SystemC: Co-specification and System Modeling

Overview:

- Hardware-Software Co-Specification
- SystemC and Co-specification
- Introduction to SystemC for Co-specification
- A SystemC Primer

Hardware-Software Codesign

Co-design of Embedded Systems consists of the following parts:

- **Co-Specification**
  Developing system specification that describes hardware, software modules and relationship between the hardware and software
- **Co-Synthesis**
  Automatic and semi-automatic design of hardware and software modules to meet the specification
- **Co-Simulation and Co-verification**
  Simultaneous simulation of hardware and software

HW/SW Co-Specification

- Model the Embedded system functionality from an abstract level.
- No concept of hardware or software yet.
- Common environment
  SystemC: based on C++.
- Specification is analyzed to generate a task graph representation of the system functionality.

Co-Specification

- A system design language is needed to describe the functionality of both software and hardware.
- The system is first defined without making any assumptions about the implementation.
- A number of ways to define new specification standards grouped in three categories:

  - SystemC An open source library in C++ that provides a modeling platform for systems with hardware and software components.

SystemC for Co-specification

Open SystemC Initiative (OSCI) 1999 by EDA venders including Synopsys, ARM, CoWare, Fujitsu, etc.

- A C++ based modeling environment containing a class library and a standard ANSI C++ compiler.
- SystemC provides a C++ based modeling platform for exchange and co-design of system-level intellectual property (SoC-IP) models.

- SystemC is not an extension to C++
  - SystemC 1.0 and 2.1, 2.2 and 2.3 versions
  - It has a new C++ class library

SystemC Library Classes

SystemC classes enable the user to

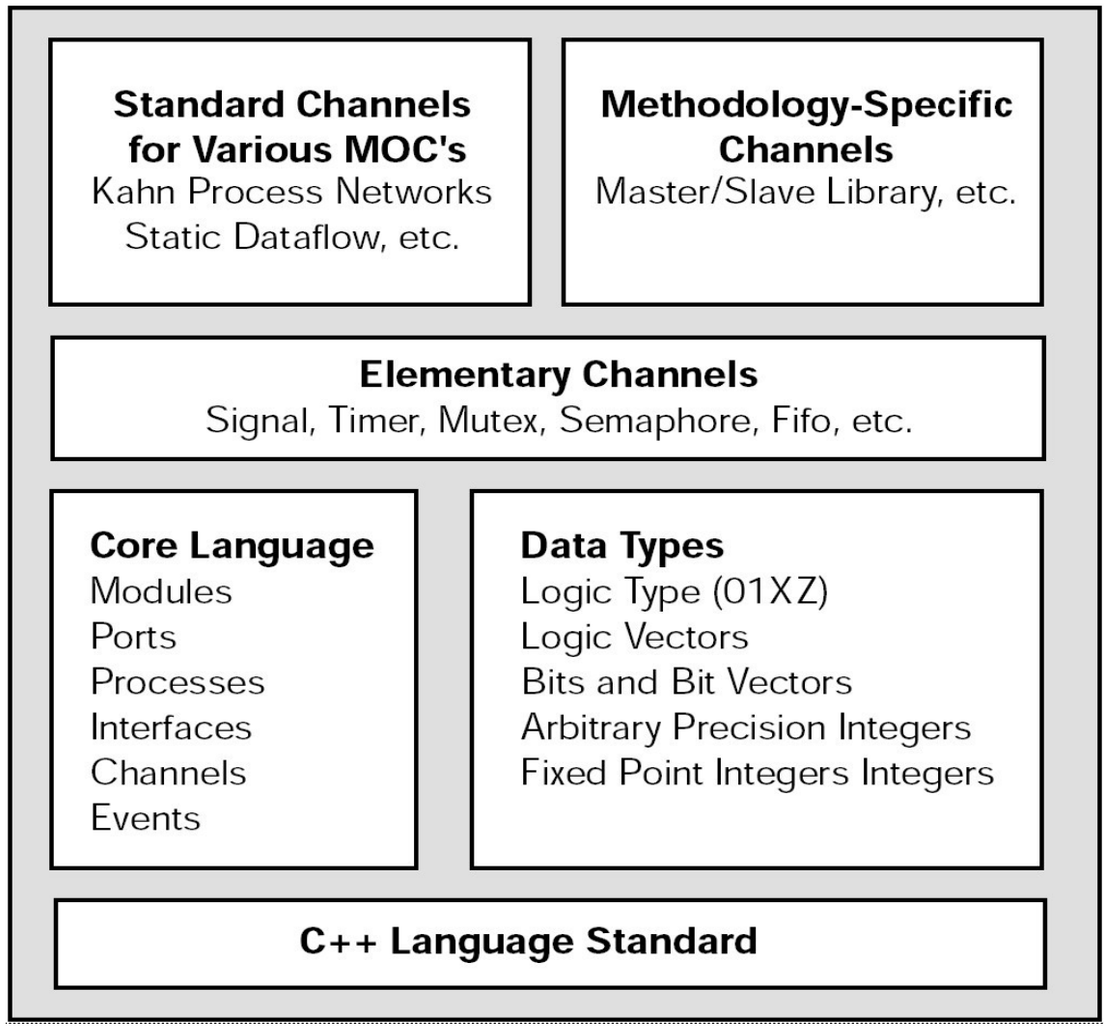
- Define modules and processes
- Add inter-process/module communication through ports and signals.

Modules/processes can handle a multitude of data types:
Ranging from bits to bit-vectors, standard C++ types to user define types like structures

Modules and processes also introduce timing, concurrency and reactive behavior.

- Using SystemC requires knowledge of C/C++

SystemC 2.0 Language Architecture



SystemC 2.0 Language Architecture

- All of SystemC builds on C++
- Upper layers are cleanly built on top of the lower layers
- The SystemC core language provides a minimal set of modeling constructs for structural description, concurrency, communication, and synchronization.
- Data types are separate from the core language and user-defined data types are fully supported.
- Commonly used communication mechanisms such as signals and FIFOs can be built on top of the core language. The MOCs can also be built on top of the core language.
- If desired, lower layers can be used without needing the upper layers.

SystemC Benefits

SystemC 2.x allows the following tasks to be performed within a single language:

- Complex system specifications can be developed and simulated
- System specifications can be refined to mixed software and hardware implementations
- Hardware implementations can be accurately modeled at all the levels.
- Complex data types can be easily modeled, and a flexible fixed-point numeric type is supported
- The extensive knowledge, infrastructure and code base built around C and C++ can be leveraged

SystemC for Co-Specification

Multiple abstraction levels:

- SystemC supports untimed models at different levels of abstraction,
  - ranging from high-level functional models to detailed clock cycle accurate RTL models.

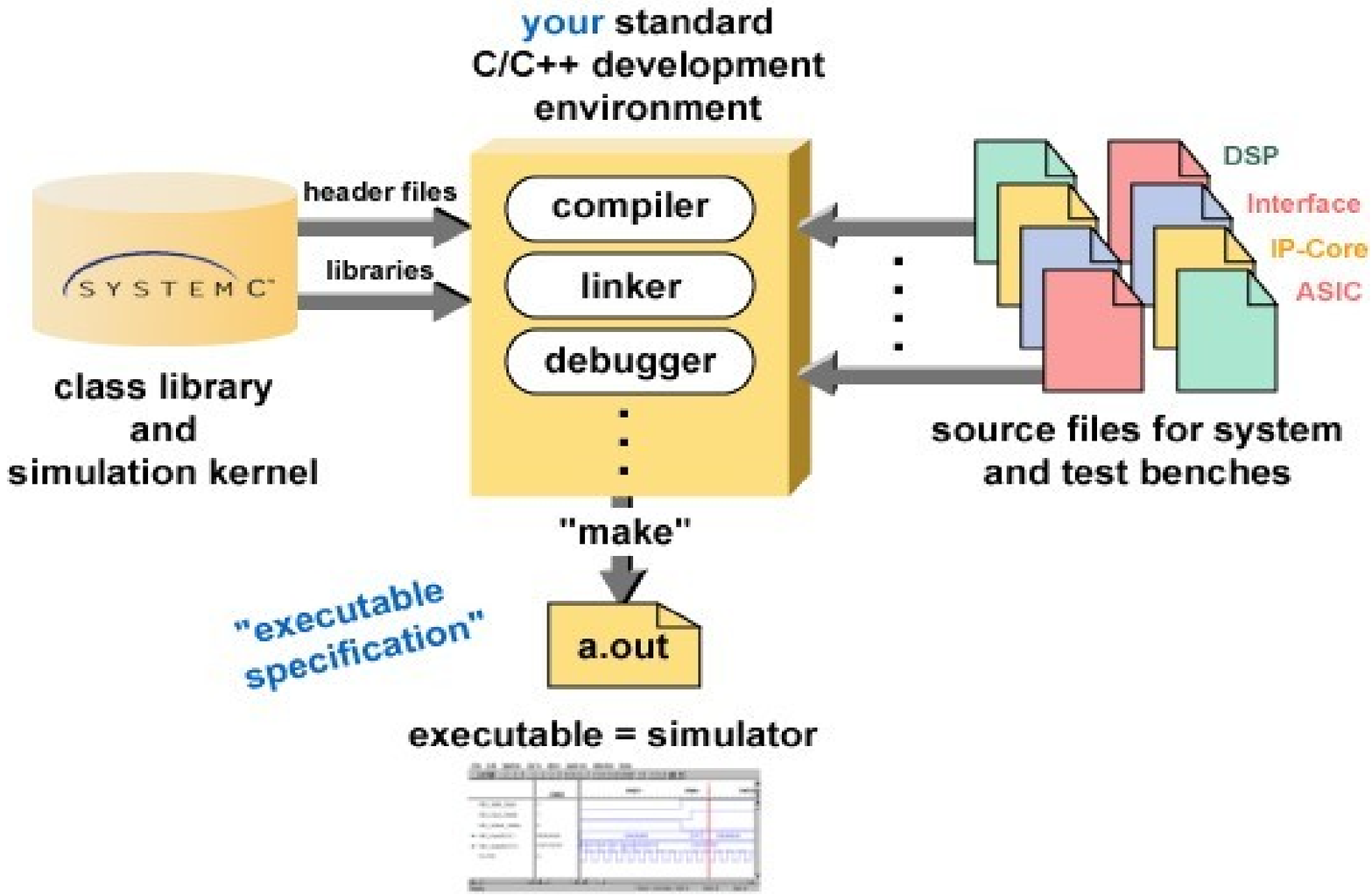
Communication protocols:

- SystemC provides multi-level communication semantics that enable you to describe the system I/O protocols at different levels of abstraction.

Waveform tracing:

- SystemC supports tracing of waveforms in VCD, WIF, and ISDB formats.

SystemC Development Environment



SystemC Features

Rich set of data types:

- to support multiple design domains and abstraction levels.
  - The fixed precision data types allow for fast simulation,
  - Arbitrary precision types can be used for computations with large numbers.
  - the fixed-point data types can be used for DSP applications.

Variety of port and signal types:

- To support modeling at different levels of abstraction, from the functional to the RTL.

Clocks:

- SystemC has the notion of clocks (as special signals).
- Multiple clocks, with arbitrary phase relationship, are supported.

Cycle-based simulation:

- SystemC includes an ultra light-weight cycle-based simulation kernel that allows high-speed simulation.

SystemC Data types

- SystemC supports:
  - all C/C++ native types
  - plus specific SystemC types

- SystemC types:
  - Types for systems modeling
  - 2 values (‘0’,’1’)
  - 4 values (‘0’,’1’,’Z’,’X’)
  - Arbitrary size integer (Signed/Unsigned)
  - Fixed point types

SC_Logic, SC_int types

SC_Logic: More general than bool, 4 values :

('0' (false), '1' (true), 'X' (undefined) , 'Z'(high-impedance))

Assignment like bool

```
my_logic = '0';
my_logic = 'Z';
```

Operators like bool but Simulation time bigger than bool

Declaration

```
sc_logic my_logic;
```

Fixed precision Integer: Used when arithmetic operations need fixed size arithmetic operands

- INT can be converted in UINT and vice-versa
- 1-64 bits integer in SystemC

```
sc_int<n>      -- signed integer with n-bits
sc_uint<n>     -- unsigned integer with n-bits
```

Other SystemC types

Bit Vector

- sc_bv<n>
  - 2-valued vector (0/1)
  - Not used in arithmetics operations
  - Faster simulation than sc_lv

Logic Vector

- sc_lv<n>
  - Vector of the 4-valued sc_logic type

Assignment operator (=)

- my_vector = “XZ01”
- Conversion between vector and integer (int or uint)
- Assignment between sc_bv and sc_lv

Additional Operators:

Reduction --	and_reduction()	or_reduction()	xor_ reduction()
Conversion --	to_string()		

SystemC Data types

Type	Description
sc_logic	Simple bit with 4 values(0/1/X/Z)
sc_int	Signed Integer from 1-64 bits
sc_uint	Unsigned Integer from 1-64 bits
sc_bigint	Arbitrary size signed integer
sc_biguint	Arbitrary size unsigned integer
sc_bv	Arbitrary size 2-values vector
sc_lv	Arbitrary size 4-values vector
sc_fixed	templated signed fixed point
sc_ufixed	templated unsigned fixed point
sc_fix	untemplated signed fixed point
sc_ufix	untemplated unsigned fixed point

SystemC types

Operators of fixed precision types

Bitwise	~	&	|	^	>>	<<			
Arithmetics	+	-	*	/	%				
Assignement	=	+=	-=	*=	/=	%=	&=	|=	^=
Equality	==	!=							
Relational	<	<=	>	> =					
Auto-Inc/Dec	++	--							

Bit selection	[x]	e.g. mybit = myint[7]
Part select	range()	e.g. myrange = myint.range(7,4)
Concatenation	(,)	e.g. intc = (inta, intb);

Usage of SystemC types

```
sc_bit y; sc_bv<8> x;
y = x[6];

sc_bv<16> x; sc_bv<8> y;
y = x.range(0,7);

sc_bv<64> databus; sc_logic result;
result = databus.or_reduce();
```

```
sc_lv<32> bus2;
cout << “bus = “ << bus2.to_string();
```

SystemC Specific Features

- Modules:
  - A class called a module: A hierarchical entity that can have other modules or processes contained in it.
- Ports:
  - Modules have ports through which they connect to other modules.
  - Single-direction and bidirectional ports.
- Signals:
  - SystemC supports resolved and unresolved signals.
- Processes:
  - used to describe functionality.
  - contained inside modules.

Modules

The basic building block in SystemC to partition a design.

- Modules are similar to „entity“ in VHDL
- Modules allow designers to hide internal data representation and algorithms from other modules.

Declaration

- Using the macro SC_MODULE

  SC_MODULE(modulename) {
- Using typical C++ struct or class declaration:

  struct modulename : sc_module {

Elements:

Ports, local signals, local data, other modules, processes, and constructors

SystemC Constructor

Constructor: Each module should include a constructor that identifies processes as methods using the SC_METHOD macro.

SC_METHOD (funct) ; Identifies the function or process funct

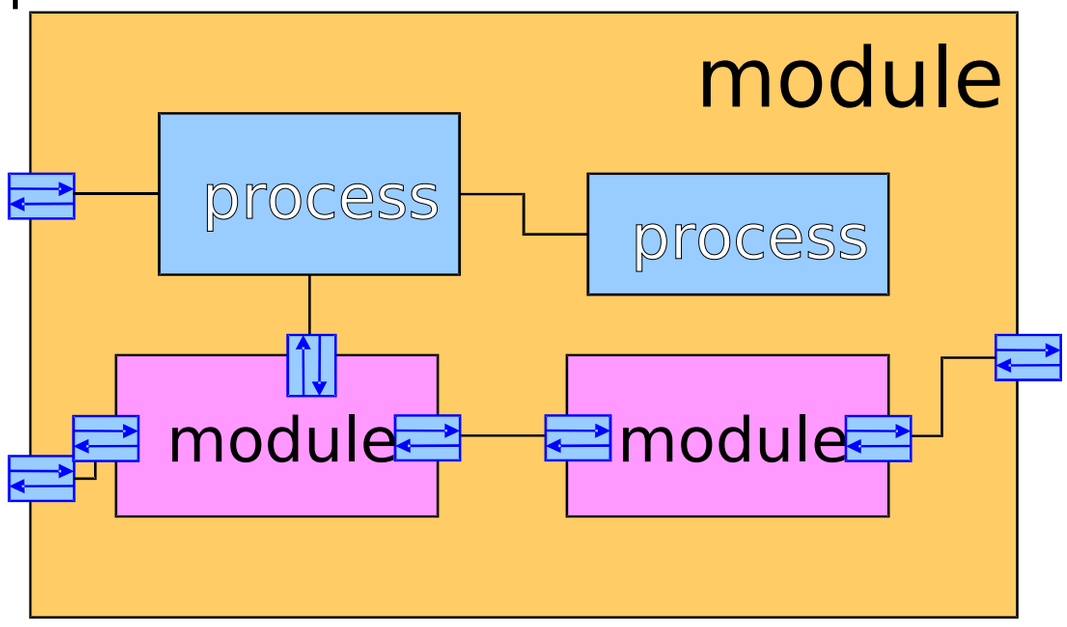
Methods are called similar to C++ as:

function_type module_name::function_name(data_type var_name) { ... }

- SC_METHOD process is triggered by events and executes all the statements in it before returning control to the SystemC kernel.
- A Method needs to be made sensitive to some internal or external signal. e.g. sensitive_pos << clock or sensitive_neg << clock
- Process and threads get executed automatically in the constrcutor even if an event in sensitivity list does not occur. To prevent this un-intentional execution, dont_initialize() function is used.

SystemC Module

```
SC_MODULE( module_name ){
 // Ports declaration
 // Signals declaration
 // Module constructor : SC_CTOR
 // Process constructors and sensibility list
 //          SC_METHOD
 // Sub-Modules creation and port mappings
 // Signals initialization
}
```



Signals and Ports

Ports of a module are the external interfaces that pass information to and from a module.

sc_inout<data_type> port_name;

- Create an input-output port of 'data_type' with name 'port_name'.
- sc_in and sc_out create input and output ports respectively.

Signals are used to connect module ports allowing modules to communicate.

sc_signal<data_type> sig_name ;

- Create a signal of type 'data_type' and name it 'sig_name'.
- hardware module has its own input and output ports to which these signals are mapped or bound.

For example:

```
in_tmp = in.read( );
out.write(out_temp);
```

2-to-1 Mux Modules

Module constructor – SC_CTOR is Similar to an "architecture" in VHDL

```
SC_MODULE( Mux21 ) {
    sc_in< sc_uint<8> >   in1;
    sc_in< sc_uint<8> >   in2;
    sc_in< bool >         selection;
    sc_out< sc_uint<8> >  out;

    void MuxImplement( void );
    SC_CTOR( Mux21 ) {
        SC_METHOD( MuxImplement );
        sensitive << selection;
        sensitive << in1;
        sensitive << in2;
    }
}
```

SystemC Counter Code

```
struct counter : sc_module {    // the counter module
   sc_inout<int> in; // the input/output port of int type
   sc_in<bool> clk; // Boolean input port for clock
   void counter_fn(); // counter module function
   SC_CTOR( counter ) {
         SC_METHOD( counter_fn ); // declare the counter_fn as a method
         dont_initialize(); // don’t run it at first execution
         sensitive_pos << clk; // make it sensitive to +ve clock edge
   }
}

// software block that check/reset the counter value, part of sc_main
   void check_for_10(int *counted) {
         if (*counted == 10) {
           printf(“Max count (10) reached ... Reset count to Zero\n");
           *counted = 0;
         }
   }
```

BCD Counter Example Main Code

```
void check_for_10(int *counted);
 int sc_main(int argc, char *argv[ ]) {
    sc_signal<int> counting; // the signal for the counting variable
    sc_clock clock("clock",20, 0.5); // clock period = 20 duty cycle = 50%
    int counted; // internal variable, to store the value in counting signal
    counting.write(0); // reset the counting signal to zero at start
    counter COUNT("counter"); // call counter module
    COUNT.in(counting); // map the ports by name
    COUNT.clk(clock);     // map the ports by name
    for (unsigned char i = 0; i < 21; i++) {
            counted = counting.read(); // copy the signal onto the variable
            check_for_10(&counted); // call the software block & check for 10
            counting.write(counted); // copy the variable onto the signal
            sc_start(20);                // run the clock for one period
    } return 0;
}
```

Counter Main Code with Tracing

```
int sc_main(int argc, char *argv[]) {
   sc_signal<int> counting; // the signal for the counting variable
   sc_clock clock("clock", 20, 0.5); // clock; time period = 20 duty cycle = 50%
   int counted; // internal variable, to stores the value in counting signal
                  // create the trace- file by the name of "counter_tracefile.vcd“
  sc_trace_file *tf = sc_create_vcd_trace_file("counter_tracefile");
                 // trace the clock and the counting signals
   sc_trace(tf, clock.signal(), "clock");
   sc_trace(tf, counting, "counting");
   counting.write(0); // reset the counting signal to zero at start
   counter COUNT("counter"); // call counter module. COUNT is just a temp var
   COUNT.in(counting); // map the ports by name
   COUNT.clk(clock);     // map the ports by name
   for (unsigned char i = 0; i < 21; i++) {
          .....
    }
   sc_close_vcd_trace_file(tf); // close the tracefile
 return 0;
}
```

SystemC Counter Module

```
#include "systemc.h"
#define COUNTER
struct counter : sc_module {    // the counter module
   sc_inout<int> in; // the input/output port of int type
   sc_in<bool> clk; // Boolean input port for clock
   void counter_fn(); // counter module function
   SC_CTOR( counter ) { // counter constructor
    SC_METHOD( counter_fn ); // declare the counter_fn as a method
    dont_initialize(); // don't run it at first execution
    sensitive_pos << clk; // make it sensitive to +ve clock edge
   }
  } ;
   void counter :: counter_fn() {
      in.write(in.read() + 1);
      printf("in=%d\n", in.read());
  }
```

Module Instantiation

- Instantiate module

  Module_type Inst_module (“label”);

- Instantiate module as a pointer

  Module_type *pInst_module;

  // Instantiate at the module constructor SC_CTOR

  pInst_module = new module_type (“label”);

  Inst_module.a(s);
  Inst_module.b(c);
  Inst_module.q(q);

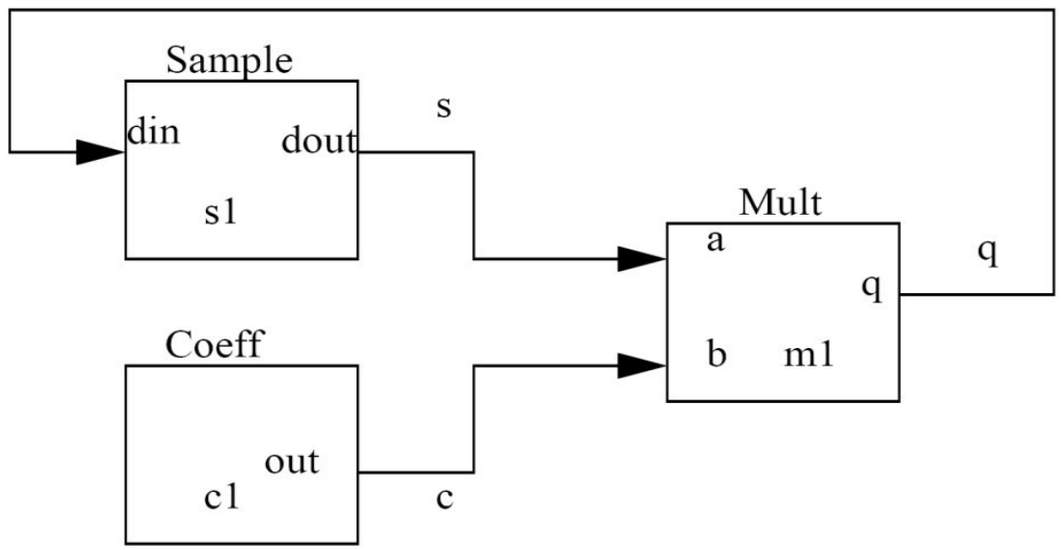
  pInst_module -> a(s);
  pInst_module -> b(c);
  pInst_module -> q(q);

Sub-module Connections

Signals

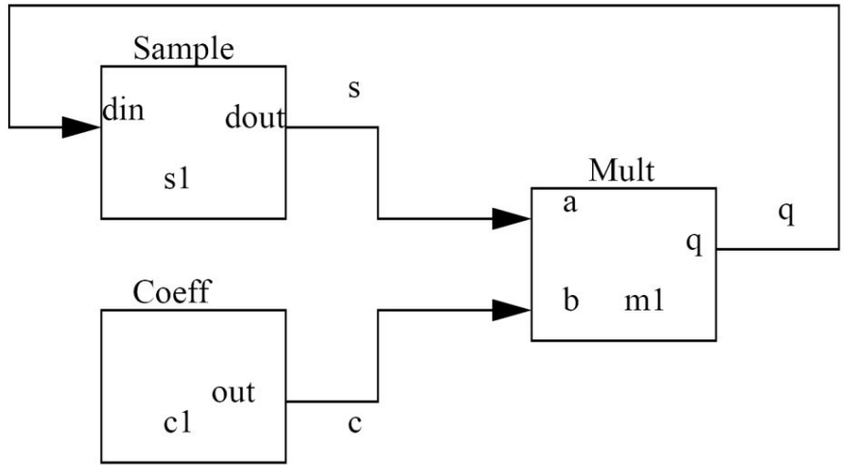
sc_signal<type > q, s, c;

- Positional Connection
- Named Connection



Named and Positional Connections

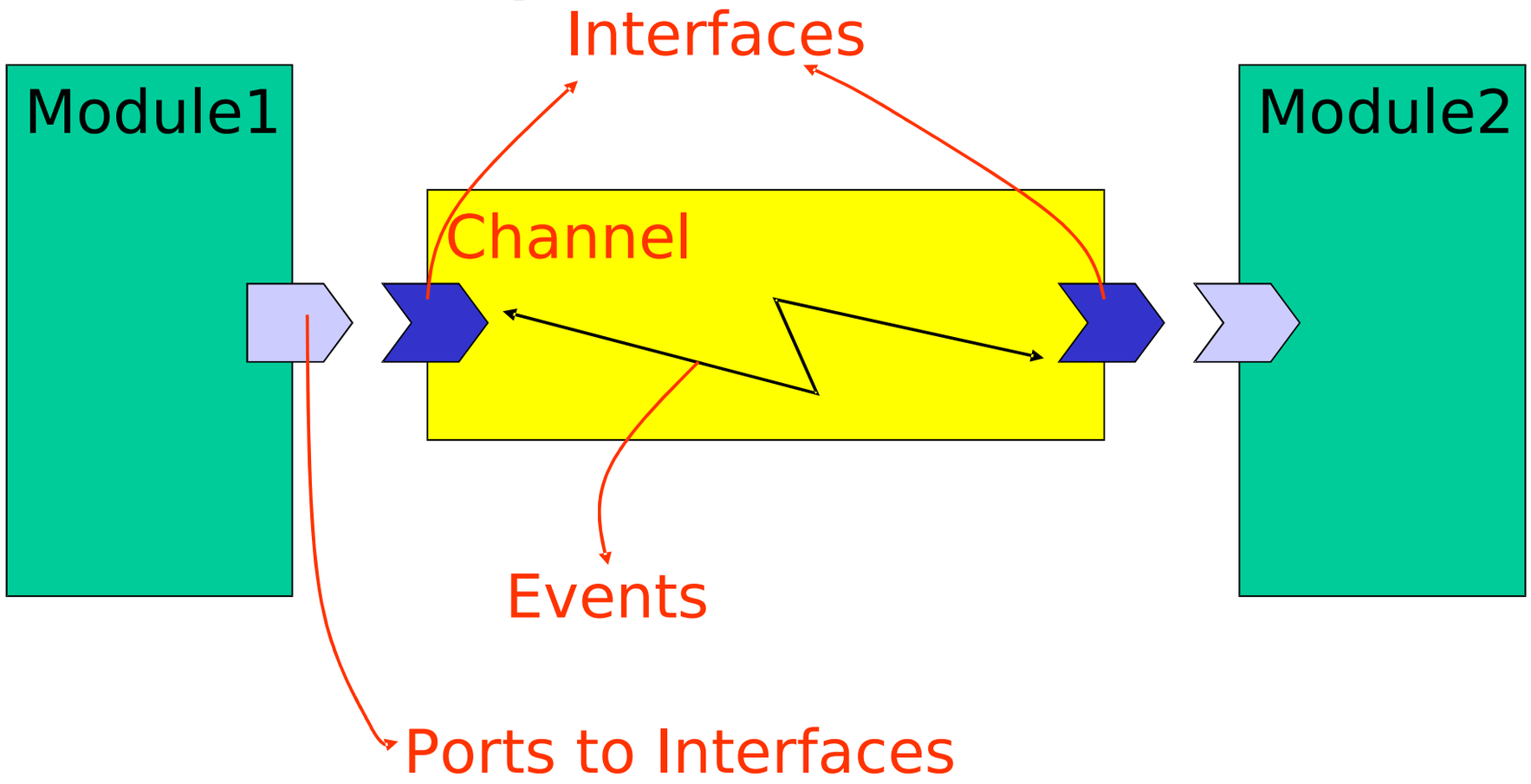
```
SC_MODULE(filter) {
   // Sub-modules: “components
   sample *s1;
   coeff     *c1;
   mult      *m1;
   sc_signal<sc_uint <32> > q,s,c;
    // Constructor :“architecture”
    SC_CTOR(filter) {
    //Sub-modules instantiation/mapping
        s1 = new sample (“s1”);
        s1->din(q);   // named mapping
        s1->dout(s);
        c1 = new coeff(“c1”);
        c1->out(c);   // named mapping
        m1 = new mult (“m1”);
        (*m1)(s, c, q)//positional mapping
    }
}
```



Communication and Synchronization

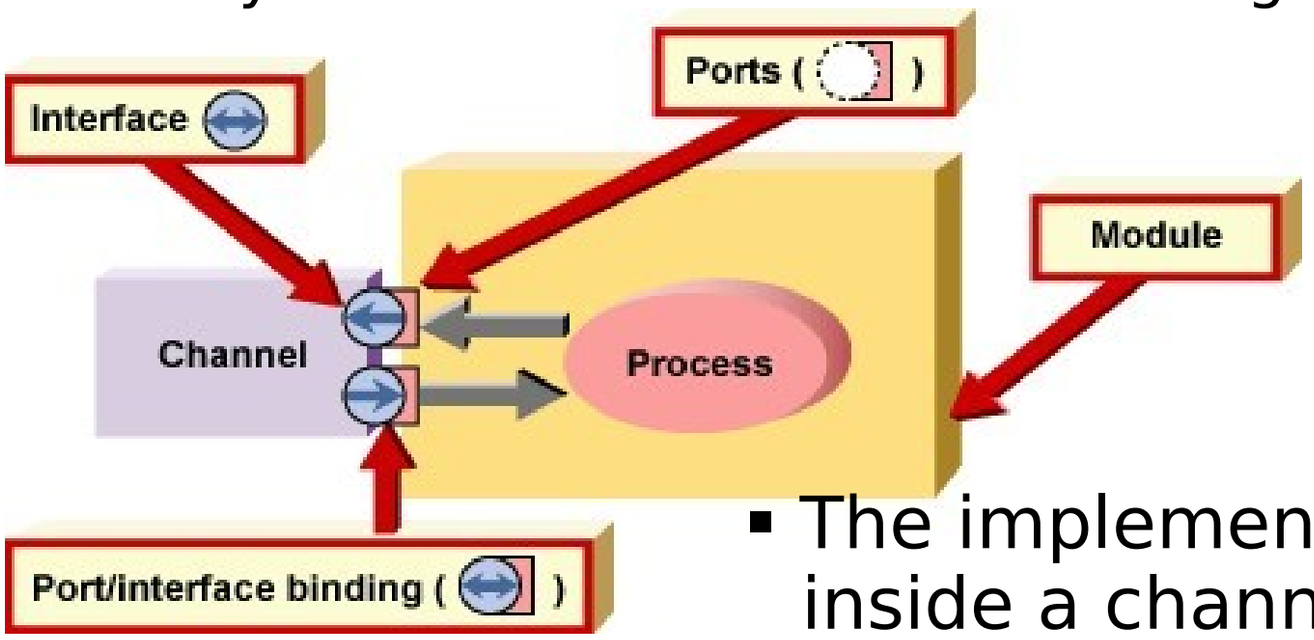
- SystemC 2.0 and higher has general-purpose
  - Channel
    - A mechanism for communication and synchronization
    - They implement one or more interfaces
  - Interface
    - Specify a set of access methods to the channel
      But it does not implement those methods
  - Event
    - Flexible, low-level synchronization primitive
    - Used to construct other forms of synchronization

Communication and Synchronization



Interfaces

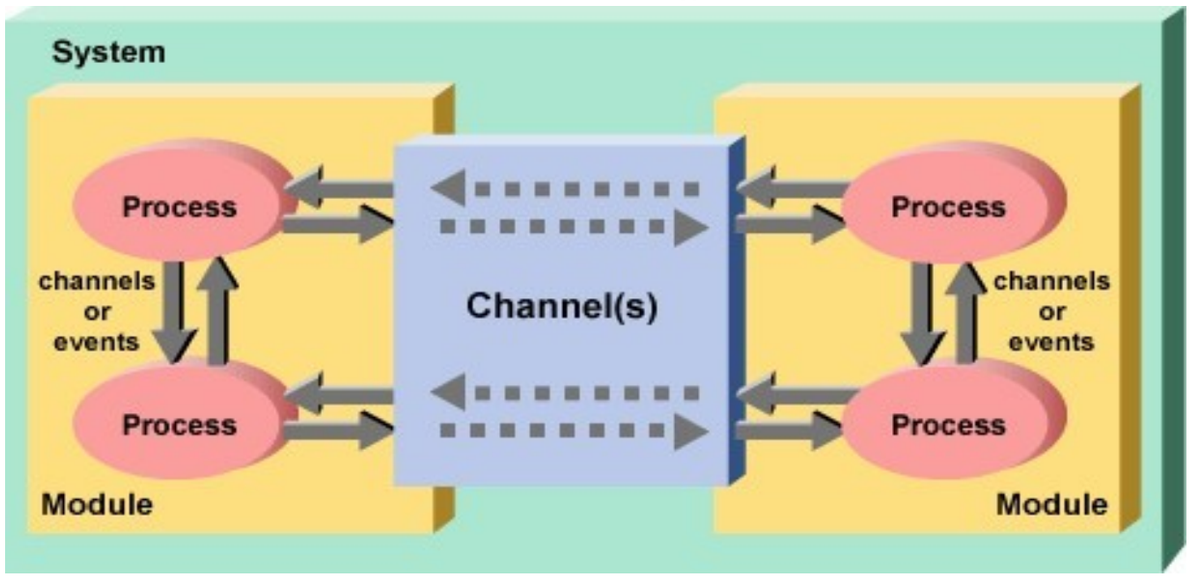
- Interface is purely functional and does not provide the implementation of the methods.
  - Interface only provides the method's signature.
- Interfaces are bound to ports.
  - They define what can be done through a particular port.



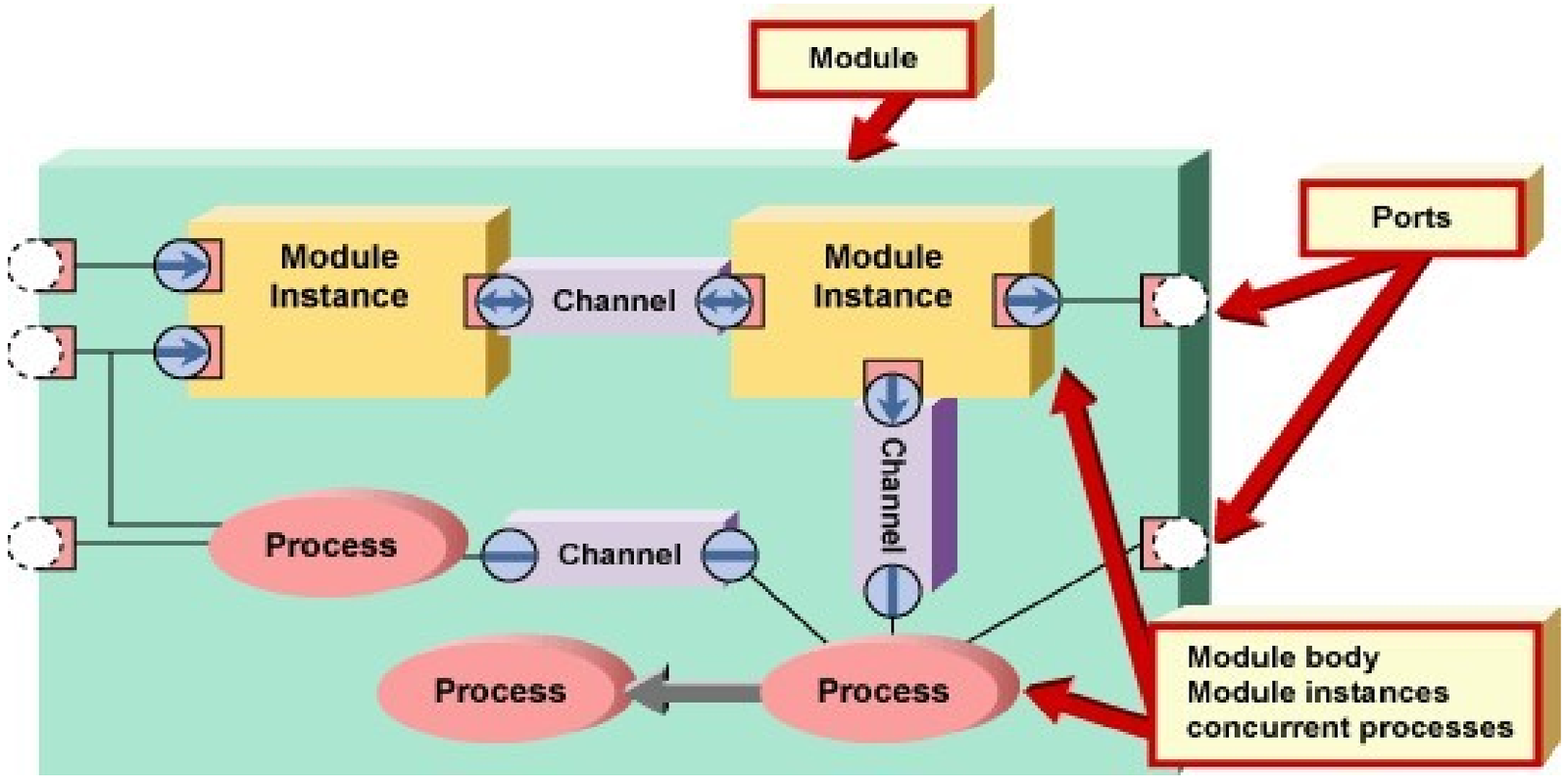
- The implementation is done inside a channel.

Channels

- Channel implements an interface
  - It must implement all of its defined methods.
- Channel are used for communication between processes inside of modules and between modules.
- Inside of a module a process may directly access a channel.
- If a channel is connected to a port of a module, the process accesses the channel through the port.



Channels


Module
Module Instance
Channel
Module Instance
Ports
Channel
Process
Channel
Process
Process
Module body
Module instances
concurrent processes

Channels

Two types of Channels: Primitive and Hierarchical

- Primitive Channels:
  - They have no visible structure and no processes
  - Theye cannot directly access other primitive channels.
    - o sc_signal
    - o sc_signal_rv
    - o sc_fifo
    - o sc_mutex
    - o sc_semaphore
    - o sc_buffer
- Hierarchical Channels:
  - These are modules themselves,
  - may contain processes, other modules etc.
  - may directly access other hierarchical channels.

Channel Usage

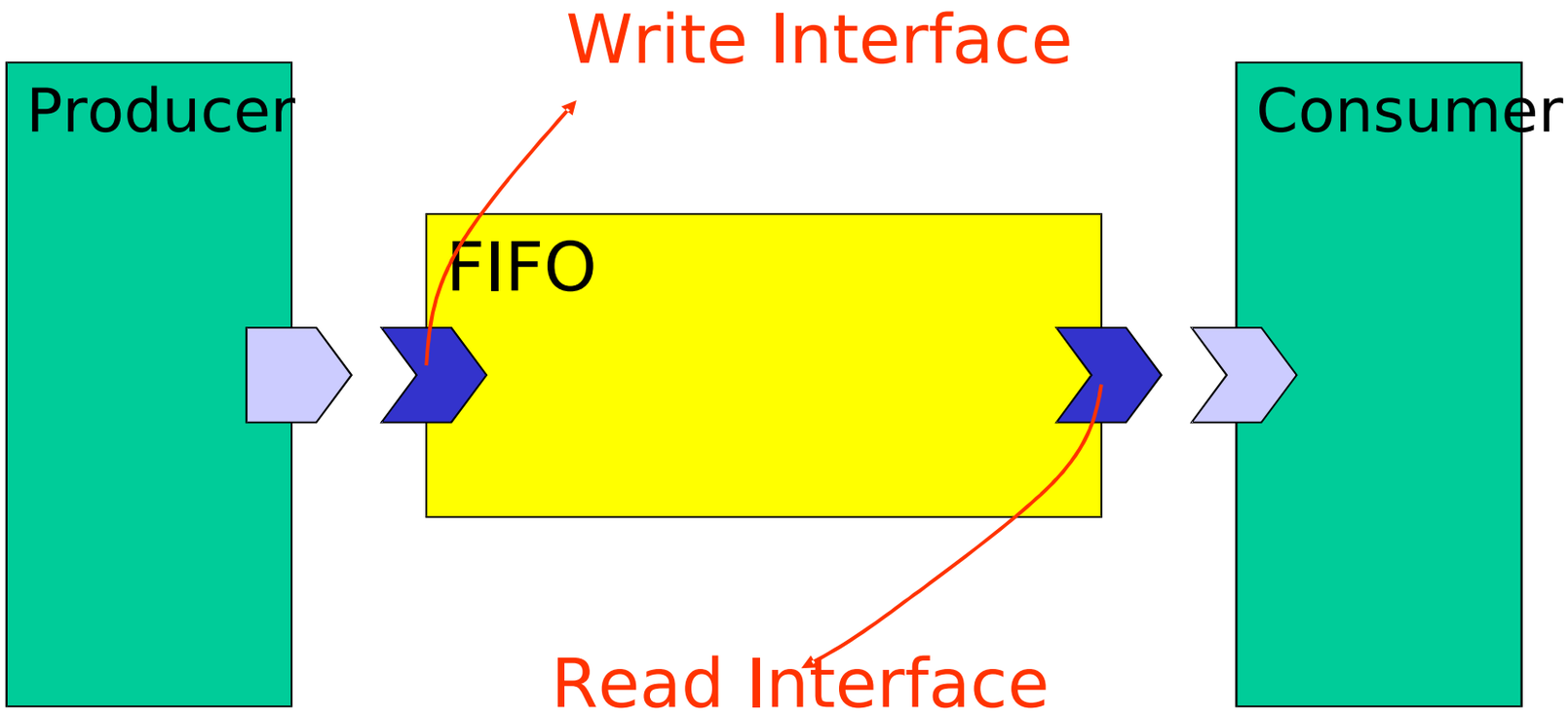
Use Primitive Channels:

- when you need to use the request-update semantics.
- when channels are atomic and cannot reasonably be chopped into smaller pieces.
- when speed is absolutely crucial.
  - Using primitive channels can often reduce the number of delta cycles.
- when it doesn't make any sense i.e. trying to build a channel out of processes and other channels such as a semaphore or a mutex.

Use Hierarchical Channels:

- when you would want to be able to explore the underlying structure,
- when channels contain processes or ports,
- when channels contain other channels.

A Communication Modeling FIFO Example



Problem definition: FIFO communication channel with blocking read and write operation
Source available in SystemC installation, under “examples\systemc” subdirectory