensitive << Clk.pos();

    cout << "Time\tA B F" << endl;
  }
};
```

Modeling an EXOR gate

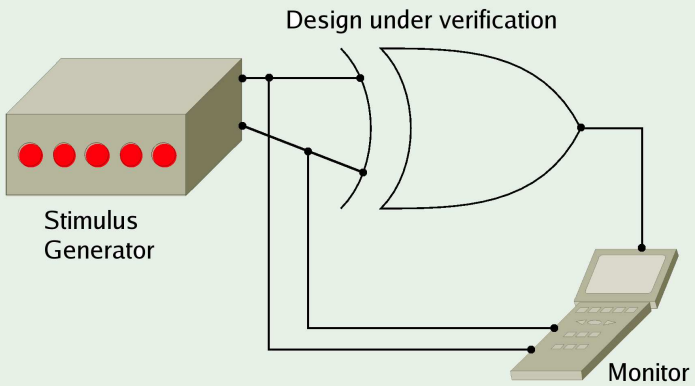
Example (Putting it all together)

```
#include "stim.h"
#include "exor.h"
#include "mon.h"

int sc_main(int argc, char* argv[]) {
  sc_signal<bool> ASig, BSig, FSig;
  sc_clock TestClk("TestClock", 10, SC_NS, 0.5);

  stim Stim1("Stimulus");     mon Monitor1("Monitor");
  Stim1.A(ASig);              Monitor1.A(ASig);
  Stim1.B(BSig);              Monitor1.B(BSig);
  Stim1.Clk(TestClk);         Monitor1.F(FSig);
                              Monitor1.Clk(TestClk);
  exor DUV("exor");
  DUV.A(ASig);
  DUV.B(BSig);
  DUV.F(FSig);

  sc_start();  // run forever
  return 0;
}
```



Modeling an EXOR gate

Example (Putting it all together)

```
#> ./sandbox

            SystemC 2.1.v1 --- Jun 13 2007 04:39:21
        Copyright (c) 1996-2005 by all Contributors
                    ALL RIGHTS RESERVED
Time     A B F
0 s      0 0 1
10 ns    0 1 1
20 ns    1 0 1
30 ns    1 1 0
SystemC: simulation stopped by user.
#>
```

Modules: the building blocks of SystemC models

- Hierarchy
- Abstraction
- IP reuse

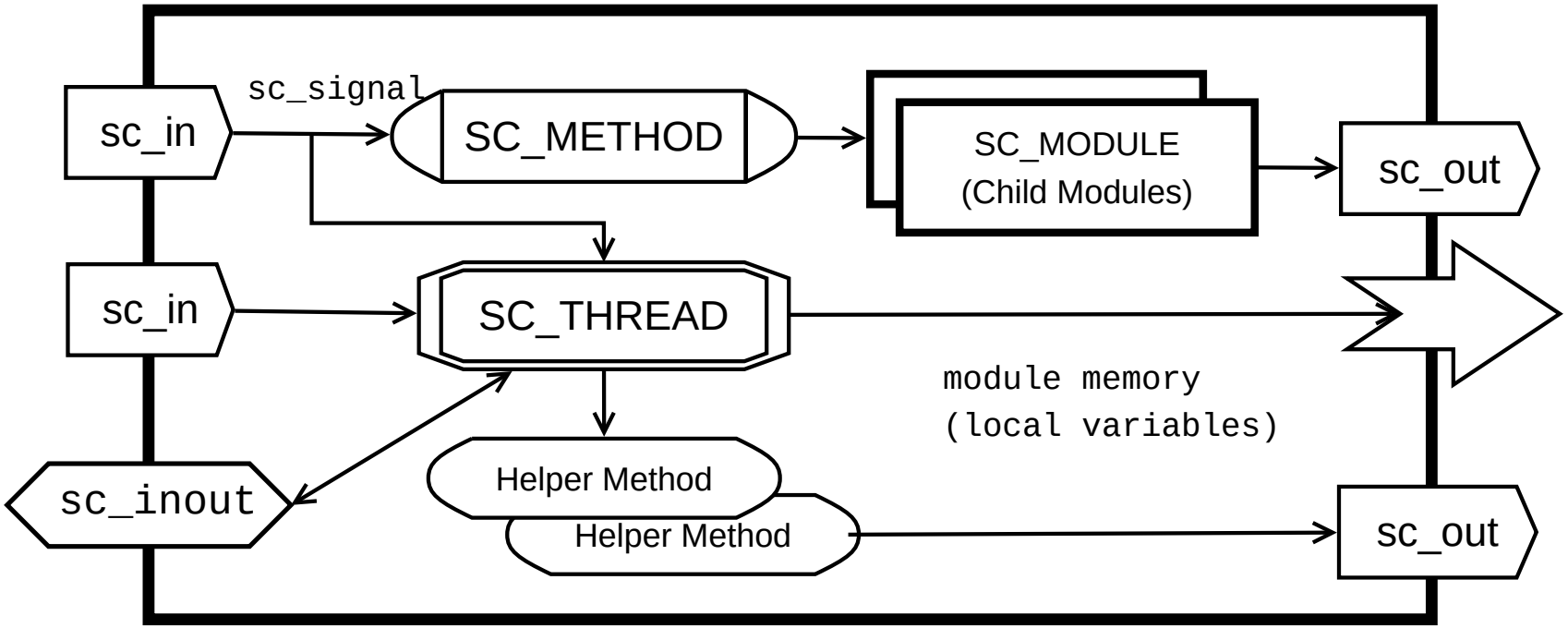


Figure: SC_MODULE

Modules: the building blocks of SystemC models

Example (Structure of a module)

```
SC_MODULE(Module_name) {
  // Declare ports, internal data, etc.
  // Declare and/or define module functions

  SC_CTOR(Module_name) {
    // Body of the constructor

    // Process declarations and sensitivities
    SC_METHOD(function1);
    sensitive << input1 << input2;

    SC_THREAD(function2);
    sensitive << input1 << clk;
  }
};
```

Processes: basic units of functionality

- `SC_THREAD`s
  - Can be suspended (`wait()`)
  - Implicitly keep state of execution
- `SC_METHOD`s
  - Execute their body from beginning to end
  - Simulate faster
  - Do not keep implicit state of execution
- Processes must be contained in a module (but not every member function is a process!)

Data types

- Four-valued logic types (01XZ)
- Arbitrary-precision integers
- Time
  - SC_SEC, SC_MS, SC_US, SC_NS, etc.
  - `sc_time t1(42, SC_NS)` creates a time object representing 42 nanoseconds
  - Integer-valued model of time

Mechanisms for Abstraction

Interfaces: windows into the communication channels

- Signature of an operation: name, params, return value
- Ignore implementation details
- No local data

Ports: helper objects of communication

- Agents for connecting modules with their environment
- Forward calls to the channel on behalf of the module

Channels: workhorses for transmitting data

- Communication between modules
- Provide the functionality defined in the interfaces

Mechanisms for Abstraction

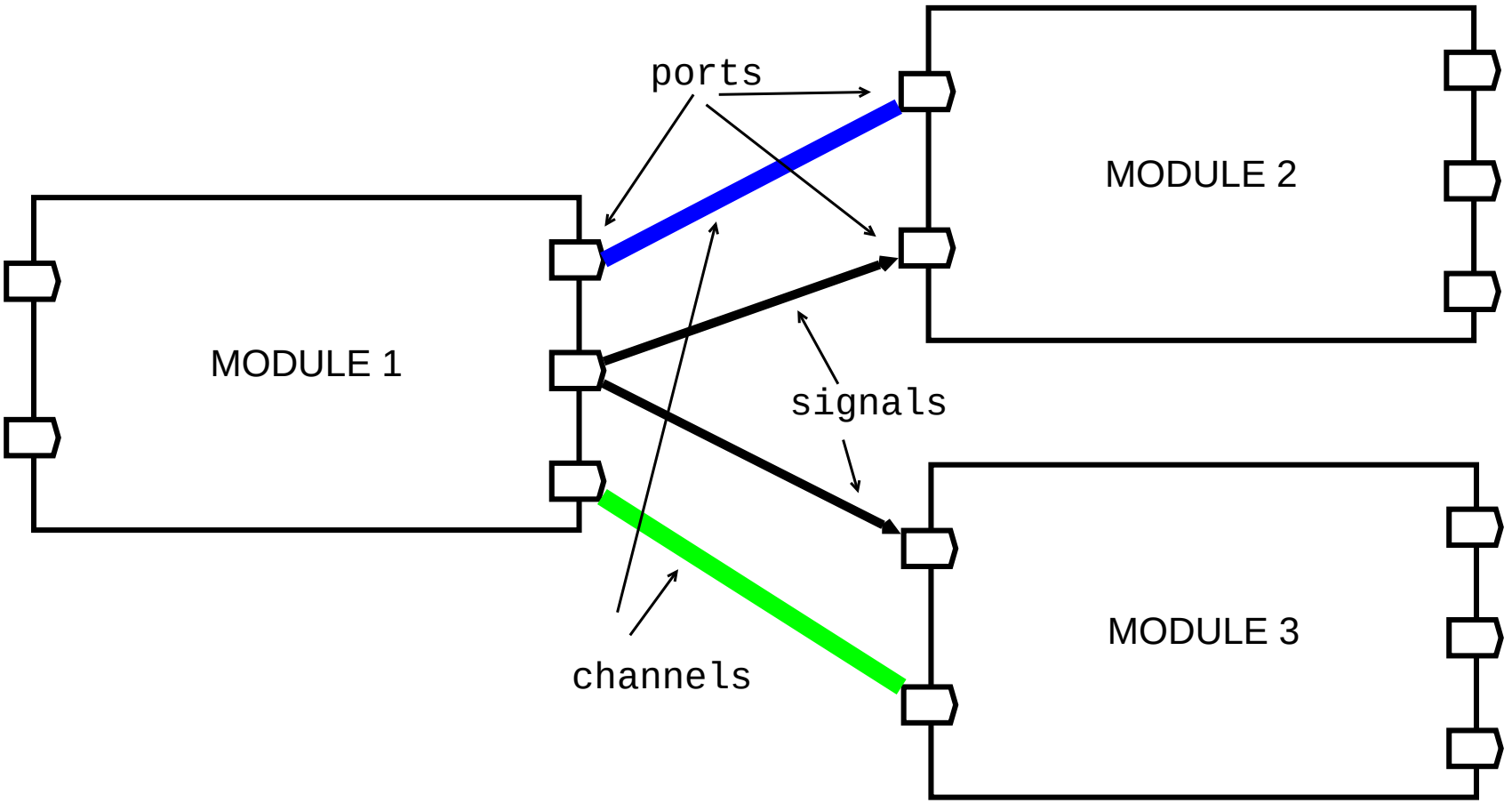


Figure: Modules, channels, ports, signals

Interfaces and Ports

Every port expects a particular type of interface

Example (nand.h)

```
#include "systemc.h"
SC_MODULE(nand) {          // declare a NAND sc_module
  sc_in<bool> A, B;        // input signal ports
  sc_out<bool> F;          // output signal ports
...
```

Interfaces and Ports

Every port expects a particular type of interface

Example (nand.h)

```
#include "systemc.h"
SC_MODULE(nand) {          // declare a NAND sc_module
  sc_port<sc_signal_in_if<bool>,1> A,B; // input sig.
  sc_out<bool> F;          // output signal ports
...
```

Interfaces and Ports

Every port expects a particular type of interface

Example (nand.h)

```
#include "systemc.h"
SC_MODULE(nand) {          // declare a NAND sc_module
  sc_port<sc_signal_in_if<bool>,1> A,B; // input sig.
  sc_port<sc_signal_out_if<bool>,1> F; // output sig.
...
```

Interfaces and Ports

Every port expects a particular type of interface

Example (nand.h)

```
#include "systemc.h"
SC_MODULE(nand) {          // declare a NAND sc_module
  sc_port<sc_signal_in_if<bool>,1> A,B; // input sig.
  sc_port<sc_signal_out_if<bool>,1> F; // output sig.
...
```

Every port can use only the methods defined in the interface

Example (nand.h)

```
...
  void do_it() {              // a C++ function
    F.write( !(A.read() && B.read()) );
  }
...
```

Channels

- Channels implement interfaces
- One channel can implement multiple interfaces
- Many channels can implement the same interface

Example (exor.h)

```
SC_MODULE(exor) {
  sc_in<bool> A, B;
  sc_out<bool> F;
  nand n1, n2, n3, n4;
  sc_signal<bool> S1, S2, S3;
...
```

For example, `sc_signal<T>` implements two interfaces: `sc_signal_in_if<T>` and `sc_signal_inout_if<T>`

Events

In SystemC events are objects (`sc_event`) that determine whether a process' execution should be triggered or resumed
An event is *notified* when a particular condition occurs

- Rising edge of a clock
- Change of a value of a signal
- Explicit call (`e.notify()`)
- An empty FIFO channel is written to
- Many others

Three types of notification: immediate, delta, and timed.
"Event notification causes processes that are sensitive to it to be triggered."

Simulation Semantics

Simulating parallel execution sequentially

- Simulation Kernel: the heart of SystemC
- Controls timing
- Controls order of execution of processes
- Handles event notification
- Updates the channels if so requested
- *Non-preemptive execution*

Simulation Semantics

Simulating parallel execution sequentially

1. Initialize
   - Mark all processes runnable (or *pending*, or *eligible*)
   - Each SC_METHOD will be executed once
   - Each SC_THREAD will be executed until the first synch point

Simulation Semantics

Simulating parallel execution sequentially

1. Initialize
2. Evaluate
   - Select an eligible process and run it
   - If the process uses *immediate* notification, other processes may become eligible too (*This is bad practice!*)
   - Continue selecting eligible processes until none are left

Simulation Semantics

Simulating parallel execution sequentially

1. Initialize
2. Evaluate
3. Update
   - One type of channels (*primitive channels*) are allowed to request an update phase
   - Handle pending update requests
   - May generate additional delta event notifications, thus rendering more processes eligible

Simulation Semantics

Simulating parallel execution sequentially

1. Initialize
2. Evaluate
3. Update
4. If there are eligible processes, loop back to the Evaluate phase (step 2)

Simulation Semantics

Simulating parallel execution sequentially

1. Initialize
2. Evaluate
3. Update
4. If there are eligible processes, loop back to the Evaluate phase (step 2)
5. Advance the time
   - At this point there are no eligible processes
   - Advance the simulation clock to the earliest pending timed notification

Simulation Semantics

Simulating parallel execution sequentially

1. Initialize
2. Evaluate
3. Update
4. If there are eligible processes, loop back to the Evaluate phase (step 2)
5. Advance the time
6. Determine eligible processes and go to Evaluate phase (step 2)

Simulation Semantics

Simulating parallel execution sequentially

1. Initialize
2. Evaluate
3. Update
4. If there are eligible processes, loop back to the Evaluate phase (step 2)
5. Advance the time
6. Determine eligible processes and go to Evaluate phase (step 2)

Note: Simulation Kernel = SystemC Scheduler = Simulation Scheduler

Simulation Semantics

Algorithm (Simulation Semantics)

```
Declare all processes runnable                    // initialization
While exists a runnable process do {              // simulation loop
   While exists a runnable process do {           // delta cycle
      /** Simulation time does not advance **/
      Run all runnable processes                  // evaluation phase
      Run all pending update requests             // update phase
      Execute all pending delta notifications     // delta phase
   }

   If exists a pending timed notification
                              or timeout t then   // timed phase
      /** Simulation time advances **/
      Advance time to earliest such t
   else
      End simulation
}
```

Modeling an EXOR gate

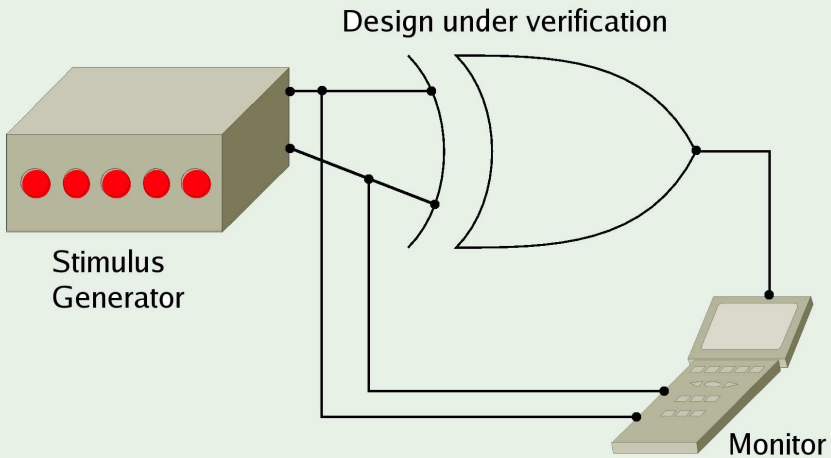
Example (Putting it all together)

```
#> ./sandbox

            SystemC 2.1.v1 --- Jun 13 2007 04:39:21
        Copyright (c) 1996-2005 by all Contributors
                    ALL RIGHTS RESERVED
Time     A B F
0 s      0 0 1
10 ns    0 1 1
20 ns    1 0 1
30 ns    1 1 0

Simulation stopped by user.

#>
```



Simulation Semantics

Example (main.cc)

```
#include "stim.h"
#include "exor.h"
#include "mon.h"
int sc_main(int argc, char* argv[]) {
  sc_signal<bool> ASig, BSig, FSig;
  sc_clock TestClk("TestClock", 10, SC_NS, 0.5);
...
```

Simulation Semantics

Example (main.cc)

```
#include "stim.h"
#include "exor.h"
#include "mon.h"
int sc_main(int argc, char* argv[]) {
  sc_signal<bool> ASig, BSig, FSig;
  sc_clock TestClk("TestClock", 10, SC_NS, 0.5, 1, SC_NS);
...
```

Simulation Semantics

Example (main.cc)

```
#include "stim.h"
#include "exor.h"
#include "mon.h"
int sc_main(int argc, char* argv[]) {
  sc_signal<bool> ASig, BSig, FSig;
  sc_clock TestClk("TestClock", 10, SC_NS, 0.5, 1, SC_NS);
...
```

Example (Post-correction output)

```
Time     A B F
1 s      0 0 0
11 ns    0 1 1
21 ns    1 0 1
31 ns    1 1 0
Simulation stopped by user.
#>
```

Summary

Core SystemC concepts

- Process : functionality
- Event : process synchronization
- Sensitivity : basis of event-driven simulation
- Channel : process communication
- Module : encapsulation
- Ports : external view of the module

Buzzwords

EDA	Electronic Design Automation
SoC	System-on-chip
TLM	Transaction-Level Model(ing)
PV	Programmer's View
PVt	Programmer's View with Time
HDL	Hardware Description Language
RTL	Register Transfer Level
ASIC	Application-Specific Integrated Circuit
IP	Intellectual Property
DSP	Digital Signal Processing
DUV	Design Under Verification