Universität Bremen

SystemC – Features of SystemC 2.0

Rolf Drechsler

Daniel Große

University of Bremen

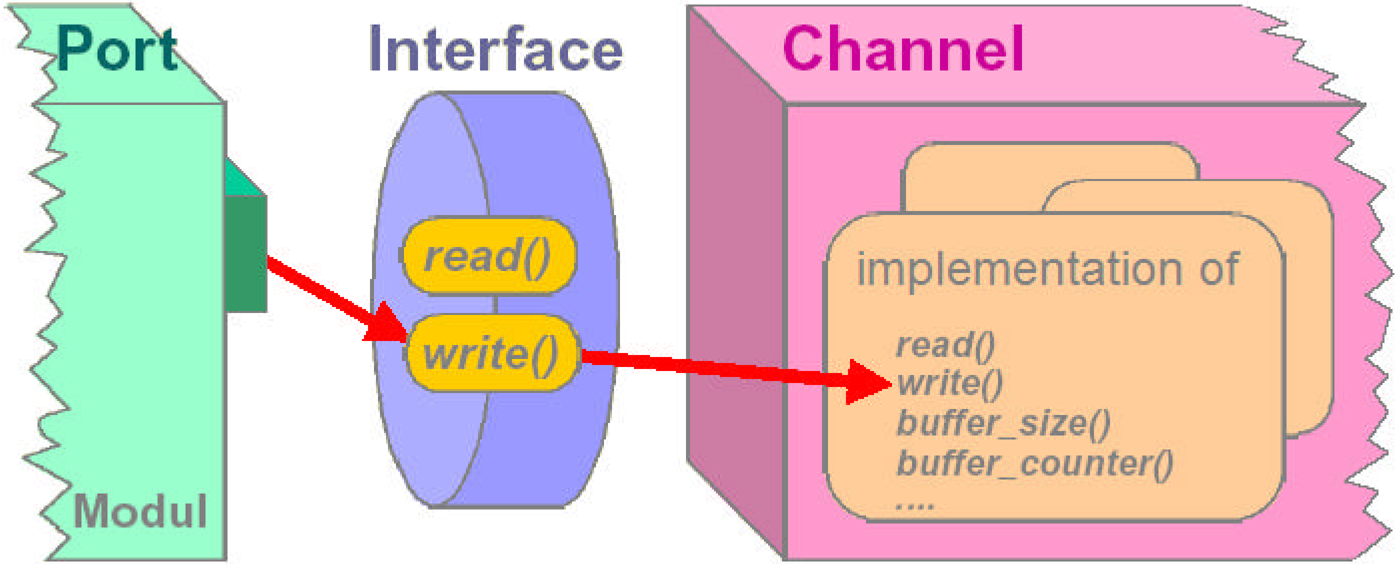
SystemC 2.0 – Communication and Synchronization

- Hardware signal for communication is not sufficiently general for system-level
- At system-level you need more:
  - Delayed connections
  - Buffered connections (FIFO, message queues)
  - Communication through arbitrary events
  - Synchronization (access to shared data) using mutexes

=> Concept of **Interfaces, Ports and Channels**

Abstract Communication

- Connect a module port through an interface with a channel



Interfaces

- Defines a set of access methods, but does *not* implement these methods (*abstract class*)
- Has no data fields
- A port sees only those channel methods that ar defined by the interface
- A port is not able to access any other method o data field in the channel
- Define by deriving from class sc_interface

Ports

- Processes can access a channel methods through ports
- More than simple read and write is possible:
  - Transmit additional data (e.g. data address)
  - Get status of a channel (e.g. data available)
  - More complex sensitivity (wait for request)
- Binding of a channel to a port by operator (..)

Channels (1)

- Container for communication functionality
- Implement one or more interfaces
- A channel must be
  - 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*

Channels (2)

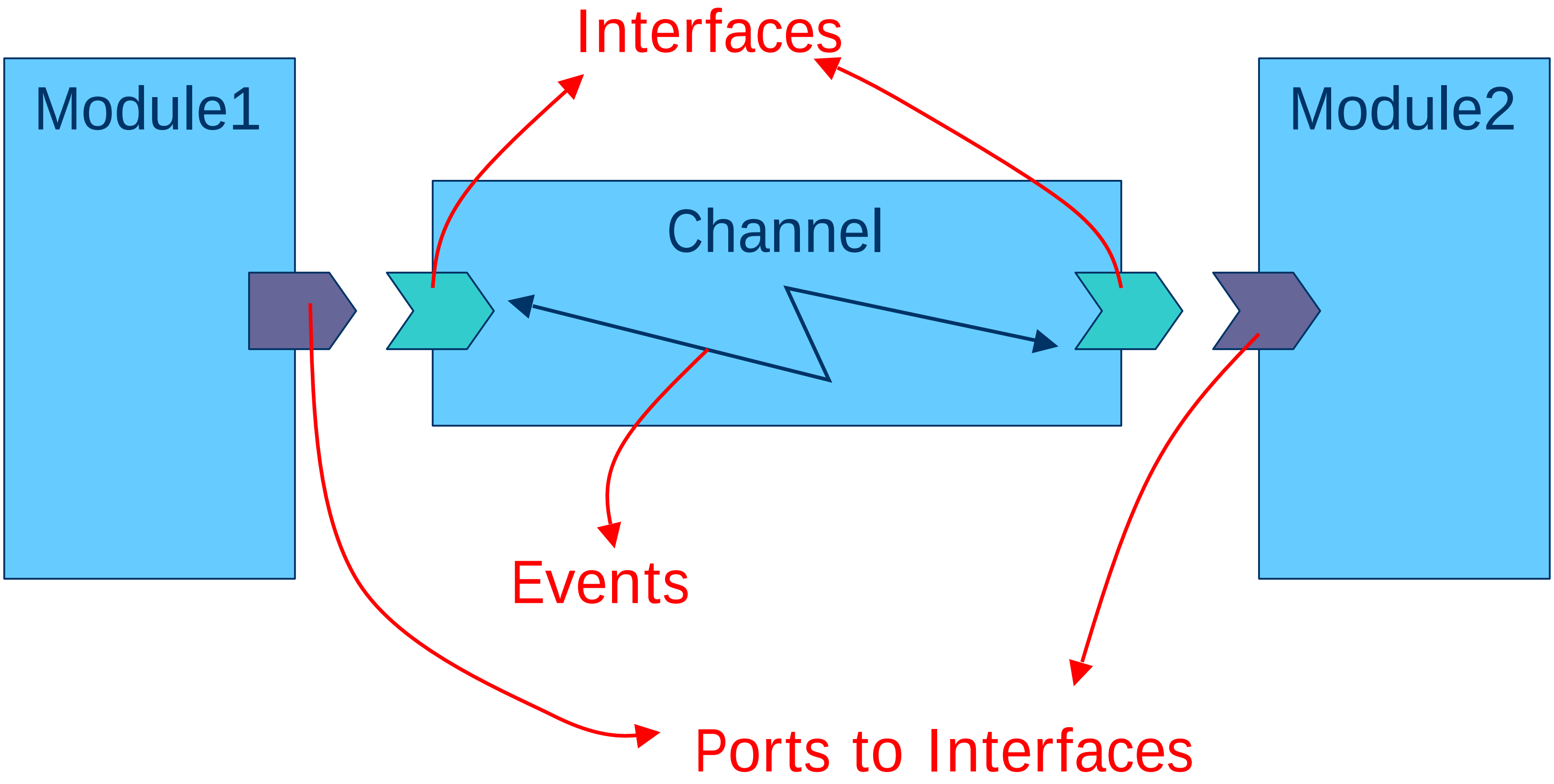
- Distinction between
  - Primitive channels
    - Do not contain processes or modules
    - Can not access other channels
  - Hierarchical channels
    - Complete SystemC-Modules
    - Can access other channels
- Example of primitive channels:

  sc_signal<T>, sc_fifo<T>, sc_mutex

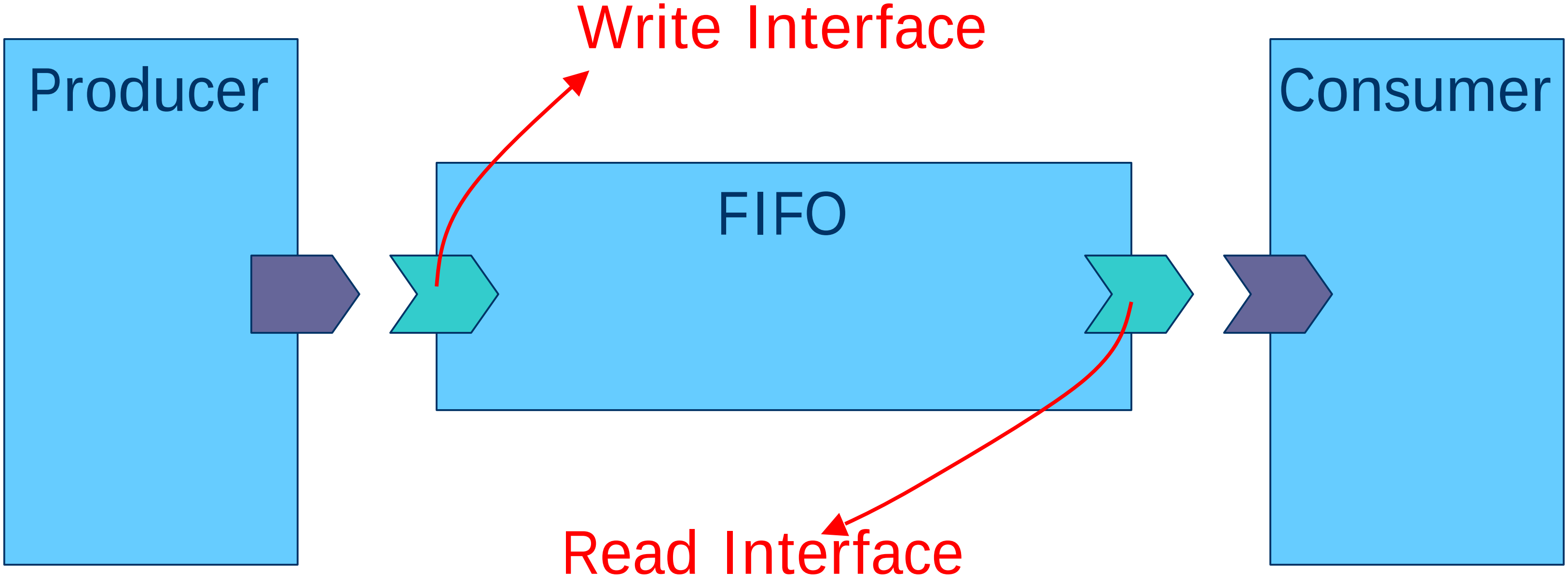
Example: FIFO

- FIFO of 10 characters, along with a producer and a consumer process, communicating through the FIFO

Communication and Synchronization



Example: FIFO (1)



Example: FIFO (2)
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;
};
```

Example: FIFO (3)
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();
}
```

Example: FIFO (4)
FIFO channel (cont’d)

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

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

Example: FIFO (5)

- Note the following extensions beyond SystemC 1.0:
  - 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 (only: sc_event e)
    - always cause sensitive processes to be resumed
    - can be specified to occur:
      - immediately/ one delta-step later/ some specific time later

Completing the FIFO Example (1)

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

   SC_CTOR(producer) {
      SC_THREAD(main);
   }

   void main() {
     char c;
     while (true) {
        out->write(c); // write c to FIFO
        if(...)
          out->reset(); // reset FIFO
      }
   }
};
```

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

   SC_CTOR(consumer) {
      SC_THREAD(main);
   }

   void main() {
     char c;
     while (true) {
        in->read(c); // read c
        if (in->num_available()>5)
        //perhaps speed up processing
     }
   }
};
```

Completing the FIFO Example (2)

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

    SC_CTOR(top) {
            pfifo = new fifo(“fifo”);
            pproducer=new producer(“Producer”);
            // bind the FIFO to the producer´s port
            pproducer->out(fifo);

            pconsumer=new consumer(“Consumer”);
            // bind the FIFO to the consumer´s port
            pconsumer->in(fifo);
    };
```

Completing the FIFO Example (3)

- 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 FIFO Example (4)

- 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 Roadmap

- SystemC 1.0: Hardware Design Flow
  - RTL and Behavioral Hardware Modeling
- SystemC 1.X: Master-Slave Comm. Library
- SystemC 2.0: System Design Flow
  - General purpose: communication and synchronization
  - Communication Refinement
  - Multiple, customizable models of computation
  - Dynamic thread creation

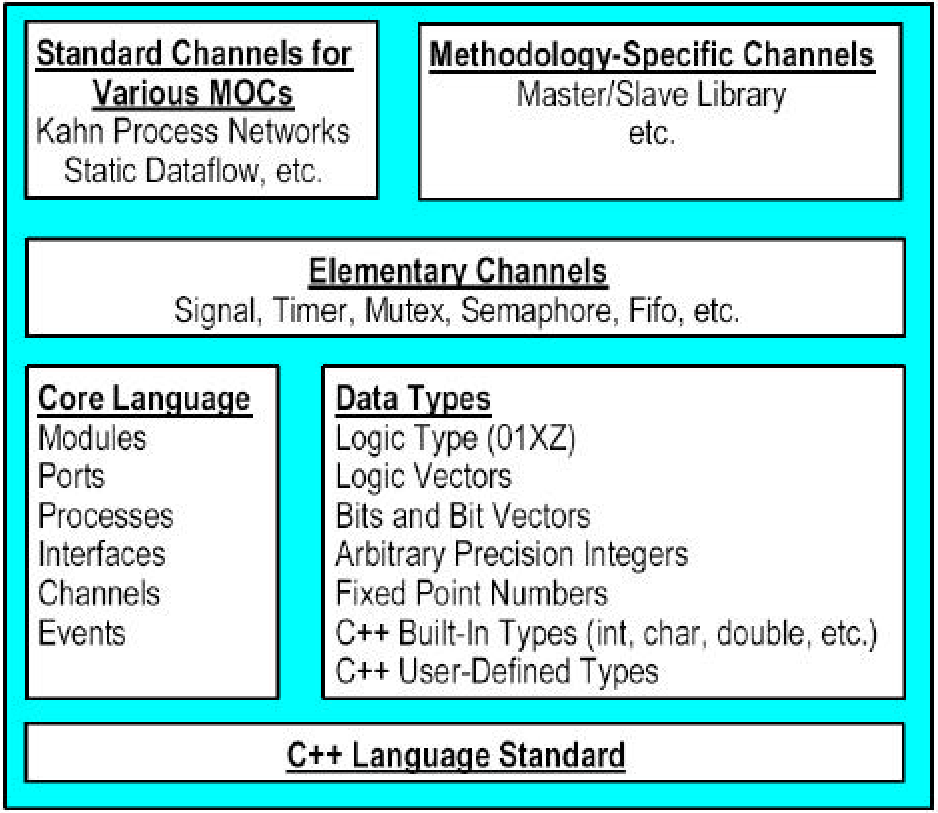
SystemC Roadmap (cont´d)

- SystemC 2.X: Extensions to System Design Flow
  - Fork & Join
  - Interrupt / abort for behavioral hierarchy
  - Timing specification and constrains
- SystemC 3.X: Software Design Flow
  - Abstract RTOS modeling & scheduler modeling
- SystemC 4.X: Analog/Mixed Signal Systems Modeling

SystemC Language Architecture

Upper layers are built cleanly on lower layers.

Lower layers can be used without upper layers.



Standard Channels for Various MOCs
Kahn Process Networks
Static Dataflow, etc.

Methodology-Specific Channels
Master/Slave Library
etc.

Elementary Channels
Signal, Timer, Mutex, Semaphore, Fifo, etc.

Core Language
Modules
Ports
Processes
Interfaces
Channels
Events

Data Types
Logic Type (01XZ)
Logic Vectors
Bits and Bit Vectors
Arbitrary Precision Integers
Fixed Point Numbers
C++ Built-In Types (int, char, double, etc.)
C++ User-Defined Types

C++ Language Standard

Summary

- SystemC is a C++ based modeling environment
  - Powerful constructs for system-level design
  - Full RTL capabilities
- Common language infrastructure for
  - All levels of abstraction
  - For both hardware and software
- Based on ANSI-standard C++ - not a new language
- Future Releases (www.systemc.org)