

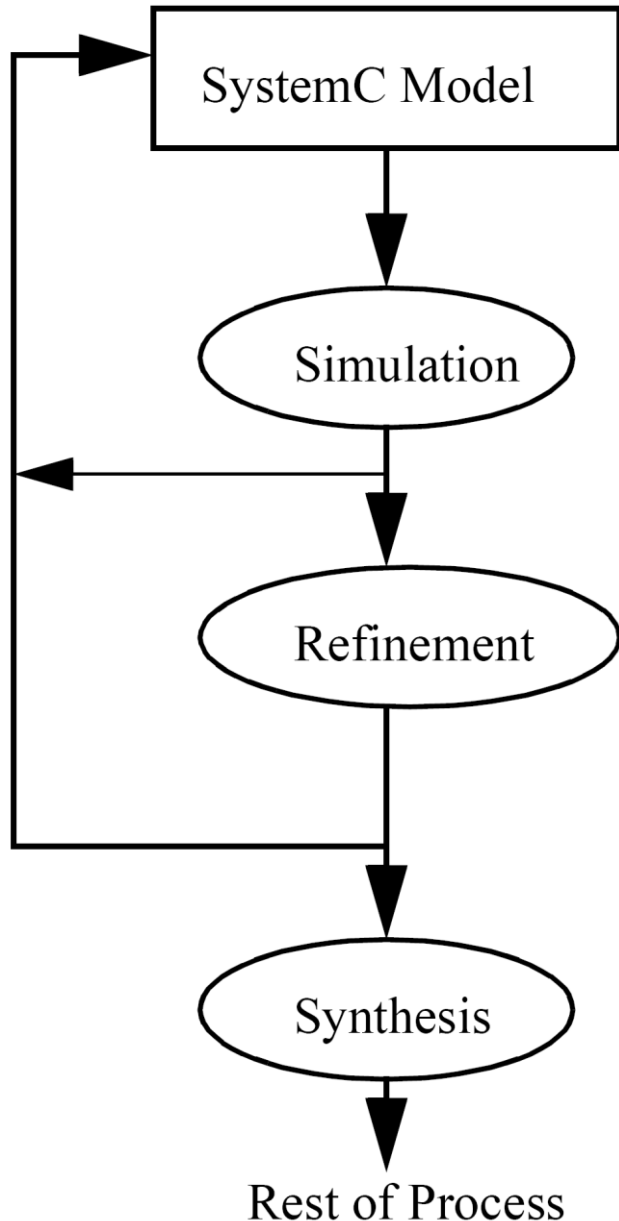


Projektiranje programabilnih SoC platformi
ID 222683

SYSTEMC SYSTEM-LEVEL MODELING



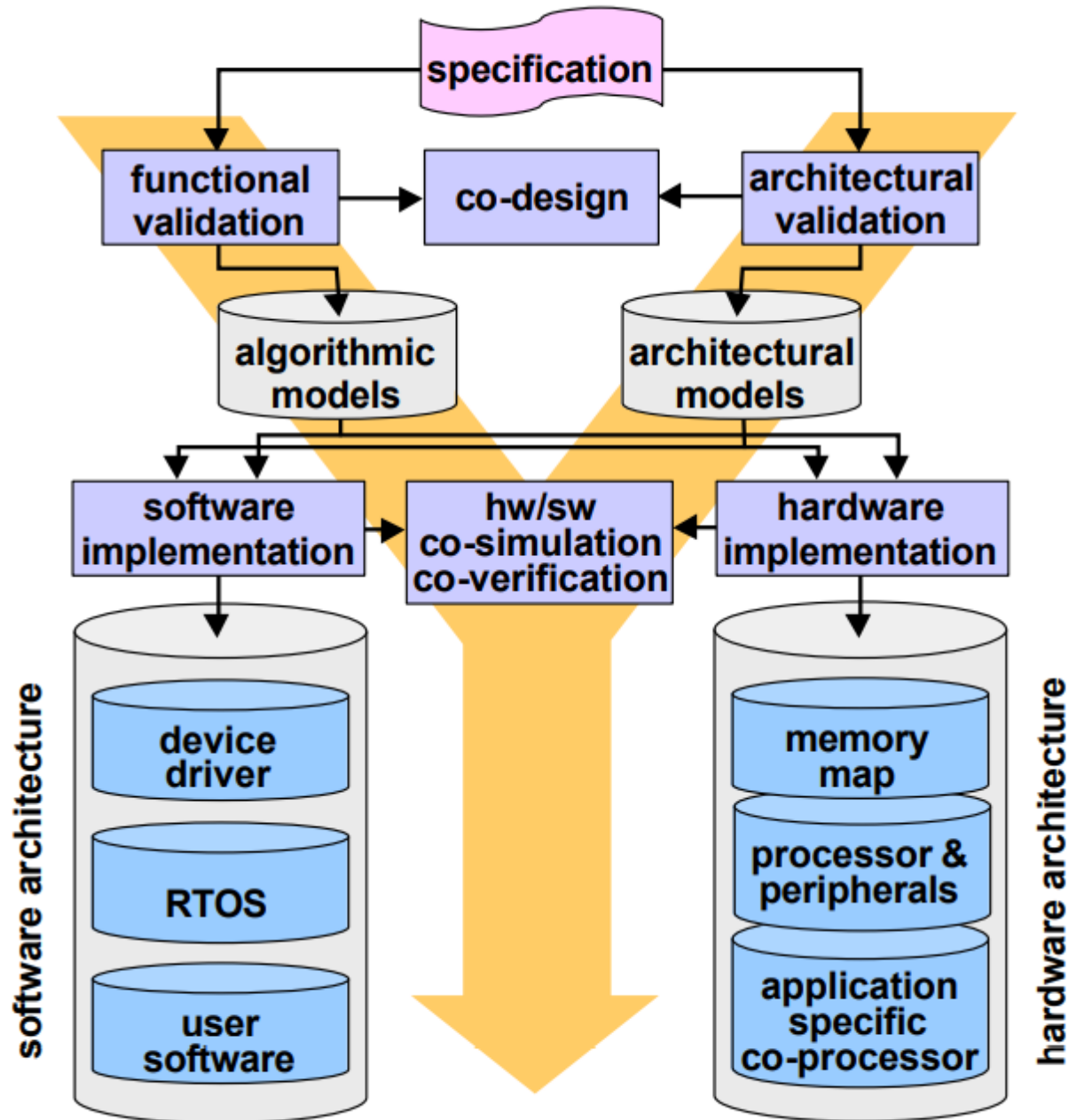
SYSTEMC SYSTEM DESIGN METHODOLOGY



- used for accelerating the software development by modelling a SOC
- Refinement Methodology:
 - The design is not converted from a C level description to an HDL in one large effort.
 - The design is slowly refined in small sections to add the necessary hardware and timing constructs to produce a good design.
 - Using this refinement methodology, the designer can more easily implement design changes and detect bugs during refinement.

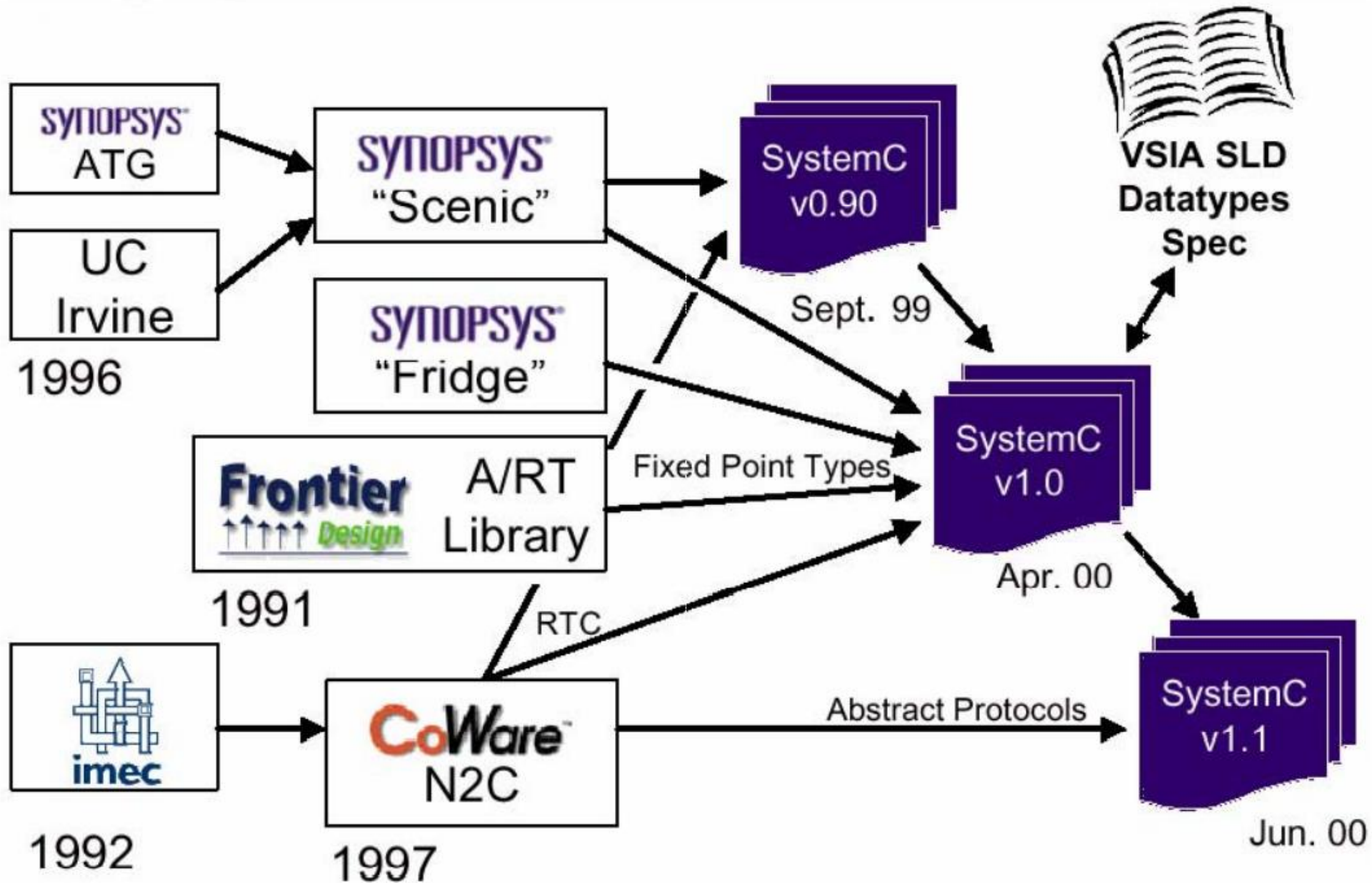


SYSTEM LEVEL DESIGN FLOW





SYSTEMC SYSTEM DESIGN METHODOLOGY



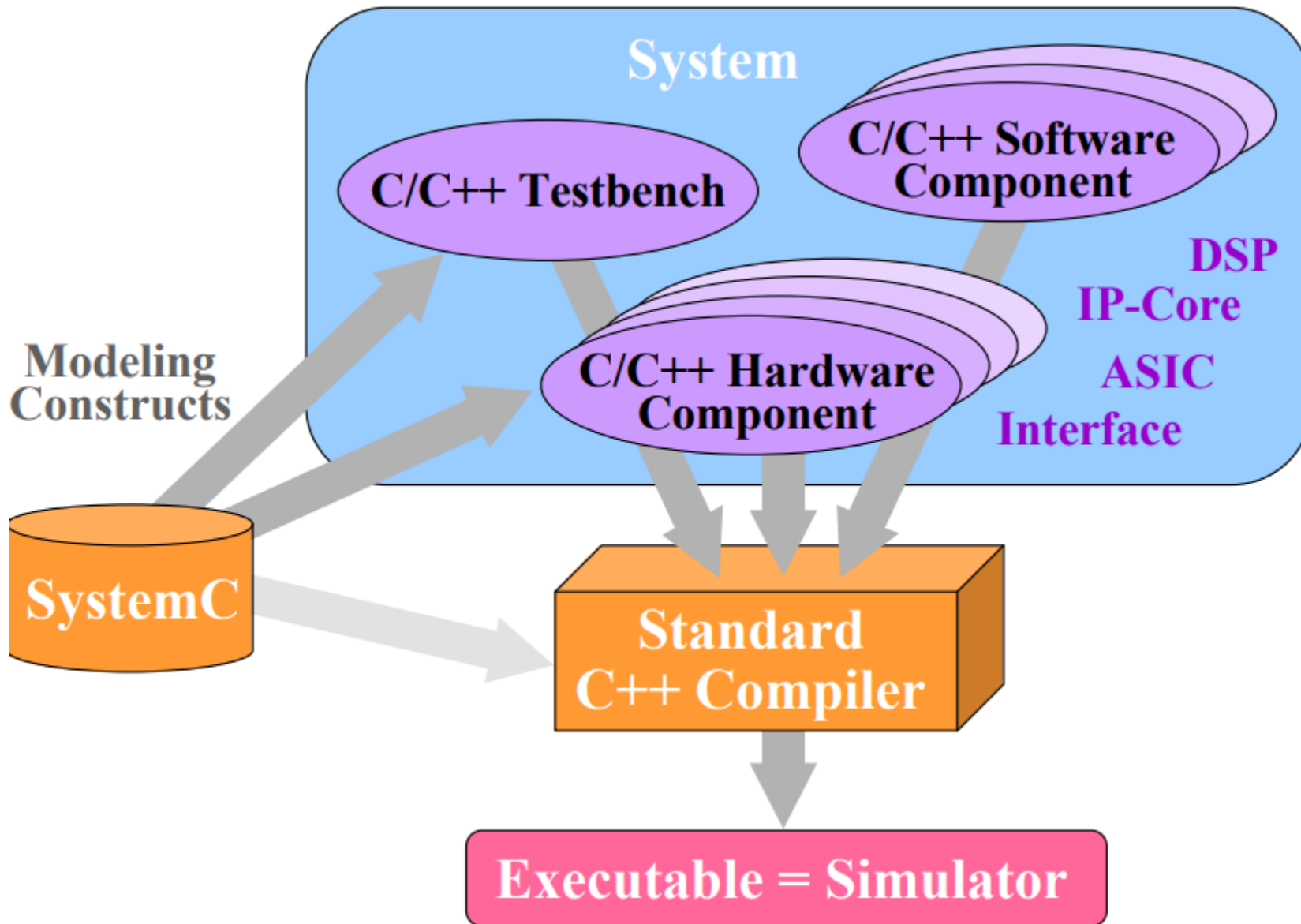


SYSTEM C

- A library of C++ classes
 - Processes (for concurrency)
 - Clocks (for time)
 - Modules, ports, signals (for hierarchy)
 - Waiting, watching (for reactivity)
 - Hardware data types
- A modeling style for modeling systems consisting of multiple design domains, abstraction levels, architectural components, real-life constraints
- A light-weight simulation kernel for high-speed cycle-accurate simulation



PRINCIPLE OF OPERATIONS





RESULTS

- executable specification
- testbenches
- written specification
- HW
 - understand specification
 - refine
 - validate re-using testbenches
 - synthesize

Why?

C/C++



SYSTEMC EXTENSIONS TO C++

- The SystemC Class Library:

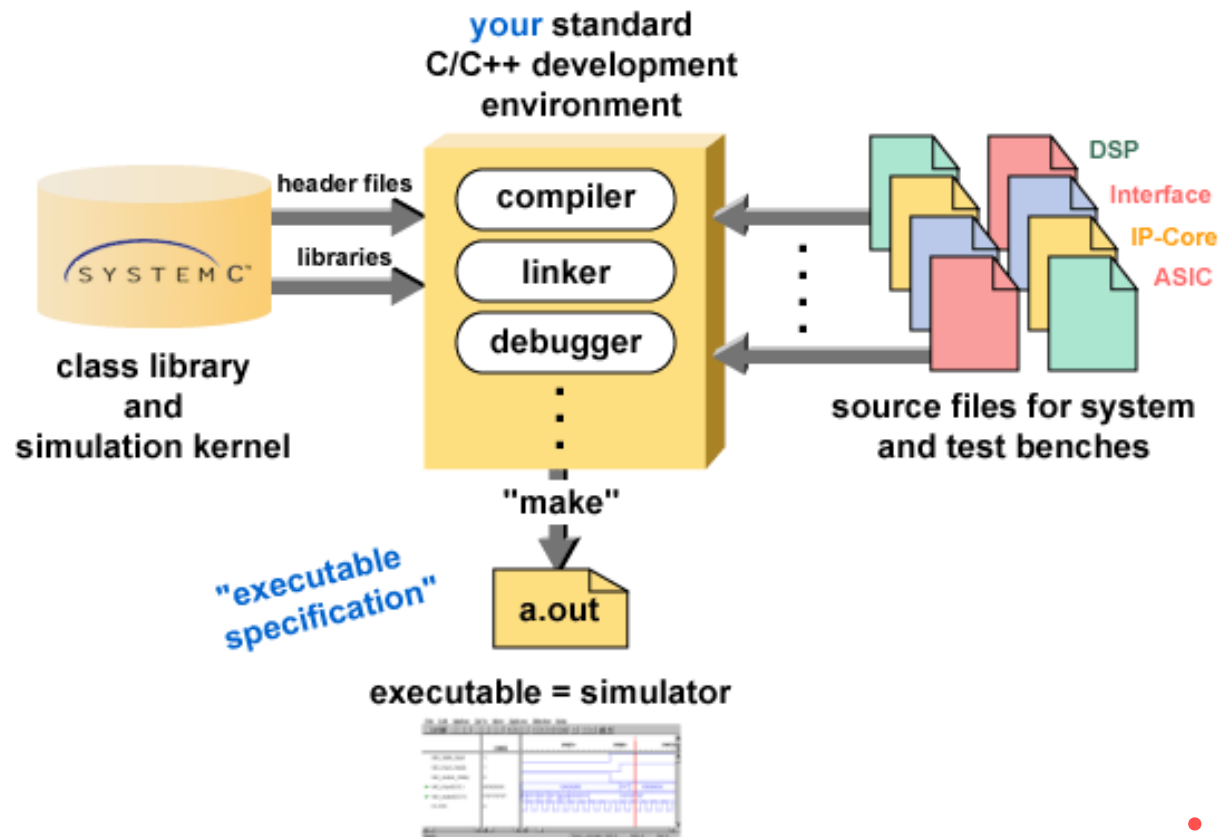
- provides the necessary constructs to model system architectures, including hardware timing, concurrency, and reactive behaviors that are missing in standard C++.

- The C++ object-oriented programming language:

- provides the ability to extend the language through classes, without adding new syntactic constructs.
- SystemC provides these necessary classes and allows designers to continue to use the familiar C++ language and development tools.



SYSTEMC DEVELOPMENT ENVIRONMENT



- **Simulator**

- Can be downloaded from www.systemc.org



MODULES AND HIERARCHY

■ Modules

- the basic building block within SystemC to partition a design.
 - Modules allow designers to hide internal data representation and algorithms from other modules.

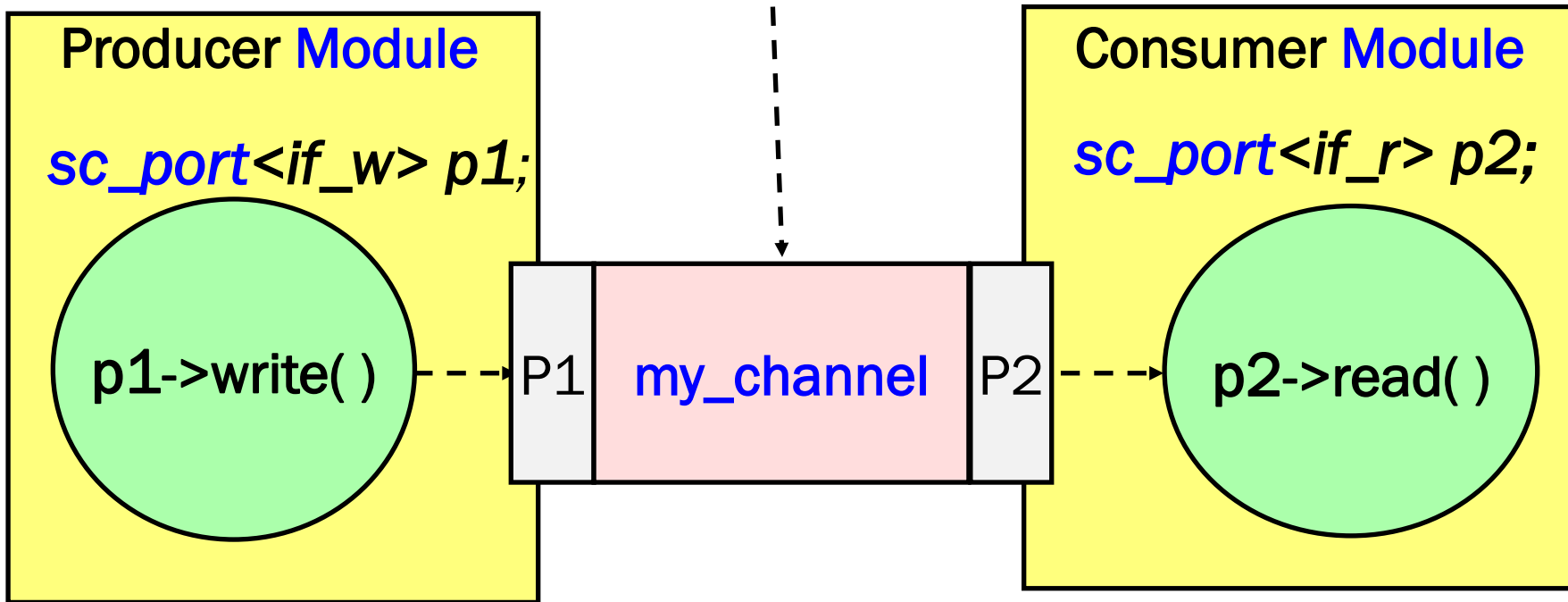
■ Declaration

- Using the macro `SC_MODULE`
 - `SC_MODULE(modulename) {`
- Using typical C++ struct or class declaration:
 - `struct modulename : sc_module {`



MODULE, PROCESS, PORTS, CHANNEL, INTERFACE

```
class my_channel: if_w, if_r, sc_module {  
    void write( ) {...}  
    void read ( ) {...}  
};
```



```
class if_w: virtual sc_interface {  
    virtual void write() = 0;  
};
```

```
class if_r: virtual sc_interface {  
    virtual void read() = 0;  
};
```



SYSTEMC DATA TYPES

- All C++ data types
- `sc_logic` - 0, 1, X, Z
 - `sc_lv<length>` - array of `sc_logic`
- `sc_uint<length>` , `sc_biguint<length>`, `sc_bv<length>`
- `sc_fix`



DATA TYPE

- Use native C++ data types wherever possible
- Use `sc_logic` if 4-valued logic is required
- Use `sc_uint<length>` for less than 64-bits and 2-valued logic
- Use `sc_biguint<length>` for more than 64-bits and 2-valued logic
- Use `sc_fix` for fixed point arithmetic



SYSTEMC INTERFACE

- Declares a set of access methods needed for communication between IPs
- Do not provide any implementation of the access methods
- Implementation of interfaces are done within a **channel**
 - **sc_signal_in_if** → **T& read() const** interface
 - **sc_signal_inout_if** → **void write(const T&)** interface
- You can define your own interfaces but it should inherit from **sc_interface**

```
template <typename T>
class my_interface : public sc_interface {
    int read(int & addr) = 0;
    void write( int& addr, int & data) = 0;
}
```



INTERFACE EXAMPLES

```
// -----
```

```
// sc_signal_in_if
```

```
// this interface provides a 'read' method
```

```
// -----
```

```
template <class T>
```

```
class sc_signal_in_if : virtual public sc_in
```

```
    public:
```

```
    // interface methods
```

```
    virtual const T& read() const = 0;
```

```
};
```

```
// -----
```

```
// sc_signal_inout_if
```

```
// this interface provides 'read' & 'write'  
method
```

```
// -----
```

```
template <class T>
```

```
Class sc_signal_inout_if : virtual public  
sc_signal_in_if {
```

```
    public:
```

```
    // interface methods
```

```
    virtual void write( const T& ) = 0;
```

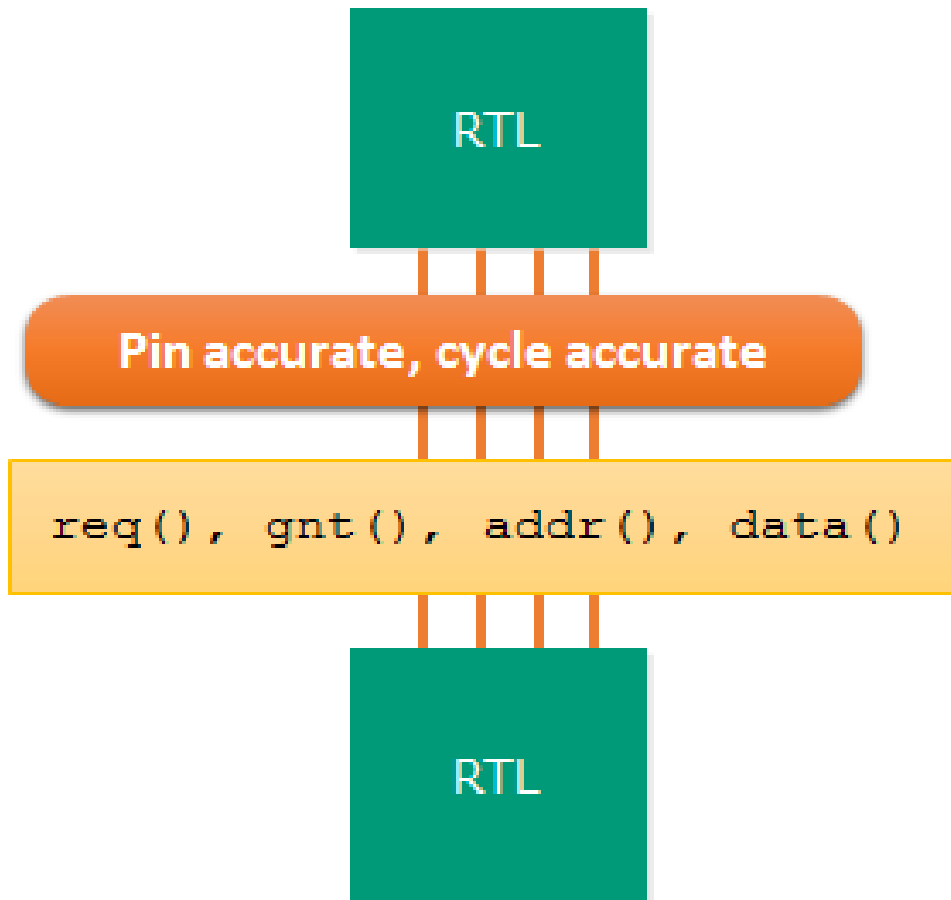
```
};
```



DIFFERENCE INTERFACE TYPE FOR DIFFERENT ABSTRACTION LEVEL

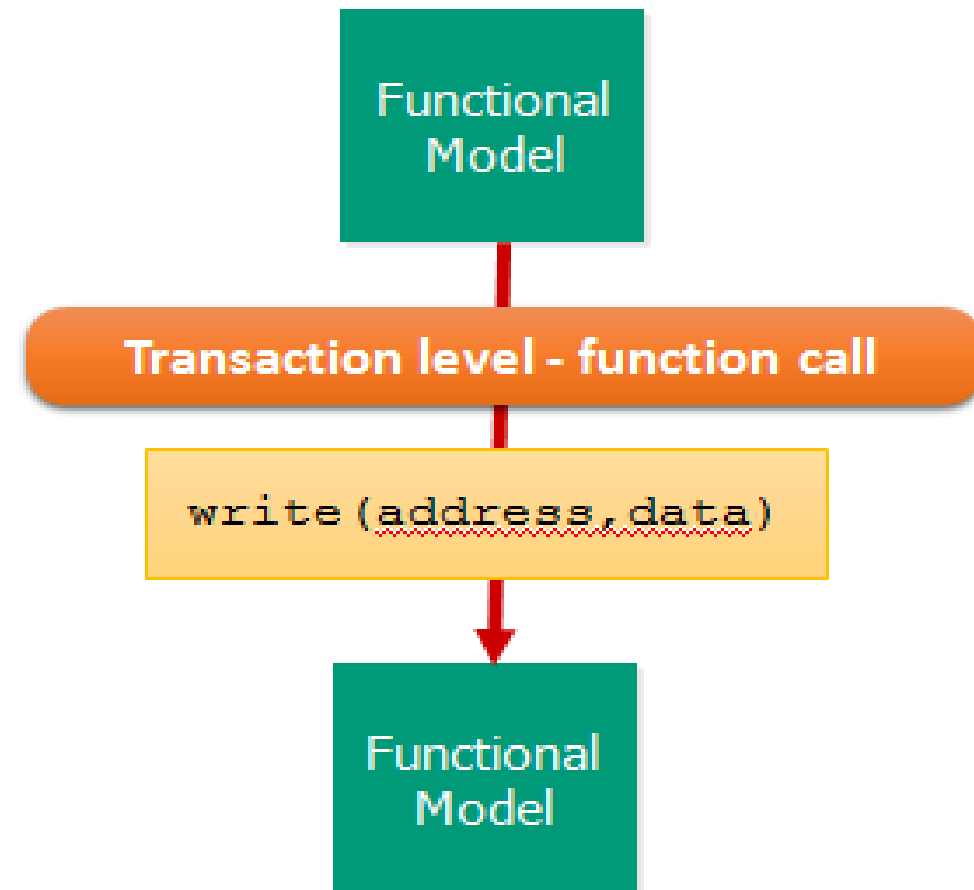
Cycle Accurate Model

Interfaces in form of signals of request, grant, address, data, etc



Transaction Level Model

Interfaces in the form of read (address) or write (address, data)

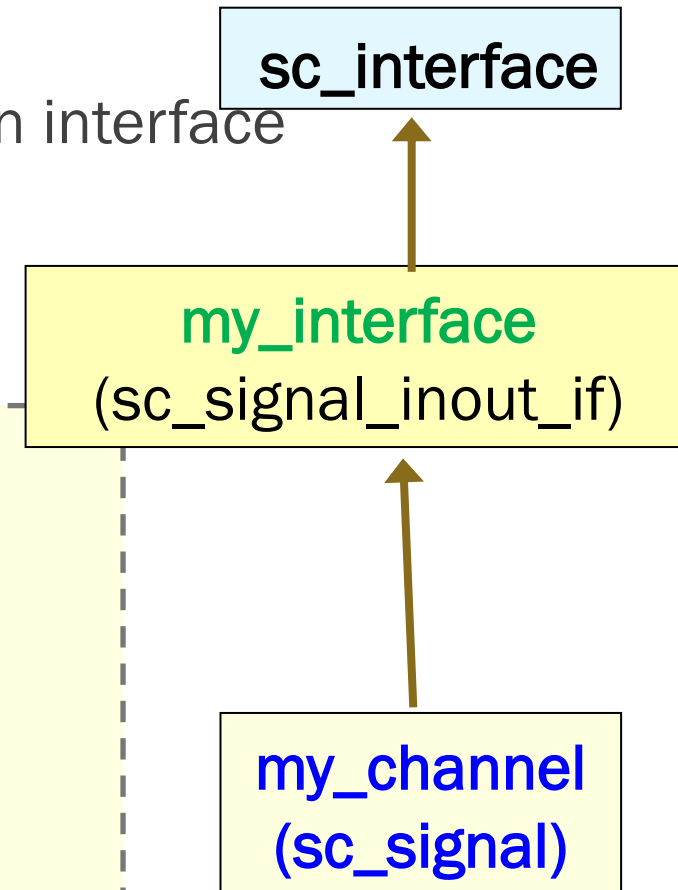




SYSTEMC CHANNEL?

- Means for communication between modules
- Provides the functionality of methods declared in interface
 - Derived from interfaces

```
template <typename T>
class my_channel : my_interface, sc_module {
    int read(int& addr) {
        return arr[addr];
    }
    void write(int & addr, int & data) {
        arr[addr]= data;
    }
}
```





CHANNEL EXAMPLE

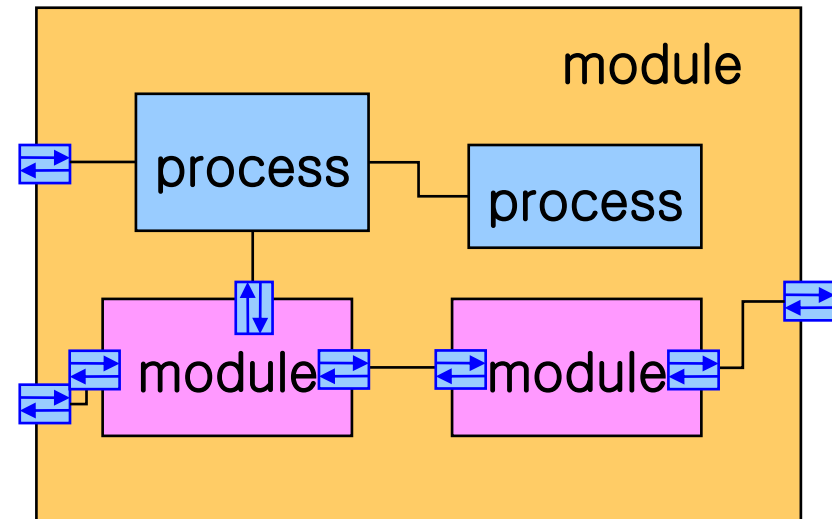
```
// -----  
// sc_signal<T>  
// -----  
  
template <class T>  
  
class sc_signal : public  
sc_signal_inout_if<T> {  
    m_value;  
  
    T& read () {  
        //immediate return  
        return m_value;  
    }  
}
```

```
// -----  
// sc_signal<T>  
// -----  
  
template <class T>  
  
class sc_signal : public  
sc_signal_inout_if<T> {  
    m_value;  
  
    T& read () {  
        //return after some time  
        wait (10, SC_NS);  
        return m_value;  
    }  
}
```



MODULES AND HIERARCHY

- Elements:
 - ports,
 - local signals,
 - local data,
 - other modules,
 - processes, and
 - constructors.





PORTS

```
SC_MODULE(fifo) {  
    sc_in<bool> load;  
    sc_in<bool> read;  
    sc_inout<int> data;  
    sc_out<bool> full;  
    sc_out<bool> empty;  
    ...  
    //rest of module not shown  
}
```

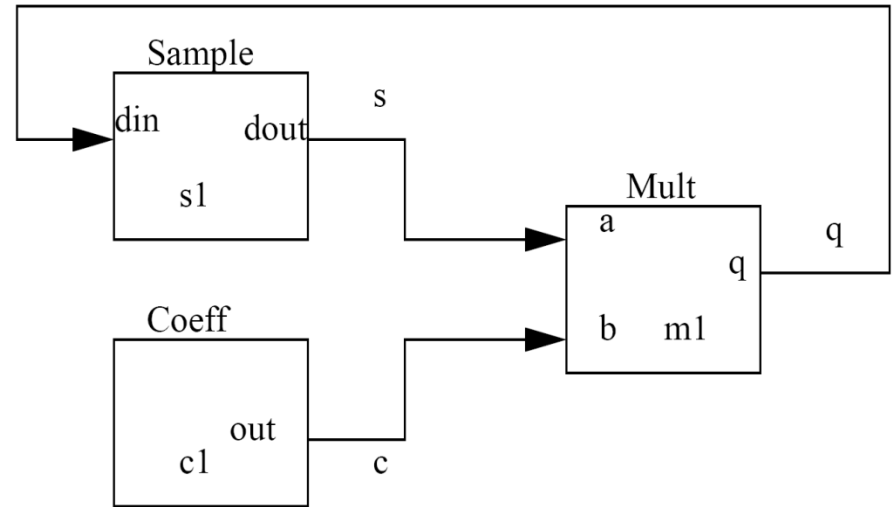




SIGNALS

```
sc_signal<type> q, s, c;
```

- **Positional Connection**
- **Named Connection**

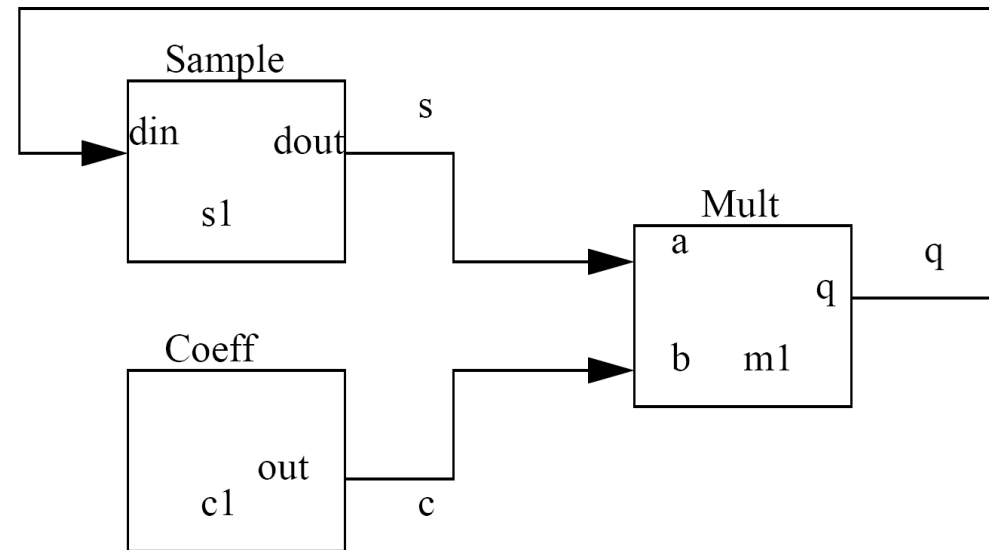




NAMED CONNECTION

- `Instancename.portname(signalname);`

```
SC_MODULE(filter) {  
    sample *s1;  
    coeff *c1;  
    mult *m1;  
  
    c; sc_signal<sc_uint<32>> q, s,  
  
    SC_CTOR(filter) {  
        s1 = new sample ("s1");  
        s1->din(q);  
        s1->dout(s);  
        c1 = new coeff ("c1");  
        c1->out(c);  
        m1 = new mult ("m1");  
        m1->a(s);  
        m1->b(c);  
        m1->q(q);  
    }  
}
```

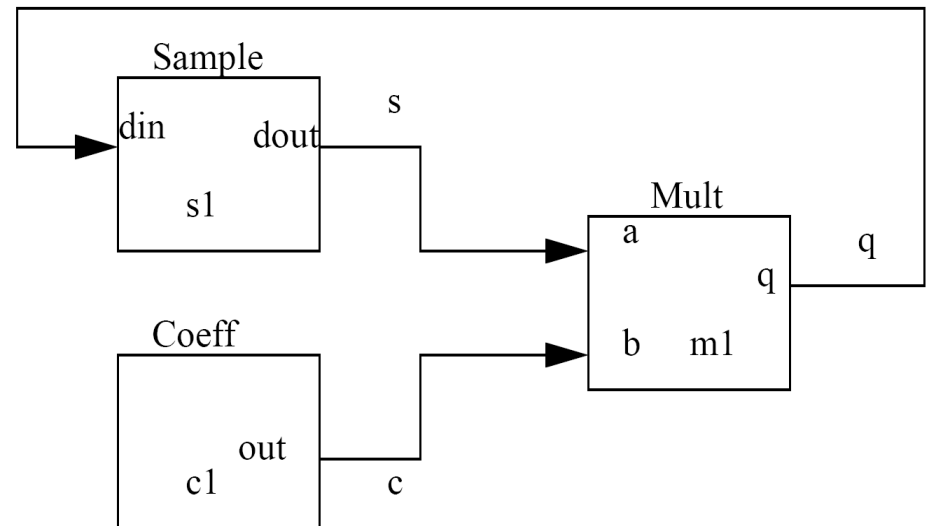




POSITIONAL CONNECTION

- `Instancename (sig1, sig2, ...);`

```
SC_MODULE(filter) {  
    sample *s1;  
    coeff *c1;  
    mult *m1;  
  
    sc_signal<sc_uint<32> > q, s, c;  
  
    SC_CTOR(filter) {  
        s1 = new sample ("s1");  
        (*s1)(q,s);  
        c1 = new coeff ("c1");  
        (*c1)(c);  
        m1 = new mult ("m1");  
        (*m1)(s,c,q);  
    }  
}
```





INSTANTIATION

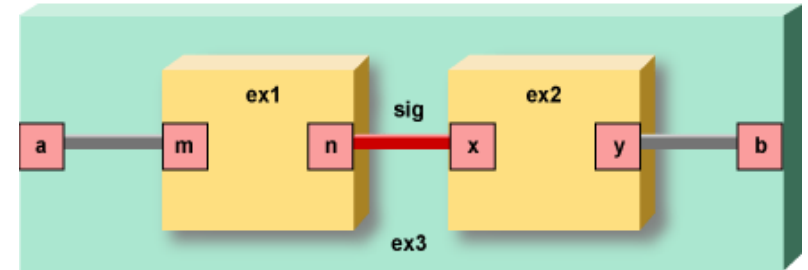
```
// file ex1.h
SC_MODULE(ex1) {
    sc_port<sc_fifo_in_if<int> > m;
    sc_port<sc_fifo_out_if<int> > n;

    SC_CTOR(ex1) {
        // Rest of the module body is not shown
    };
};
```

```
// file ex2.h
SC_MODULE(ex2) {
    sc_port<sc_fifo_in_if<int> > x;
    sc_port<sc_fifo_out_if<int> > y;

    SC_CTOR(ex2) {
        // Rest of the module body is not shown
    };
};
```

ex1_instance("ex1_instance"):
- passes the instance label to the constructors of the instance.



```
SC_MODULE(ex3) {
    sc_port<sc_fifo_in_if<int> > a;
    sc_port<sc_fifo_out_if<int> > b;
    sc_fifo<int> sig1;
    // Instances of ex1 and ex2
    ex1 ex1_instance;
    ex2 ex2_instance;
    // Module Constructor
    SC_CTOR(ex3) :
    ex1_instance("ex1_instance"), //init'n
    ex2_instance("ex2_instance") //init'n
    {
        // Named connection for ex1
        ex1_instance.m(a);
        ex1_instance.n(sig1);
        // Positional connection for ex2
        ex2_instance(sig1, b);
        // Rest of constructor body not shown
    };
};
```




SC_MAIN()

- The top level is a special function called `sc_main`.
 - It is in a file named `main.cpp` or `main.cc` (standard practice - not a requirement).
- `sc_main()` is called by SystemC and is the entry point for your code.
- The execution of `sc_main()` until the `sc_start()` function is called (described later) is considered to be the elaboration time of SystemC.

```
int sc_main (int argc, char *argv [ ] ) {  
    // body of function  
    ...  
    return 0 ;  
}
```



INSTANTIATION STEPS IN SC_MAIN()

■ Two Steps:

■ Declaration and Initialization.

```
module_name instance_name ("string_name") ;
```

- recommended to keep the string_name the same as the instance_name.

■ Module Instantiation - Port Binding

■ Named Connection:

```
instance_name.port_name(channel_or_port);
```

■ Positional Connection:

```
instance_name(channel_or_port, channel_or_port,...);
```



SYSTEMC

- A set of modeling constructs in RTL or Behavioral abstraction level
- Structural design using Modules, Ports, and Signals
- Rich set of data types including bit-true types
 - Specially: Fixed-Point data types for DSP apps
- Concurrent Behavior is described using Processes
 - Processes can suspend and resume execution
 - Limited control over awakening events
 - Events and sensitivity list are static (specified at compile-time)
- SC_THREAD and SC_CTHREAD processes
 - Can suspend and resume execution
 - Require their own execution stack
 - Memory and Context-switching time overhead
 - SC_METHOD gives best simulation performance



SYSTEMC

- Hardware Signals are hard to model in software
 - Initialization to X
 - Used to detect reset problems
 - `sc_logic`, `sc_lv` data types
 - Multiple drivers
 - resolved logic signals (`sc_signal_rv`)
 - Not immediately change their output value
 - Delay is essential
 - Capability to swap two registers on clock edge
- Delayed assignment and delta cycles
 - Same as VHDL and Verilog
 - Essential to model hardware signal assignments
 - Each assignment to a signal isn't seen by other processes until the next delta cycle
 - Delta cycles don't increase user-visible time
 - Multiple delta cycles may occur



SYSTEMC 2.0

- Primary goal: Enable System-Level Modeling
 - Systems include hardware, software, or both
 - Challenges:
 - Wide range of design models of computation
 - Wide range of design abstraction levels
 - Wide range of design methodologies
- SystemC 2.0
 - Introduces a compact general purpose modeling foundation => Core Language
 - Elementary channels
 - library models provided (FIFO, Timers, ...)
 - Includes SystemC 1.0 Signals
 - Support for various models of computation, methodologies, etc.
 - Built on top of the core language, hence are separate from it



COMMUNICATION AND SYNCHRONIZATION

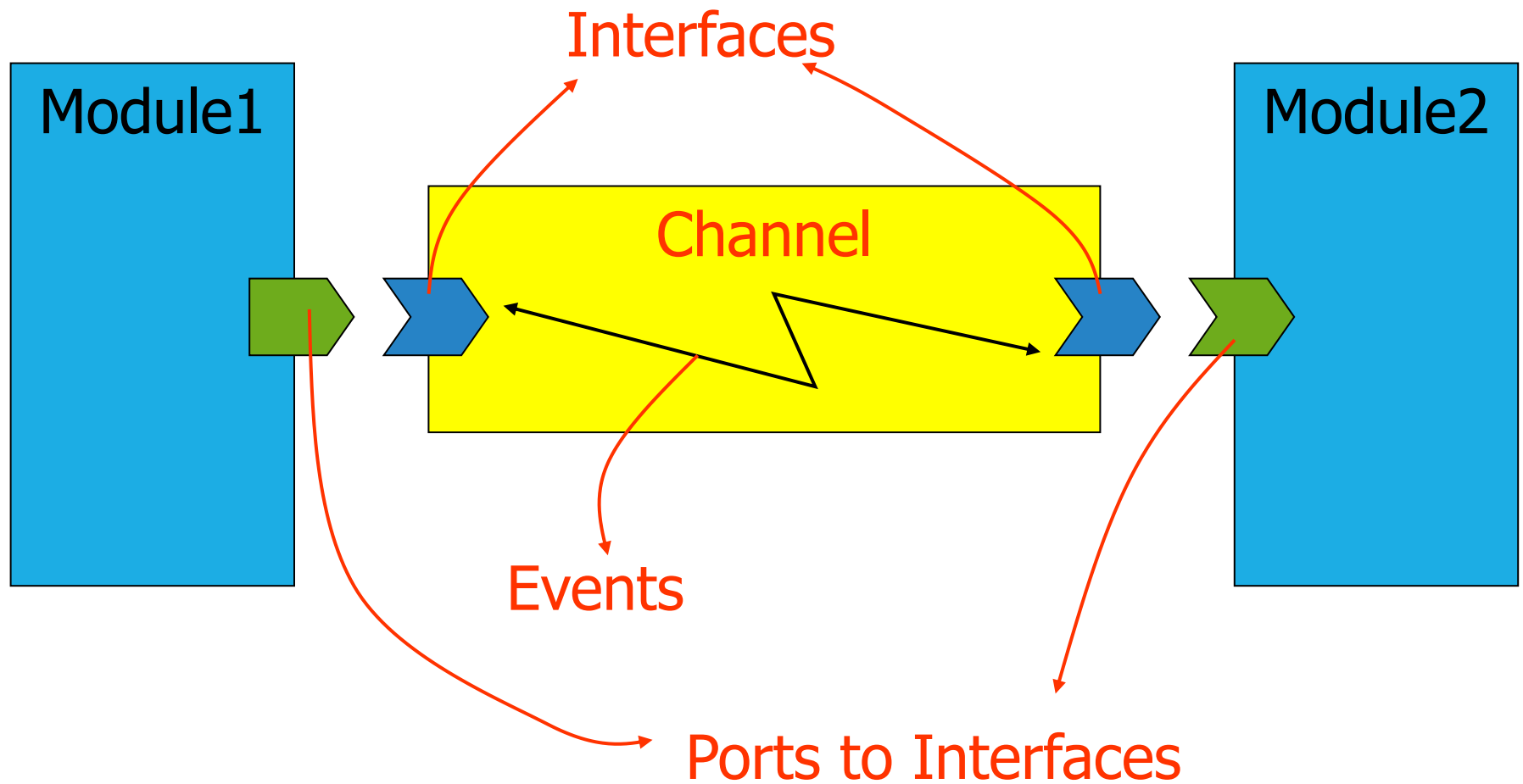
- SystemC 1.0 *Modules* and *Processes* are still useful in system design
- NOTE: communication and synchronization mechanisms available in SystemC 1.0 (*Signals*) are **restrictive** for system-level modeling
 - Communication using queues
 - Synchronization (access to shared data) using mutexes
- SystemC 2.0 includes general-purpose mechanisms
 - **Channel**
 - A container 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

Builds advanced
comm. & sync.
models

e.g. HW-signals, queues
(FIFO, LIFO, message
queues,...) semaphores,
memories and busses
(RTL and TLM)

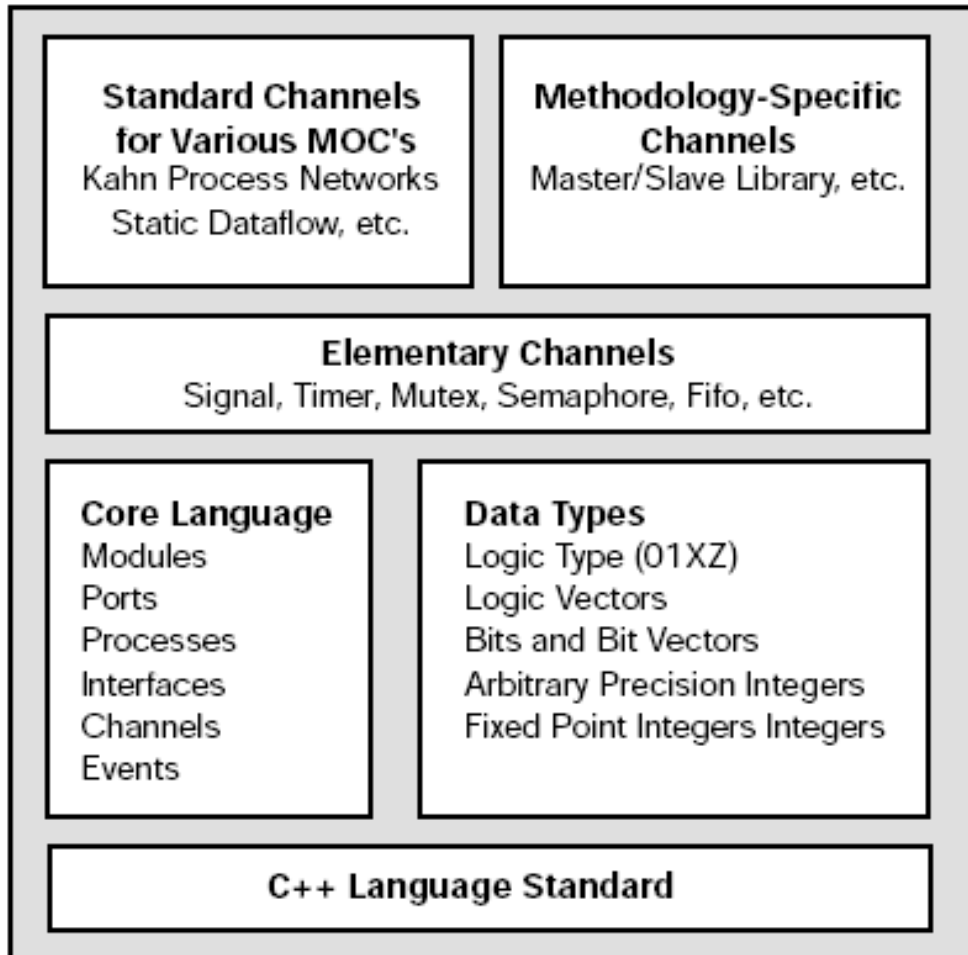


COMMUNICATION AND SYNCHRONIZATION





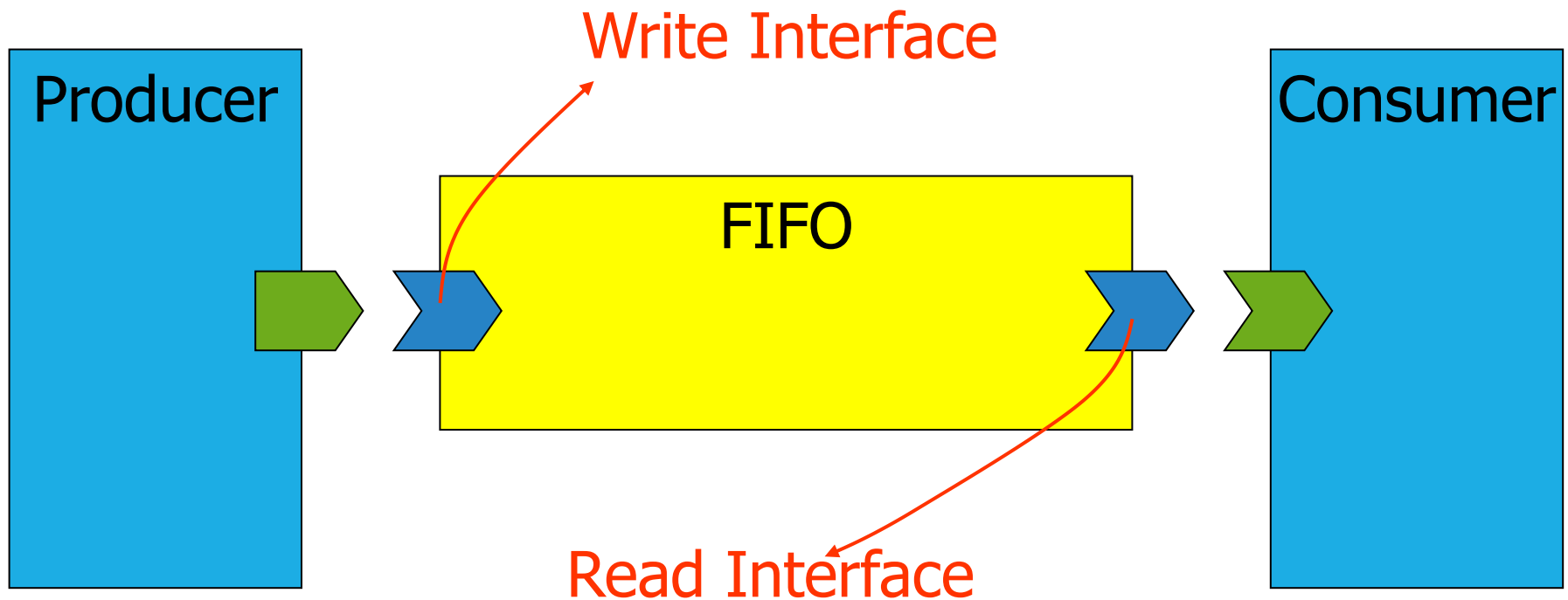
SYSTEMC 2.0 LANGUAGE ARCHITECTURE



- All built on C++
- Upper layers cleanly built on lower ones
- Core language
 - Structure
 - Concurrency
 - Communication
 - Synchronization
- Data types separate from the core language
- Commonly used communication mechanisms and MOC built on top of core language
- Lower layers can be used without upper ones



FIFO



Problem definition: FIFO communication channel with blocking read and write operations

Source available in SystemC installation, under “`examples\systemc`” subdirectory

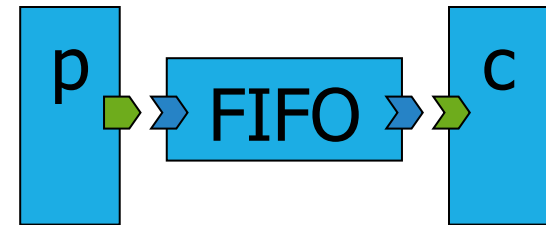


FIFO EXAMPLE

■ Interfaces

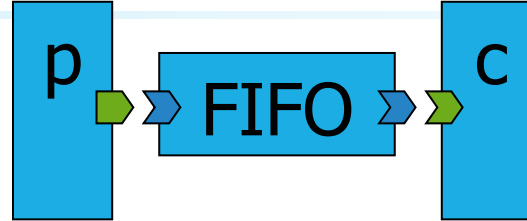
```
class write_if : public sc_interface
{
    public:
        virtual void write(char) = 0;
        virtual void reset() = 0;
};

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





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:
        SC_CTOR(fifo) {
            num_elements = first=0;
        }
}
```

```
void write(char c) {
    if ( fifo_full() )
        wait(read_event);

    data[ <you say> ]=c;
    ++num_elements;
    write_event.notify();
}

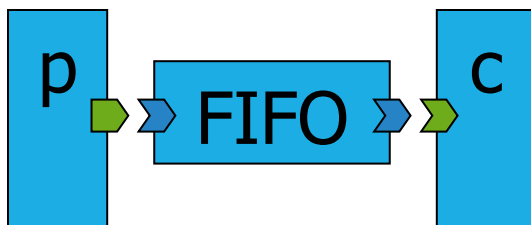
void read(char &c) {
    if( fifo_empty() )
        wait(write_event);

    c = data[first];
    --num_elements;
    first = <you say>;
    read_event.notify();
}
```



FIFO CHANNEL

```
void reset() {  
    num_elements = first = 0;  
}  
  
int num_available() {  
    return num_elements;  
}  
}; // end of class declarations
```



- All channels must
 - be derived from `sc_channel` class
 - SystemC internals
(kernel\sc_module.h)

```
typedef sc_module  
sc_channel;
```
 - 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

■ Note

- `wait()` call with arguments => dynamic sensitivity
 - `wait(sc_event)`
 - `wait(time) // e.g. wait(200, SC_NS);`
 - `wait(time_out, sc_event) //wait(2, SC_PS, e);`

■ Events

- the fundamental synchronisation primitive in SystemC
- different from signals!!
 - have no type and no value
 - always cause sensitive processes to be resumed
 - can be specified to occur:
 - immediately/ one delta-step later/ some specific time later



WAIT () FUNCTION

```
// wait for 200 ns.  
sc_time t(200, SC_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 );
```



NOTIFY () **METHOD OF** SC_EVENT

- Possible calls to notify():

```
sc_event my_event;
```

```
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
```



COMM. MODELING EXAMPLE

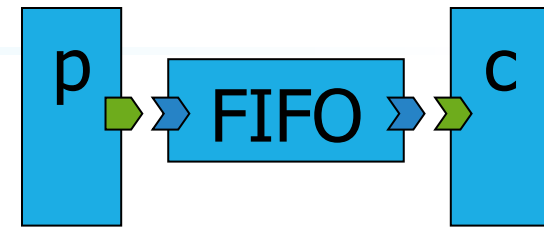
- Producer module
 - `sc_port<write_if> out;`
 - Producer can only call member functions of `write_if` interface
 - Consumer module
 - `sc_port<read_if> in;`
 - Consumer can only call member functions of `read_if` interface
- e.g. Cannot call `reset()` method
- Producer and consumer are
 - unaware of how the **p** **on** **FIFO** works
 - just aware of their respective *interfaces*
 - Channel implementation is hidden from communicating modules

```
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();
        }
    }
};
```




COMM. MODELING EXAMPLE



```
SC_MODULE(top) {  
    public:  
        fifo *afifo;  
        producer *pproducer;  
        consumer *pconsumer;  
  
    SC_CTOR(top) {  
        afifo = new fifo("Fifo");  
  
        pproducer=new producer("Producer");  
        pproducer->out(afifo);  
  
        pconsumer=new consumer("Consumer");  
        pconsumer->in(afifo);  
    };  
};
```



COMM. MODELING EXAMPLE

- 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 MODELS OF COMPUTATION

- Many different models
 - The best choice is not always clear!
- Basic topics in a computation model
 - The model of time, and event ordering constraints
 - Time model (real valued, integer-valued, untimed)
 - Event ordering (globally ordered, partially ordered)
 - Supported methods of communication between concurrent processes
 - Rules for process activation
- Generic model of computation
 - The designer can implement desired model
- All discrete-time models are supported
 - Static Multi-rate Data-flow
 - Dynamic Multi-rate Data-flow
 - Kahn Process Networks
 - Communicating Sequential Processes
 - Discrete Event as used for
 - RTL hardware modeling
 - network modeling (e.g. stochastic or “waiting room” models)
 - transaction-based SoC platform-modeling
- not suitable for continuous-time models (e.g. analog modeling)!!

e.g. Signals are realized on top of channels, interfaces, and events



FUTURE EVOLUTION OF SYSTEMC

- IEEE 1666-2011 - IEEE Standard for Standard SystemC Language Reference Manual
- IEEE 1666.1-2016 - IEEE Standard for Standard SystemC(R) Analog/Mixed-Signal Extensions Language Reference Manual
- download
<http://www.accellera.org/downloads/standards/systemc>
- Ver. 2.3.3
 - Support for RTOS modeling different approaches
 - Fork and join threads + dynamic thread creation
 - Interrupt or abort a thread and its children
 - Specification and checking of timing constraints
 - Abstract RTOS modeling and scheduler modeling
 - New features in the core language
 - Support for analog mixed signal modeling



- Extensions as libraries on top of the core language
 - Standardized channels for various MOC (e.g. static dataflow and Kahn process networks)
 - Testbench development
 - Libraries to facilitate development of testbenches
 - data structures that aid stimulus generation and response checking
 - functions that help generate randomized stimulus, etc.
 - System level modeling guidelines
 - library code that helps users create models following the guidelines
 - Interfacing to other simulators
 - Standard APIs for interfacing SystemC with other simulators, emulators, etc.



■ Accellera Systems Initiative Policies and Organizational Documents

- [UVM 2017-1.1 Reference Implementation](#)
- SystemC AMS 2.3
- IP-XACT Vendor Extensions
- Portable Stimulus 1.0a
- SCE-MI 2.4
- SystemC 2.3.3 (w/TLM), AMS 2.0, CCI 1.0, & Synthesis 1.4.7
- SystemRDL 2.0

All Accellera standards | IEEE standards

■ UVM-SystemC is

- standard to develop structured verification environments following the Universal Verification Methodology (UVM)