SystemC Tutorial

John Moondanos

Strategic CAD Labs, INTEL Corp.

&

GSRC Visiting Fellow, UC Berkeley

SystemC Introduction

- Why not leverage experience of C/C++ developers for H/W & System Level Design?
- But C/C++ have no
  - notion of time
    - No event sequencing
  - Concurrency
    - But H/W is inherently concurrent
  - H/W Data Types
    - No ‘Z’ value for tri-state buses

SystemC is …

- C++ Class Library use for
  - Cycle-Accurate model for Software Algorithm
  - Hardware Architecture
  - Interface of SoC (System-on-Chip)
  - System-level designs
  - Executable Specification
- www.systemc.org

SystemC Environment

your standard
C/C++ development
environment

compiler

linker

debugger

header files

libraries

SYSTEMC

class library
and
simulation kernel

DSP

Interface

IP-Core

ASIC

source files for system
and test benches

"make"

"executable
specification"

a.out

executable = simulator

SystemC History

- SystemC 1.0
  - Provide VHDL like capabilities
  - Simulation kernel
  - Fixed point arithmetic data types
  - Signals (communication channels)
  - Modules
    - Break down designs into smaller parts

SystemC History

- SystemC 2.0
  - Complete library rewrite to upgrade into true SLDL
  - Events as primitive behavior triggers
  - Channels, Interfaces and Ports
  - Much more powerful modeling for Transaction Level
- Future SystemC 3.0
  - Modeling of OSs
  - Support of embedded S/W models

Objectives of SystemC 2.0

- Primary goal: Enable System-Level Modeling
  - Systems include hardware and software
  - Challenge:
    - Wide range of design models of computation
    - Wide range of design abstraction levels
    - Wide range of design methodologies

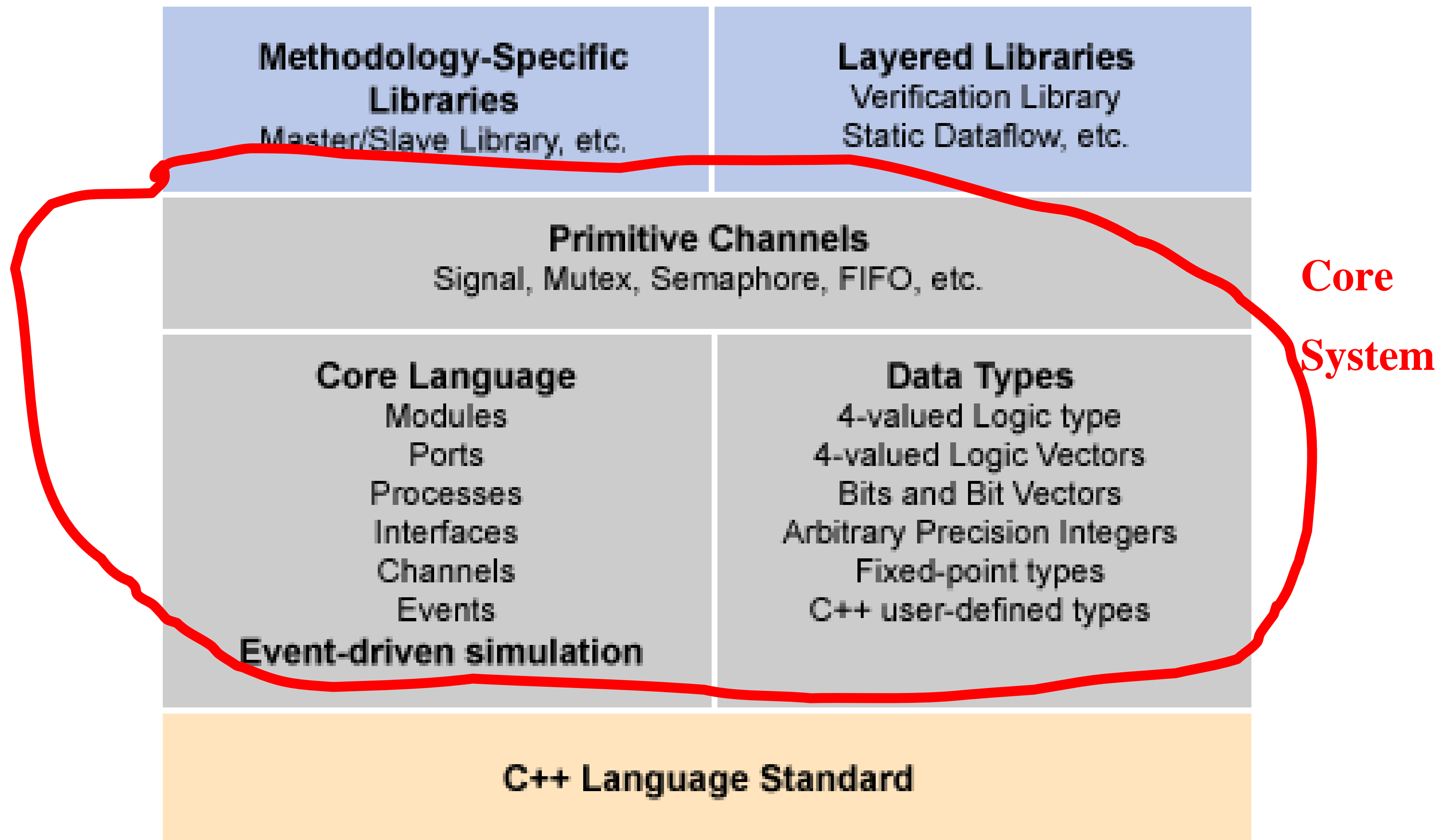
SystemC 2.0 Objectives (cont)

- SystemC 2.0
  - Introduces a small but very general purpose modeling foundation => Core Language
  - Support for other models of computation, methodologies, etc
    - They are built on top of the core language, hence are separate from it
      - Even SystemC 1.0 *Signals* are built on top of this core in SystemC 2.0
      - Other library models are provided:
        - FIFO, Timers, ...

Communication and Synchronization

- SystemC 1.0 *Modules* and *Processes* are still useful in system design
- But communication and synchronization mechanisms in SystemC 1.0 (*Signals*) are restrictive for system-level modeling
  - Communication using queues
  - Synchronization (access to shared data) using mutexes

SystemC Language Architecture



SystemC vs. Metropolis

- Constructs to model system architecture
  - Hardware timing
  - Concurrency
  - Structure
- Adding these constructs to C
  - SystemC
    - C++ Class library
    - Standard C/C++ Compiler : bcc, msvc, gcc, etc…
  - Metropolis
    - New keywords & Syntax
    - Translator for SystemC
    - Many More features…

System Design Methodology

- Current
  - Manual Conversion from C to HDL Creates Errors
  - Disconnect Between System Model and HDL Model
  - Multiple System Tests
- SystemC *(Executable-Specification)*
  - Refinement Methodology
  - Written in a Single Language

Modeling Terms (I)

- Untimed Functional (UTF)
  - Refers to model I/F and functionality
  - No time used for regulating the execution
  - Execution & data transport in 0 time
- Timed Functional (TF)
  - Refers to both model I/F and functionality
  - Time is used for the execution
  - Latencies are modeled
  - Data Transport takes time

Modeling Terms (II)

- Bus Cycle Accurate (BCA)
  - Refers to model I/F, not functionality
  - Timing is cycle accurate, tied to some global clock
  - Does not infer pin level detail
  - Transactions for data transport
- Pin Cycle Accurate (PCA)
  - Refers to model I/F not model functionality
  - Timing is cycle accurate
  - Accuracy of the I/F at the pin Level
- Register Transfer (RT) Accurate
  - Refers to model functionality
  - Everything fully timed
  - Complete detailed Description, every bus, every bit is modeled

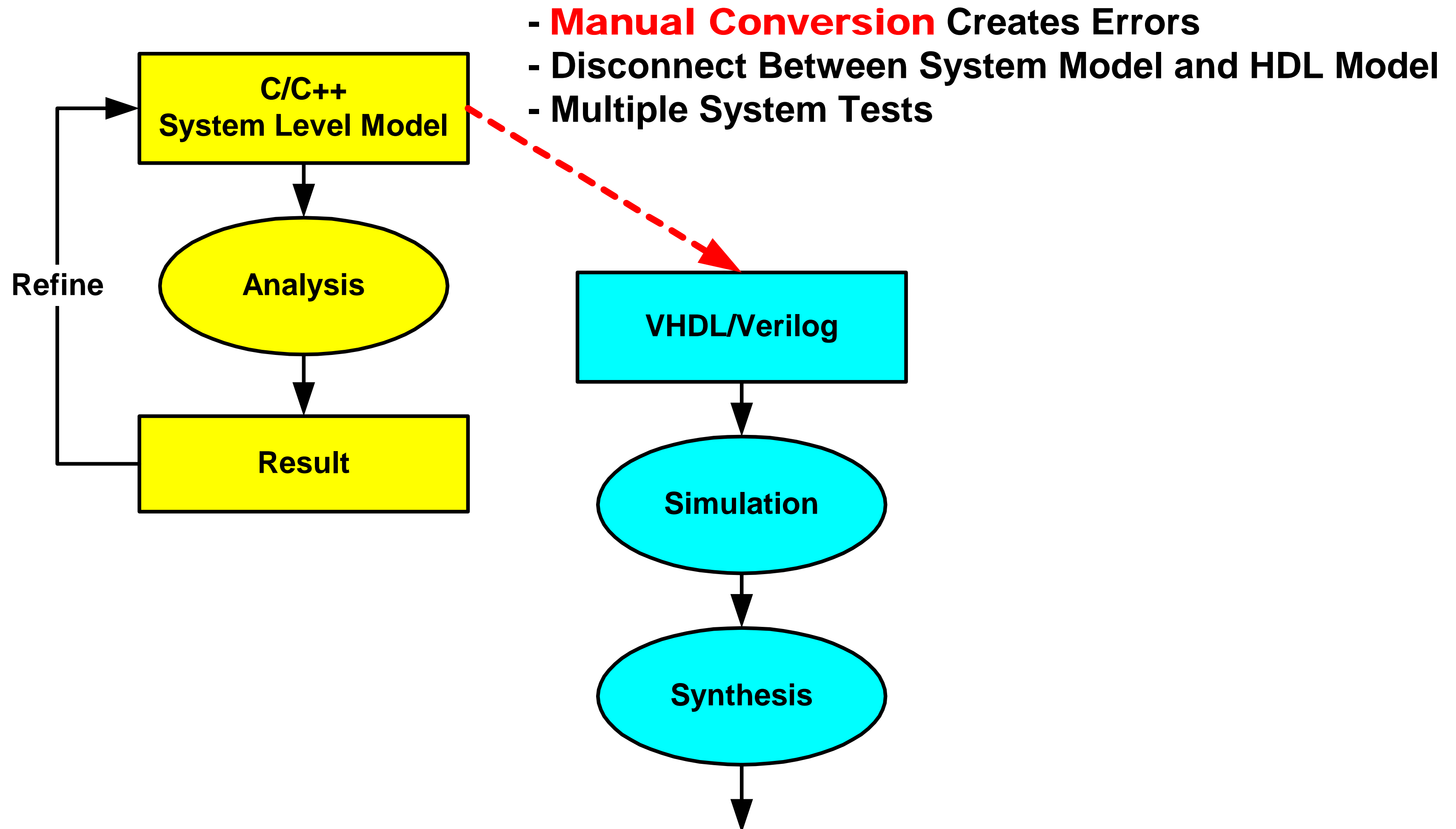
Model Types (1)

- System Architectural
  - Executable specification for H/W & S/W
  - Architecture Exploration, algorithm determination & proof
  - I/Fs are UTF with no pin detail for modeling communication protocols
  - Functionality UTF, sequential since it’s untimed
- System Performance
  - Timed executable specification for bith H/W & S/W
  - Used for time budgeting
  - Concurrent behavior modeled
- Transaction Level (TLM)
  - Typically describe H/W only
  - Model I/Fs are TF, functionality TF as well (either not cycle accurate)
  - Data Transfers & system Behavior modeled as transactions

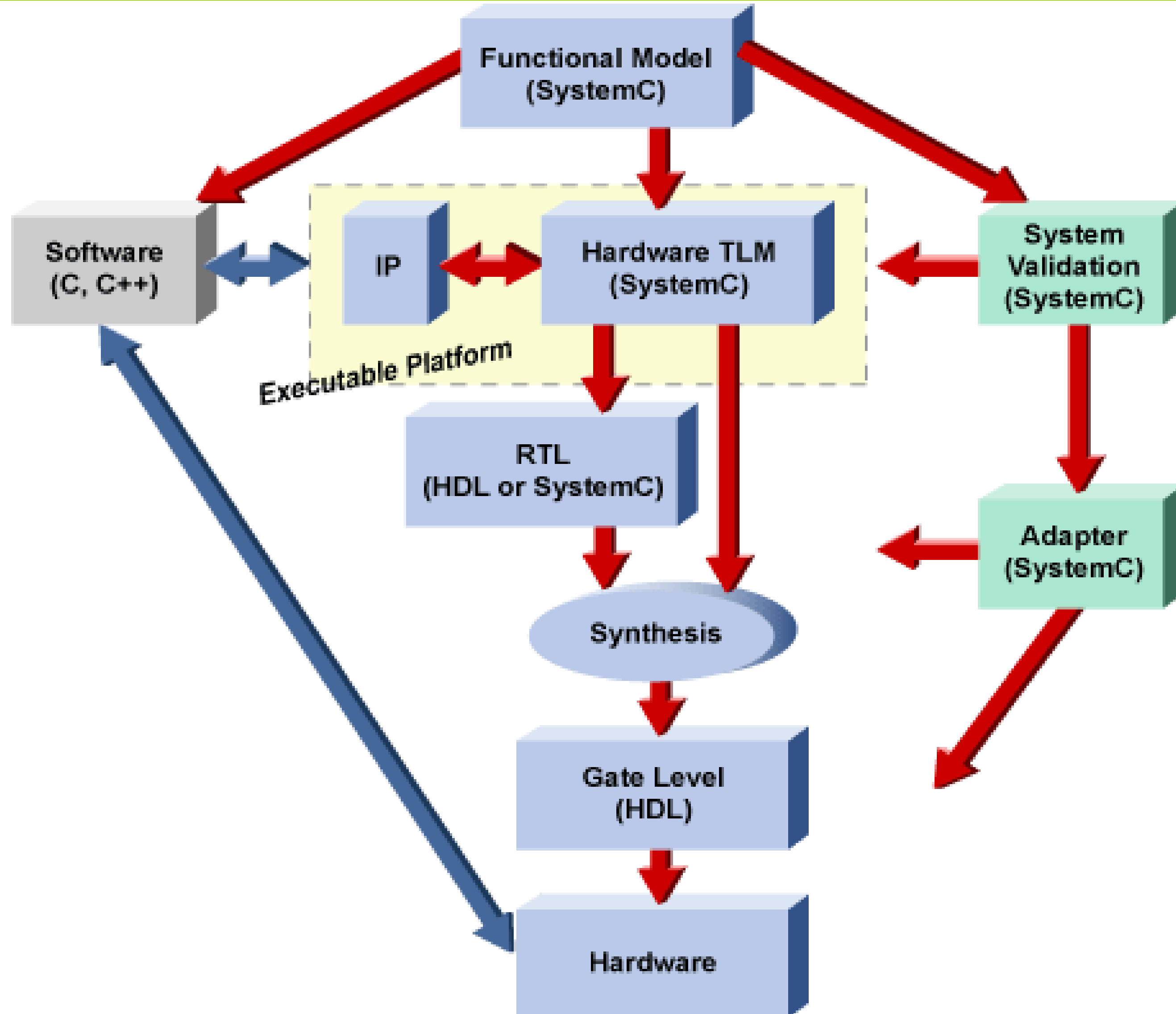
Model Types (2)

- Functional Model
  - Above TLM, i.e. System Architectural & System Performance
- System Level
  - Above RTL
- Behavioral Synthesis
  - Architectural Analysis & Implementation
  - I/F cycle accurate with pin level detail
  - Functionality TF and not cycle accurate
- Bus Functional Model (BFM)
  - Used for simulation (mainly of processors)
  - Not meant for synthesis
  - I/F pin cycle accurate
  - Transactions for functionality
- Register Transfer Level (RTL)
  - Verilog, VHDL
- Gate Level
  - not good in SystemC

Current Methodology



SystemC Methodology

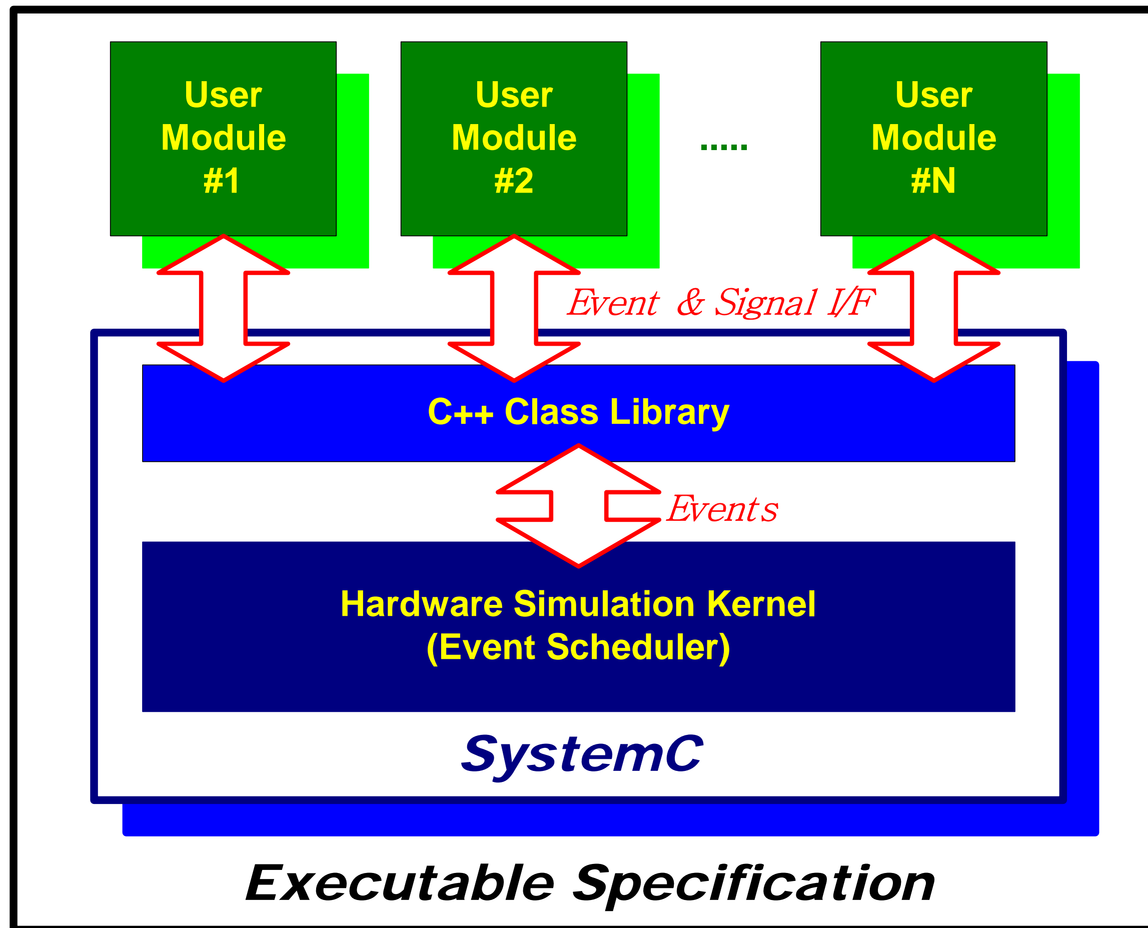


Using Executable Specifications

- **Ensure COMPLETENESS of Specification**

  "Create a program that Behave the same way as the system"
- **UNAMBIGUOUS Interpretation of the Specification**
- **Validate system functionality before implementation**
- **Create early model and Validate system performance**
- **Refine and Test the implementation of the Specification**

SystemC and User Module



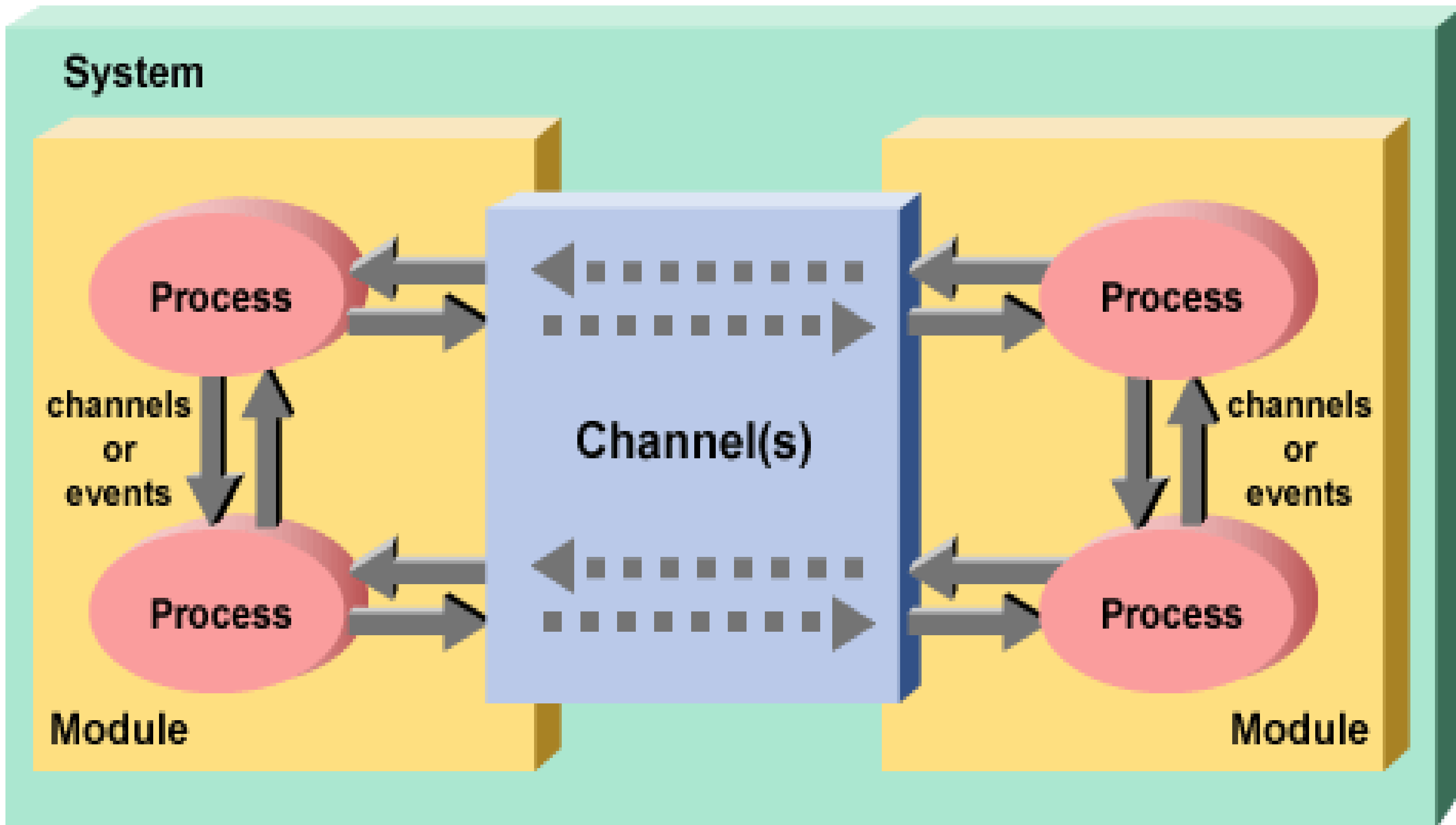
SystemC Highlights (1)

- SystemC 2.0 introduces general-purpose
  - Events
    - Flexible, low-level synchronization primitive
    - Used to construct other forms of synchronization
  - Channels
    - A container class for communication and synchronization
    - They implement one or more *interfaces*
  - Interfaces
    - Specify a set of access methods to the channel
- Other comm & sync models can be built based on the above primitives
  - Examples
    - HW-signals, queues (FIFO, LIFO, message queues, etc) semaphores, memories and busses (both at RTL and transaction-based models)

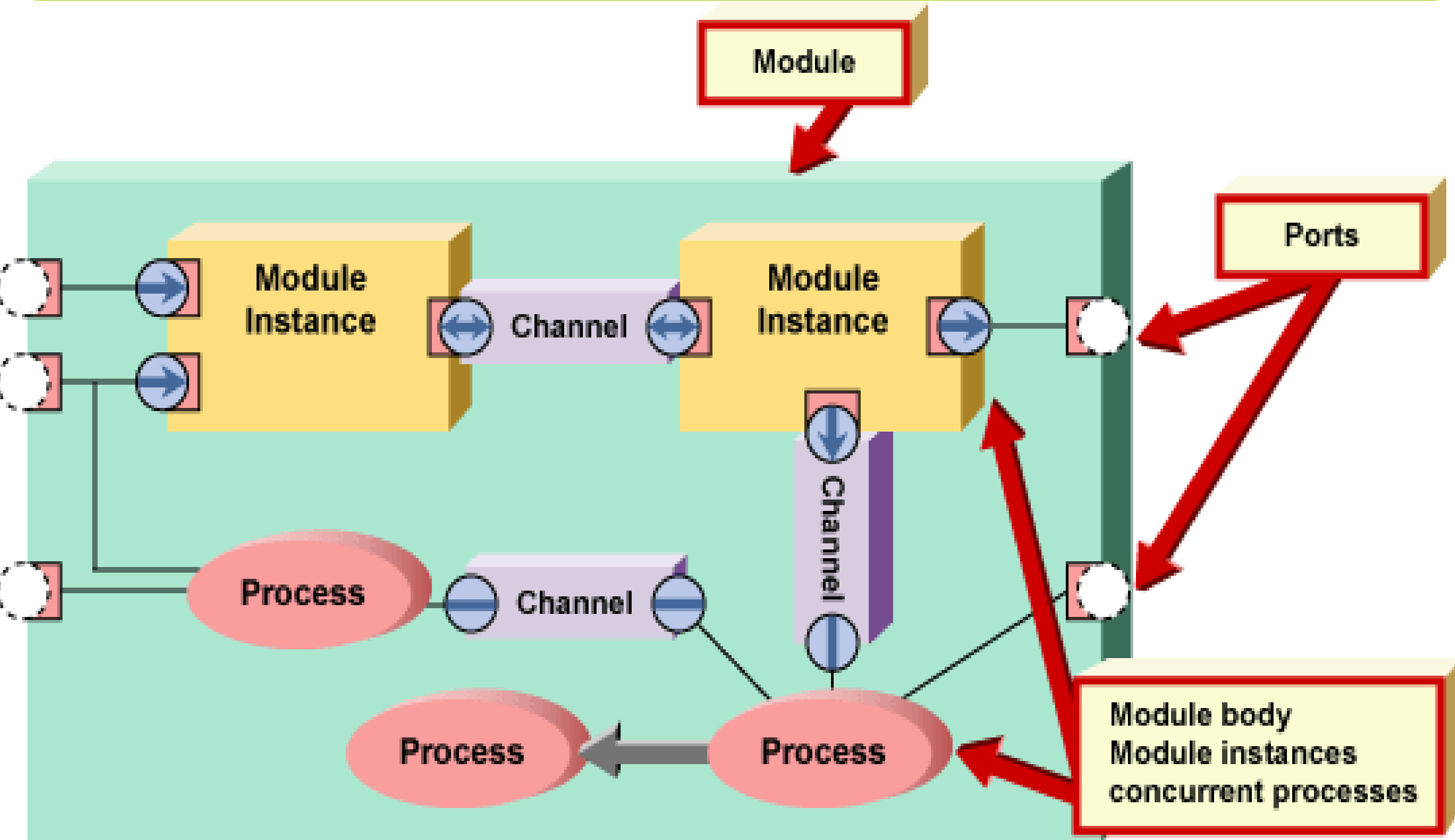
SystemC Highlights (2)

- **Support Hardware-Software Co-Design**
- **All constructs are in a C++ environment**
  - **Modules**
    - Container class includes hierarchical Modules and Processes
  - **Processes**
    - Describe functionality
    - Almost all SLDL have been developed based on some underlying model of network of processes
  - **Ports**
    - Single-directional(in, out), Bi-directional mode

A system in SystemC



A system in SystemC



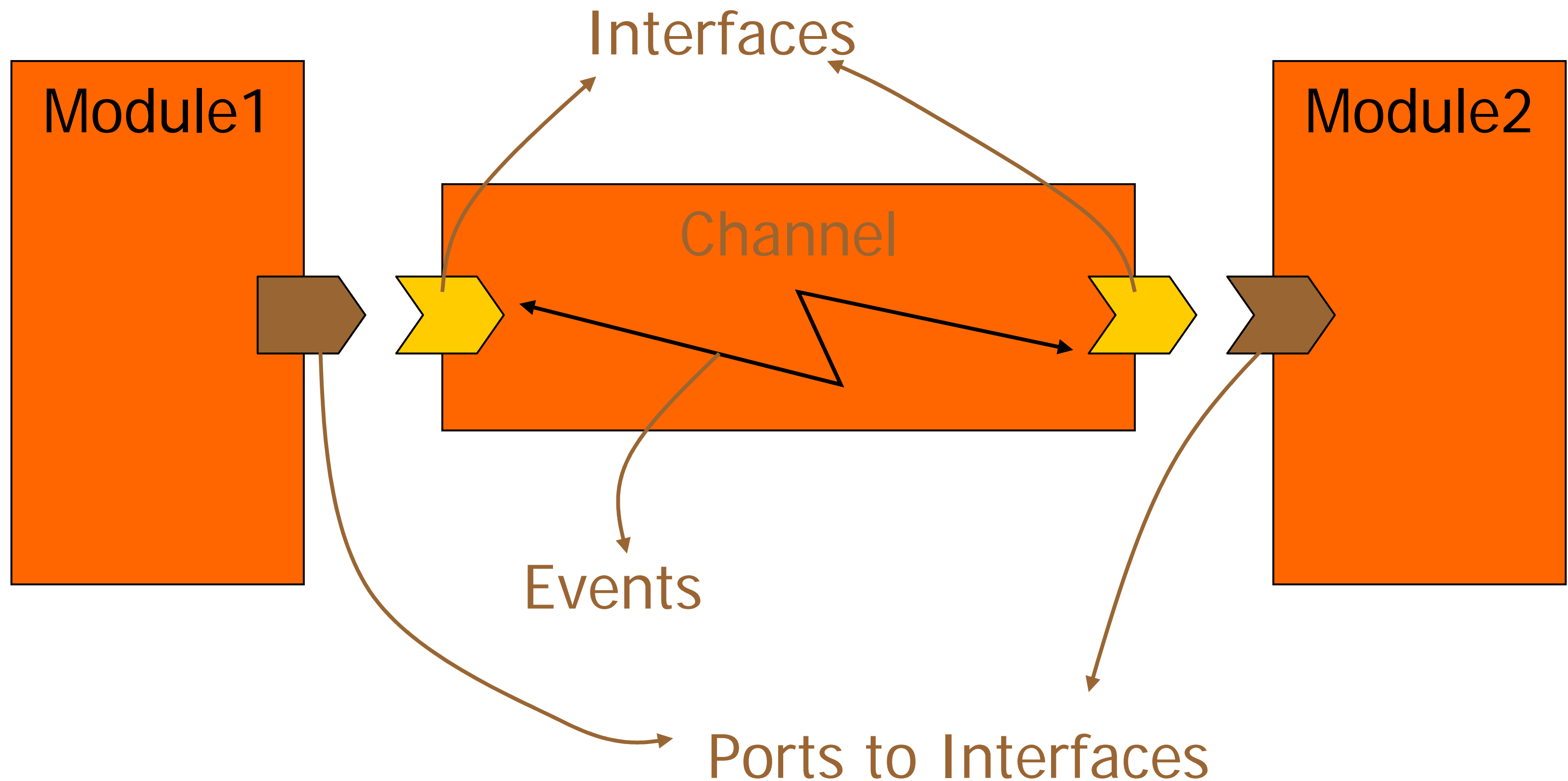
SystemC Highlights (3)

- **Constructs in a C++ environment (continued)**
  - **Clocks**
    - Special signal, Timekeeper of simulation and Multiple clocks, with arbitrary phase relationship
  - **Event Driven simulation**
    - High-SpeedEvent Driven simulation kernel
  - **Multiple abstraction levels**
    - Untimed from high-level functional model to detailed clock cycle accuracy RTL model
  - **Communication Protocols**
  - **Debugging Supports**
    - Run-Time error check
  - **Waveform Tracing**
    - Supports VCD, WIF, ISBD

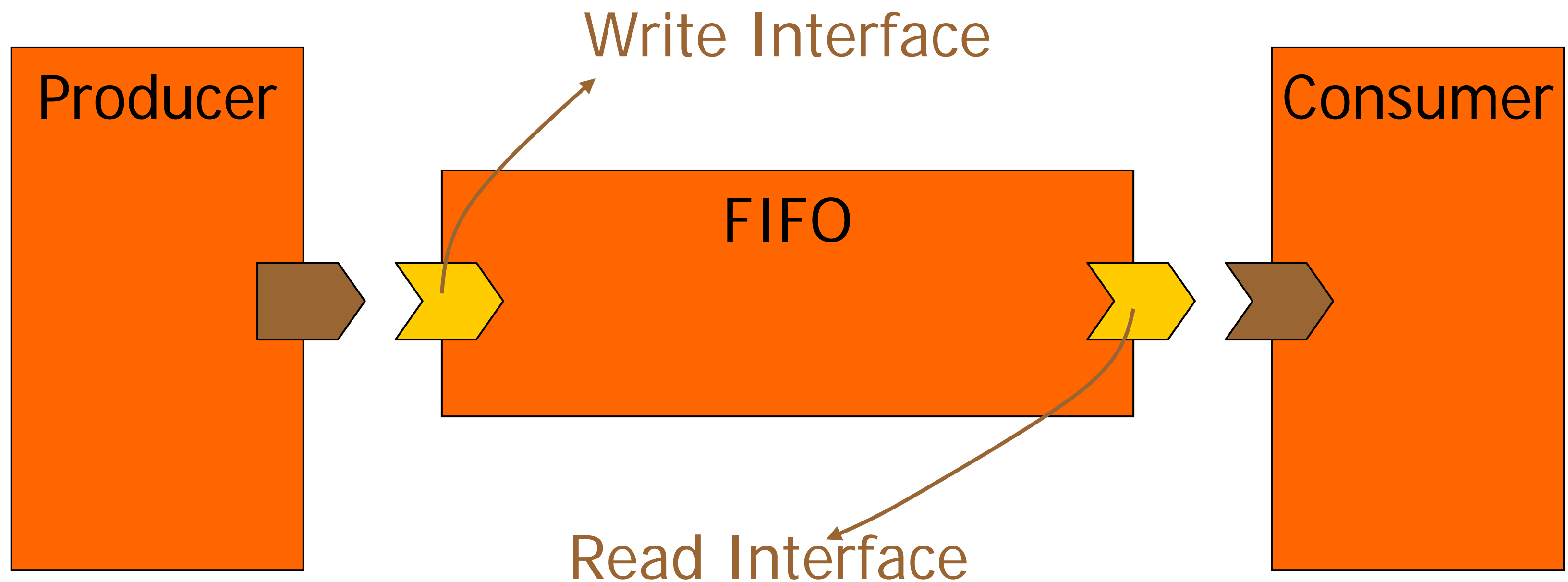
Data Types

- **SystemC supports**
  - **Native C/C++ Types**
  - **SystemC Types**
- **SystemC Types**
  - **Data type for system modeling**
  - **2 value** (‘0’,’1’) **logic/logic vector**
  - **4 value** (‘0’,’1’,’Z’,’X’) **logic/logic vector**
  - **Arbitrary sized integer (Signed/Unsigned)**
  - **Fixed Point types (Templated/Untemplated)**

Communication and Synchronization (cont'd)

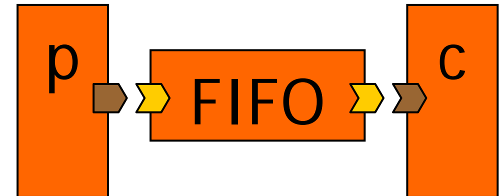


A Communication Modeling Example: FIFO



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

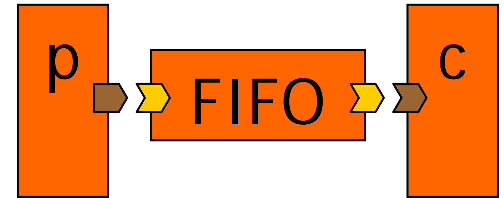
Completing the Comm. Modeling Example

```
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);
   };
```



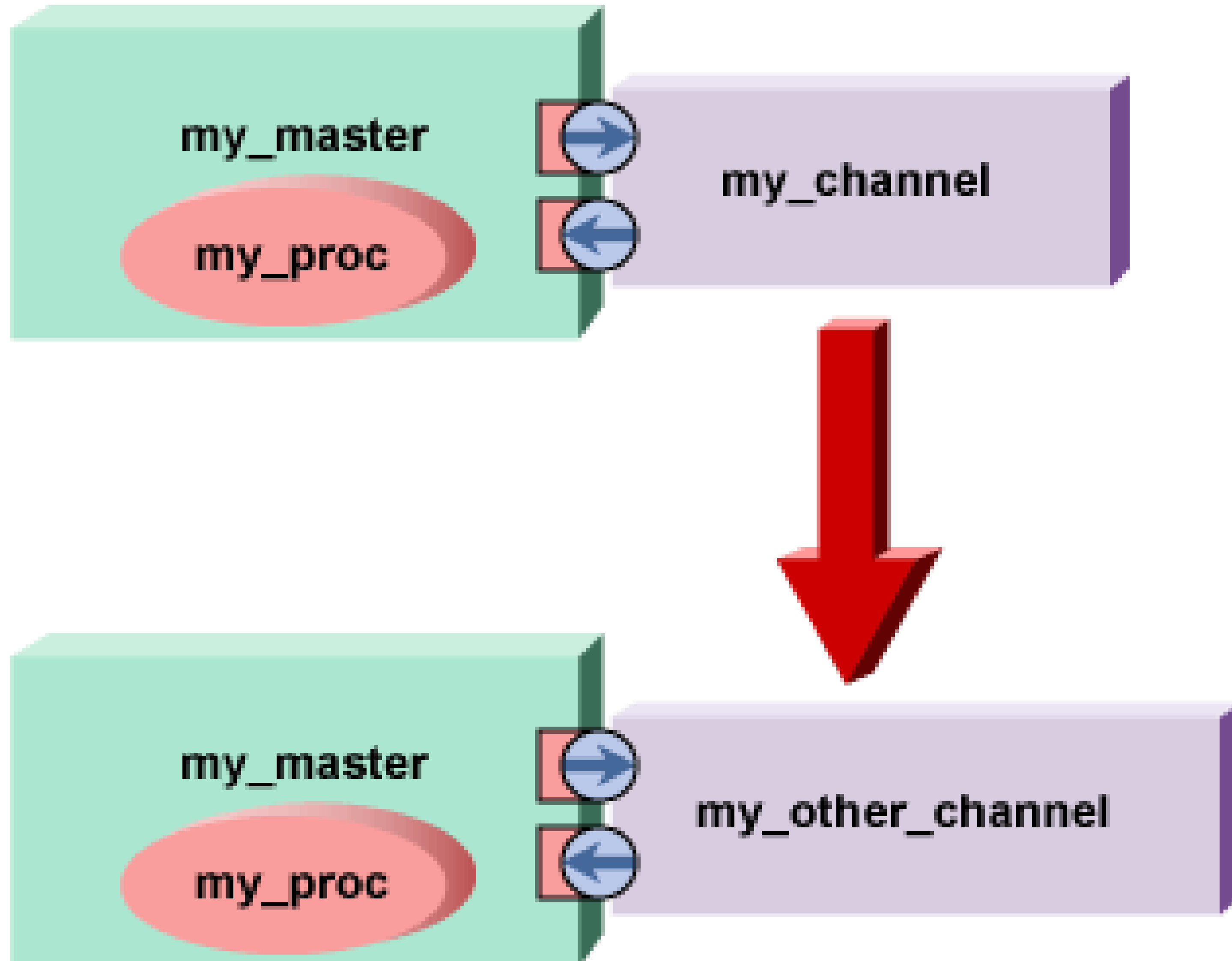
Completing the Comm. Modeling Example (cont'd)

- **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

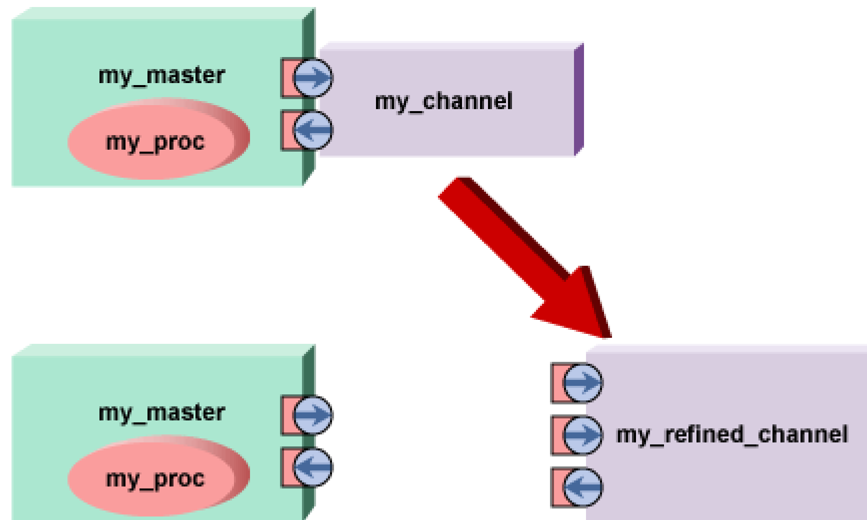
Completing the Comm. Modeling Example (cont'd)

- 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

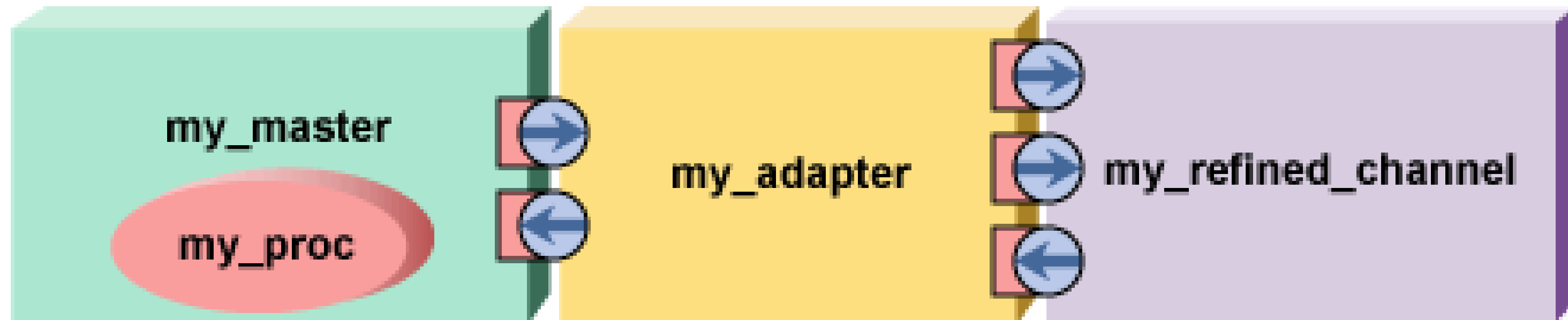
SystemC refinement



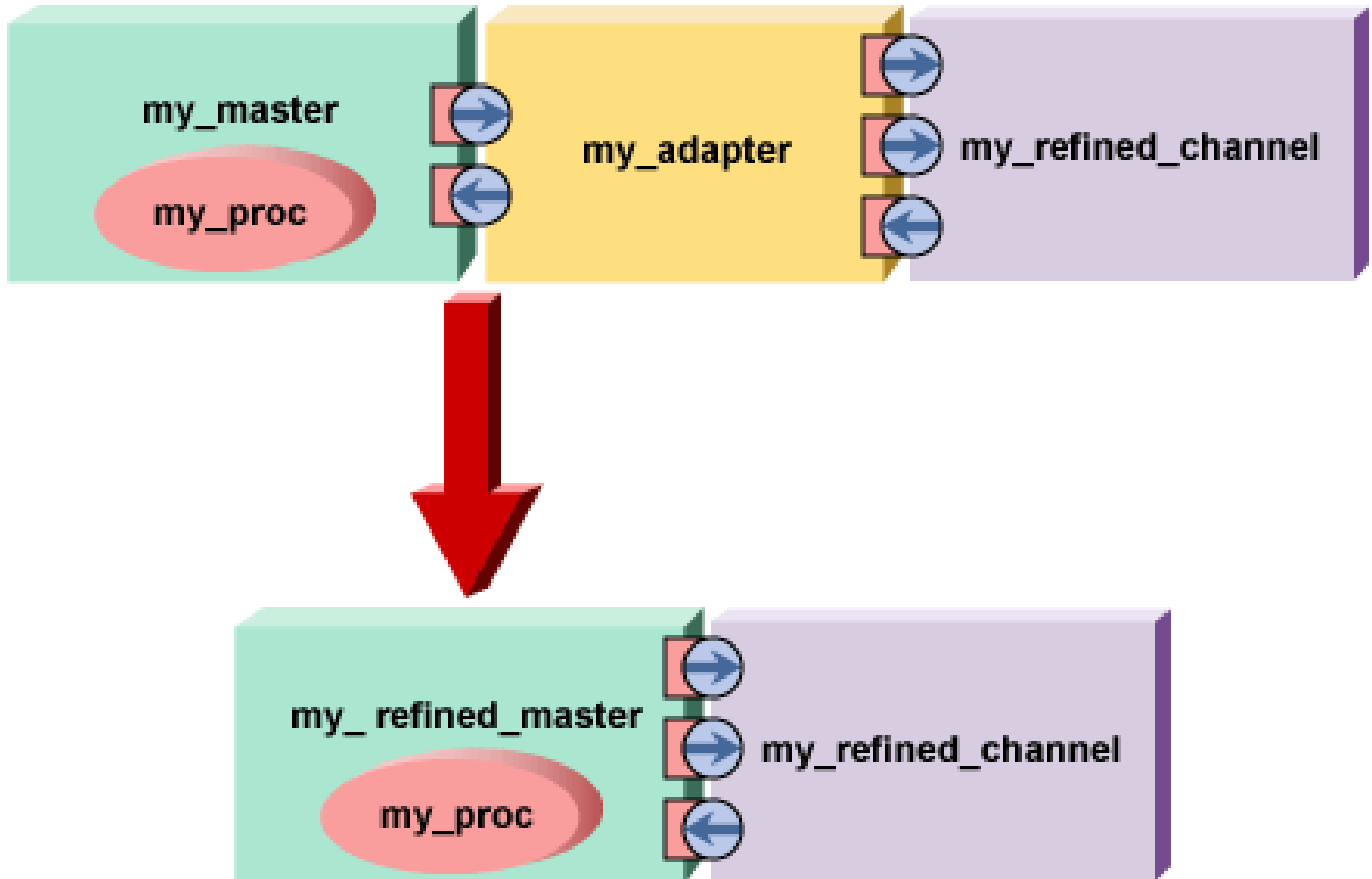
SystemC Channel Replacement



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

References

- www.doulos.com
- www.forteds.com