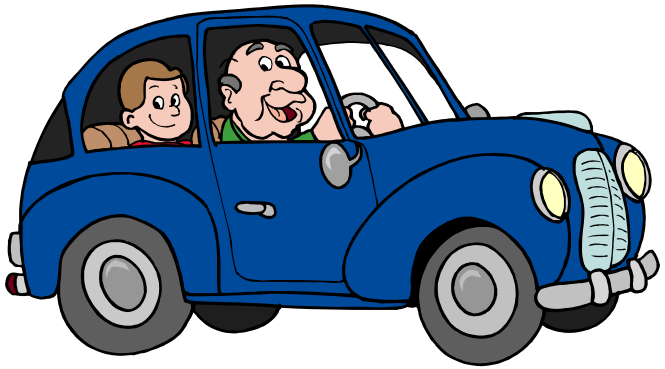
Communication

P. Marwedel*

* Partially using slides prepared by Tatjana Stankovic from the University of Nis (Serbia and Montenegro), visiting the University of Dortmund under the TEMPUS program.

Gas station example (1)



Fill'er up!

Customer 1

Filled!

Customer 2

Gas station example (2)

```
//BEGIN main.cpp
//See gas_station.h for more information
#include <systemc.h>
#include "gas_station.h"
unsigned errors = 0;
char* simulation_name = "gas_station";
int sc_main(int argc, char* argv[]) {
  sc_set_time_resolution(1,SC_NS);
  sc_set_default_time_unit(1,SC_NS);
  gas_station
Charlies("Charlies",/*full1*/10,/*full2*/12,/*filltime*/1.5,/*maxfills*/10);
  cout << "INFO: Starting gas_station simulation" << endl;
  sc_start();
  cout << "INFO: Exiting gas_station simulation" << endl;
  cout << "INFO: Simulation " << simulation_name
       << " " << (errors?"FAILED":"PASSED")
       << " with " << errors << " errors"
       << endl;
  return errors?1:0;}
```

```
#ifndef GAS_STATION_H ..
#include <iostream>
using std::cout; using std::endl;
#include <string>
#include <systemc.h>
SC_MODULE(gas_station) {   // Local module data
  const    sc_time t_MIN;   bool     m_filling; // state of attendant
  double   m_full1, m_full2;   double   m_filltime; sc_event e_request1, e_request2;
  double   m_tank1, m_tank2; unsigned m_count1, m_count2, m_maxcount;
  sc_event e_filled;   // Constructor
  SC_HAS_PROCESS(gas_station);
  gas_station(    sc_module_name  _name,
    double   full1=10.0,     double   full2=11.1,     double   filltime=1.8,     unsigned maxcount=5    ) :
    sc_module(_name), m_full1(full1), m_full2(full2), m_filltime(filltime), m_tank1(full1),
    m_tank2(full1), m_count1(0), m_count2(0),     m_maxcount(maxcount),     m_filling(false),
    t_MIN(1,SC_NS) // treat 1 minute = 1 nanosecond
  { cout << "INFO: Gas station named \"" << name() << "\"" << endl;
    cout << "INFO:   Customer1 has " << m_full1 << " gallon tank." << endl;
    cout << "INFO:   Customer2 has " << m_full2 << " gallon tank." << endl;
    cout << "INFO:   Attendant takes " << m_filltime << " minutes per gallon." << endl;
    cout << "INFO:   Maximum of " << m_maxcount << " fills per customer." << endl;
    SC_THREAD(customer1_thread);       sensitive(e_filled);
    SC_THREAD(customer2_thread);
    SC_METHOD(attendant_method);  sensitive << e_request1 << e_request2;  dont_initialize();
  }//endconstructor gas_station   // Declare processes
  void customer1_thread(void);   void customer2_thread(void);   void attendant_method(void);
std::string hms(void); }; // Helper methods
#endif
```

Gas station example (4)

```
//BEGIN gas_station.cpp (systemc)
#include "gas_station.h"
using std::cout;
using std::endl;
extern unsigned errors;
void gas_station::customer1_thread(void) {
  for (;;) {   // Simulate gas tank emptying time
   wait((m_full1+rand()%int(m_full1*0.10))*t_MIN);
    // Force 25% of all fill ups to be simultaneous
    // with other customers to check contention
    if (rand()%4==1) wait(e_request2);
    cout << "INFO: " << name() <<
   " Customer1 req. gas (1) at "  << hms() <<
endl;
    m_tank1 = 0;
    // Request fill up and then wait for
acknowledge
    do {  e_request1.notify(); // I need fill up! (2)
      wait(); // static sensitivity Somebody got filled
    } while (m_tank1 == 0); // Was it us?  yes
  }//end forever
}//end customer1_thread()
```

```
void
gas_station::customer2_thread(void) {
  for (;;) {  // Simulate emptying time
   wait((m_full2+rand()%
   int(m_full2*0.10))*t_MIN);
    cout << "INFO: " << name()   << "
    Customer2 needs gas (1) at " <<
    hms() << endl;
    m_tank2 = 0;
      do { e_request2.notify(); // fillup! (2)
      wait(e_filled); // dynamic sensitivity
    } while (m_tank2 == 0);
  }//endforever
}//end customer2_thread()
```

```
void gas_station::attendant_method(void) {
  // ASSERTION: We got here due to either (A) a request in progress
  // (B) an event request from a new customer. Since this is an SC_METHOD,
  // we maintain a small amount of state, m_filling. Initially, we're not filling.
  // Once we get a fillup request, we choose who, initiate filling, and then
  // use dynamic sensitivity to delay by the amount of time it takes to fill the
  // indicated gas tank.
  if (!m_filling) {
    // Check customer 1 first (preferential selection)
    if (m_tank1 == 0 && m_count1 < m_maxcount) {
      cout << "INFO: " << name()
           << " Filling tank1 (3) at "
           << hms() << endl;
      next_trigger(m_filltime*m_full1*t_MIN);
      m_filling = true;
    // Check customer 2 only if no customer 1
    } else if (m_tank2 == 0 && m_count2 < m_maxcount) {
      cout << "INFO: " << name()
           << " Filling tank2 (3) at "
           << hms() << endl;
      next_trigger(m_filltime*m_full2*t_MIN);
      m_filling = true;
    }//endif
  } else {
```

```
 } else {
    // We reach here by timing out on filling the tank, so first update
    // the tank, counts and issue messages about this event for the
    // appropriate customer. Then notify everyone of the event (4)
    if (m_tank1 == 0 && m_count1 < m_maxcount) {
      m_tank1 = m_full1; m_count1++;
      cout << "INFO: " << name()  << " Filled tank1 (4) at "    << hms() << endl;
    } else if (m_tank2 == 0 && m_count2 < m_maxcount) {
      m_tank2 = m_full2;
      m_count2++;
      cout << "INFO: " << name() << " Filled tank2 (4) at " << hms() << endl;
    }//endif
    e_filled.notify(SC_ZERO_TIME); // We finished filling (4) & are available!
    m_filling = false; // go back to waiting
    // See if we need to stop the simulation
    if (m_count1 == m_maxcount && m_count2 == m_maxcount) {
      cout << "WARN: " << name()
           << " No more fuel at "
           << hms() << endl;
      sc_stop();
    }//endif
  }//endif
}//end attendant_method()
```

Gas station example (7)

```
#include <sstream>
std::string gas_station::hms(void) {
  std::ostringstream now;
  double mins(sc_simulation_time());
  unsigned days = int(mins/(24*60));
  mins -= days*24.0*60.0;
  unsigned hrs = int(mins/60);
  mins -= hrs*60.0;
  if (days)          now << days << " days ";
  if (days||hrs)   now << hrs  << " hrs ";
                       now << mins << " mins";
 return now.str();
}//end hms()
```

Limitations of gas station model

- Complicated way of finding out, which customer has been served.
  e.g.: what to do after
  wait (e_request1 | e_request2)
- Events cannot be used to distribute information
- No mechanism available for accessing shared resources

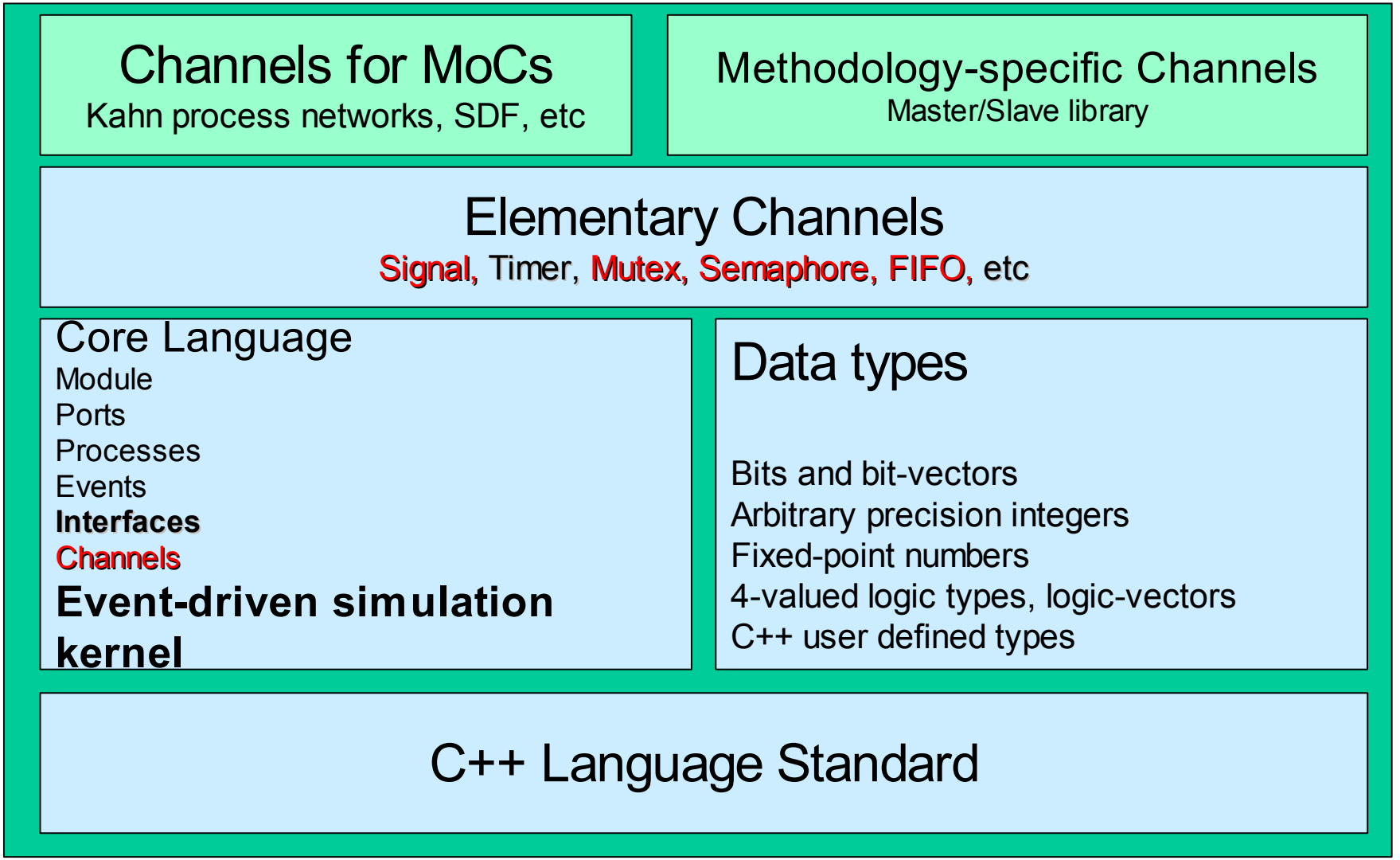
☞ Channels

  - Primitive channels
    Inherited from base class **sc_prim_channel**, include **sc_mutex**, **sc_semaphore**, and **sc_fifo.**
  - Hierarchical channels

Contents

- Introduction
- Data types
- A Notion of Time
- Modules
- Concurrency
- Structure
- Communication, Channels
- Ports & Interfaces
- Advanced Topics

SystemC language architecture



Mutual exclusion: sc_mutex

Global resources can be protected with mutexes.
Mutex associated with resource to be protected.
SystemC provides mutex methods for **sc_mutex** channels.

Syntax:

sc_mutex name; // declare mutex
name.**lock**(); // blocking: waiting until mutex is unlocked
name.**trylock**() // nonblocking: true if success, else false
name.**unlock**() // free locked mutex

No signal generated if resource is freed (wakeup??)

SC_METHOD cannot use **.lock**(), since it is blocking.

sc_mutex: Examples

```
sc_mutex drivers_seat; …
car->drivers_seat.lock();     // lock seat
car->start(); …
car->stop();
car->drivers_seat.unlock(); // unlock seat
```

```
class bus {
 sc_mutex bus_access;
 void write(int addr, int data) {
   bus_access.lock();
  // perform write
  bus_access.unlock();
} … }; //endclass
```

Semaphores

Sometimes, ≥ 1 copies of global resources are available.
Can be modeled with class **sc_semaphore**.
Number of copies is part of the declaration.

Syntax:

sc_semaphore name(count); // declare semaphore
name.**wait**(); // blocking: waiting until available
name.**trywait**() // nonblocking: true if success
name.**get_value**() // return number from semaphore
name.**post**() // free semaphore

SC_METHOD cannot use **.wait**(), since it is blocking.

sc_semaphore: Examples (1)

Example: Gas station with 12 pumps

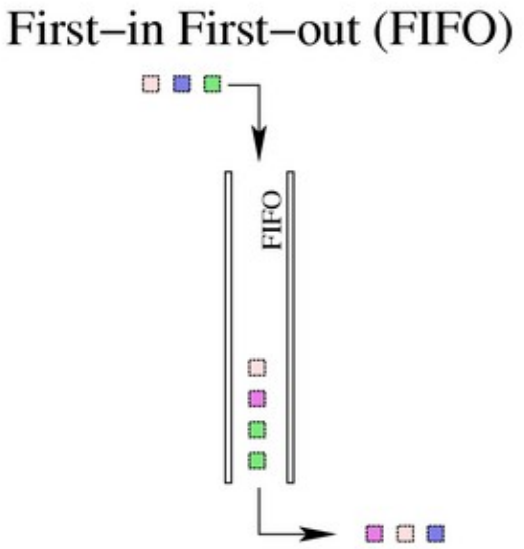
```
SC_MODULE (gas_station) {
 sc_semaphore pump(12); // 12 pumps
  void customer1_thread {
   for (;;) { // wait till tank empty
      … // find an available gas pump:
      pump.wait();
       // fill and pay
      pump.post();
}};
```

Modeling a multiport memory

```
class multiport_RAM {
  sc_semaphore read_ports(3);
  sc_semaphore write_ports(2);
  ..
  void read(int addr, int& data) {
  read_ports.wait();
  //perform read
  read_ports.post();
  }
  void write(int addr, int data) {
  write_ports.wait();
  //perform write
  write_ports.post();
  }…};//endclass
```

Communication by FIFOs

In the early design stages, STL's unbounded **deque** (double ended queue) may be sufficient.
More details available with **sc_fifo<>**.
Default depth: 16;
Data type needs to be specified.



Syntax & methods:

sc_fifo<*element_typename*> *name*(*size*);
name.**write**(*value*);
name.**read**(*reference*);
…= *name*.**read**(); // function style

Potentially blocking reads and writes.

Communication by FIFOs (2)

Syntax & methods (cont'ed):

- **if** (*name*.**num_available**()==0)
  wait(*name*.**data_written_event**());

  Allows test before waiting for next element to be read.

- **if** (*name*.**num_free**()==0)
  next_trigger(*name*.**data_read_event**());

  Allows test before waiting for next element to be written.

Kahn process networks

Assuming finite length is no problem:

```
SC_MODULE(kahn_ex) { …
 sc_fifo<double> a, b, y; …
// Constructor
kahn_ex::kahn_ex(): a(19), b(10), y(20) {…}
void kahn_ex::addsub_thread() {
  for (;;) {
     y.write(kA*a.read() + kB*b.read());
     y.write(kA*a.read() - kB*b.read());
  } //end forever
}
```

Better performance if storing pointers, not structures;
shared_ptr<> from GNU Boost library recommended.

Deterministic models of synchronous hardware

sc_signal (and **sc_buffer**) correspond to signals in VHDL: they use the *evaluation-update* paradigm:

- Every channel has 2 storage locations:
  - current value and
  - new value.
- Writes are updating new value,
- Reads are reading current value
- **request_update**(): called by **write**(), causes kernel to call **update**() during the update phase, for each channel that requested an update
- **update**() copies new value to current value, but may also resolve contention or notify events.

Evaluate-update cycle enables deterministic communication.

sc_signal

Syntax:

- **sc_signal** <*datatype*> *signame* [,*signame*] ..;

- *signame*.**write**(*newvalue*)

  write includes evaluate phase of behavior +
  call to protected **sc_prim_channel::request_update**();

  Call to **sc_signal::update**() is hidden,
  occurs in update phase due to **request_update;**

  Within each δ–cycle, only a single process can write to channels of type **sc_signal**, last value being retained.

- *signame*.**read**(*varname*)

sc_signal: Example

```
int                          c;
sc_signal<sc_string> sig;
// initialization during 1st delta cycle
sig.write("Hello");
c=1;
cout << "c: " << c << " "
        << "sig:" << sig << endl;
Wait(SC_ZERO_TIME);
// 2nd delta cycle
c=2;
sig.write("World");
cout << "c: " << c << " "
        << "sig:" << sig << endl;
wait(SC_ZERO_TIME);
// 3rd delta cycle …
```

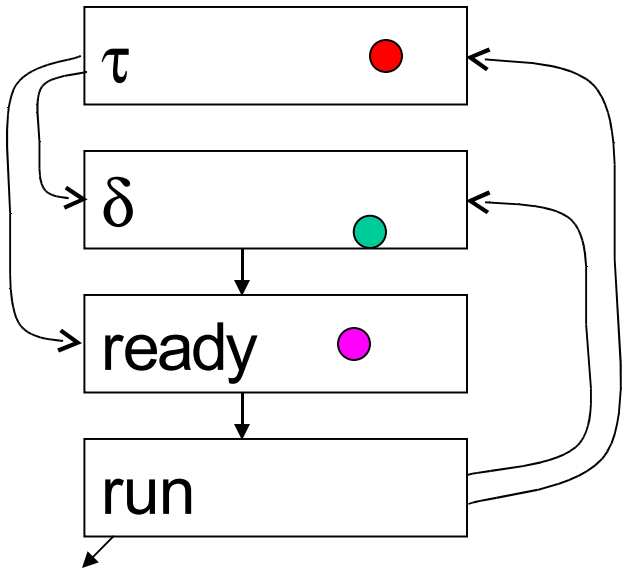
Output:

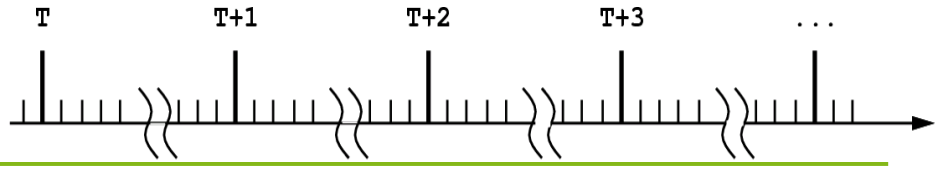
c: 1 sig: ' '
c: 2 sig: 'Hello'

Recommended suffix _sig for signals to indicate delayed update.

Control flow representation

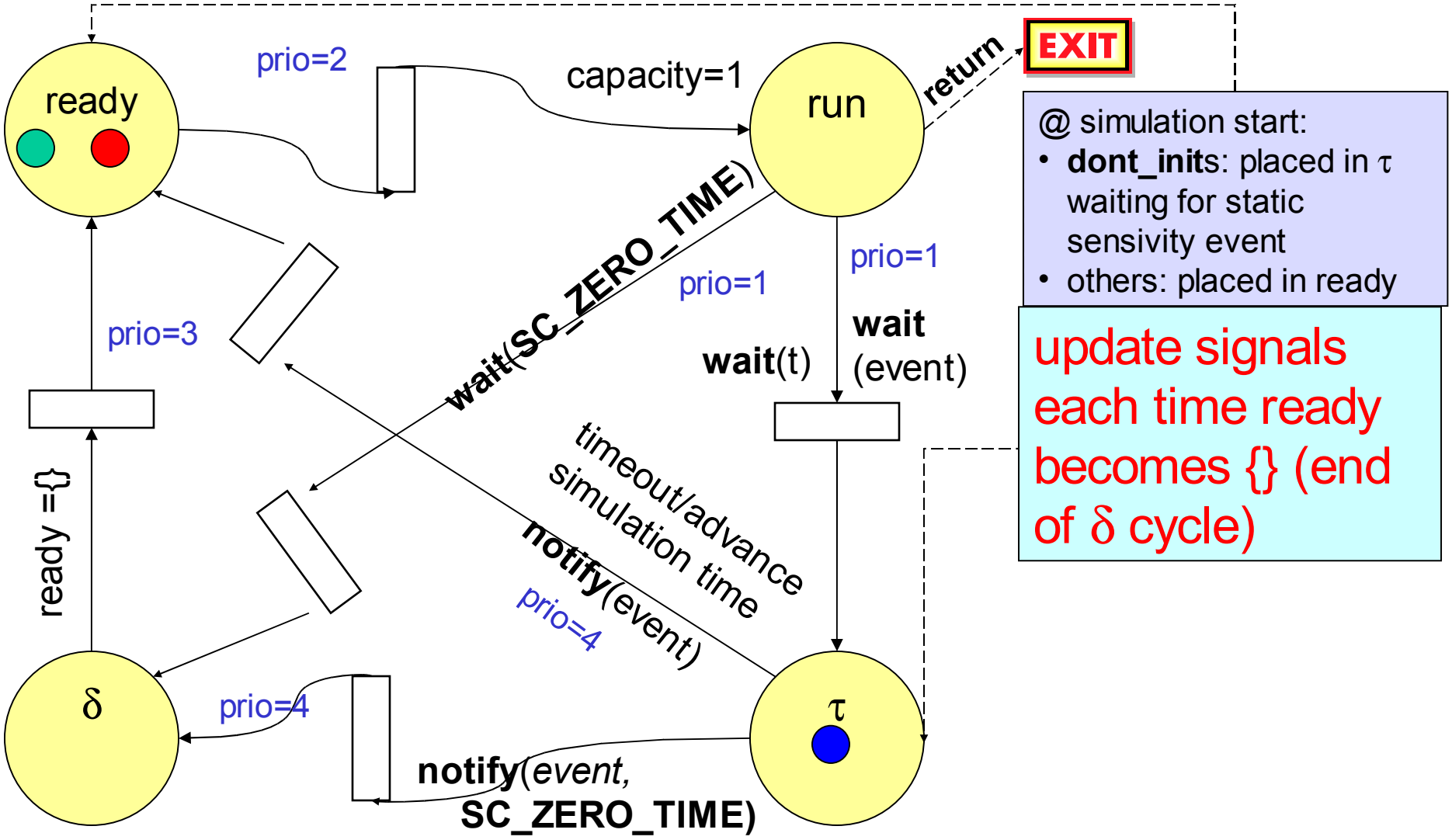
While there is a process in the τ pool // macroscopic time loop
  While there is a process in the "δ" pool // δ time loop
    While there is a process in the ready pool // same δ time
      {take any process
      execute: evaluate signal changes;
      may send event notifications
        - immediate (→ put into ready pool),
        - delayed (→ "δ" pool, with 0 time)
        - timed (→ waiting pool, with time)
      until process completes (via **return**) or
      suspends (via calls to **wait**()); process to
      pool}
    Update pending signal changes;
    processes waiting in "δ" pool → ready pool;
  If ∃ process ∈ τ pool: advance time





Transitions between thread states

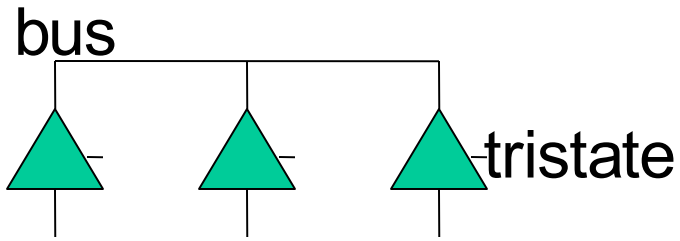
- Predicate/transition net model ≈ activity chart → queuing model -



Dangerous overloading of =

varname = *signame*.**read**();
signame = *newvalue*;
varname = *signame*;
are also legal, but dangerous, since they hide the evaluate-update cycle.

Multiple writes in a δ-cycle



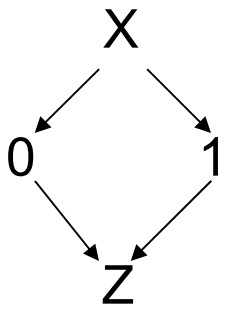
Syntax:

sc_signal_resolved *name*;

sc_signal_rv<*width*> *name*; //*

Semantics same as for **sc_signal<sc_logic>**, except that multiple writes are permitted in a δ-cycle. Resolution is predefined as follows:

A\B	'0'	'1'	'X'	'Z'
'0'	'0'	'X'	'X'	'0'
'1'	'X'	'1'	'X'	'1'
'X'	'X'	'X'	'X'	'X'
'Z'	'0'	'1'	'X'	'Z'



Changing the resolution function is a little awkward (see Black & Donovan)

* **rv** means: resolved vector

Template specializations of sc_signal<bool> and sc_signal<sc_logic>

Signals of specializations **sc_signal<bool>** and **sc_signal<sc_logic>** support the following extensions:

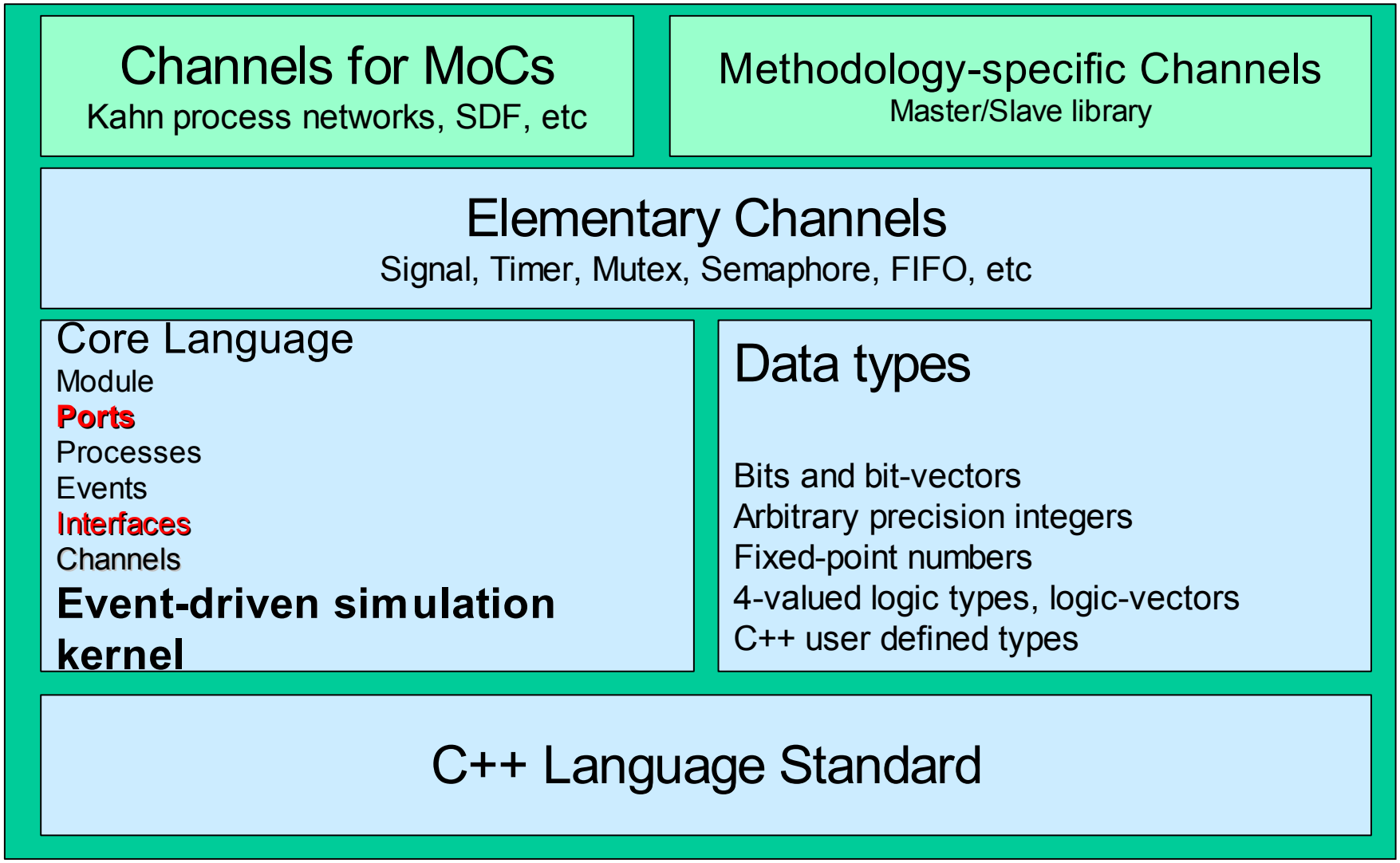
Syntax:

```
sensitive << signame.posedge_event()
          << signame.negedge_event();
wait(signame.posedge_event() |
     signame.negedge_event());
if (signame.posedge_event() |
    signame.negedge_event()) {…
```

Semantics:

posedge is any transition to '1',
negedge is any transition to '0'.

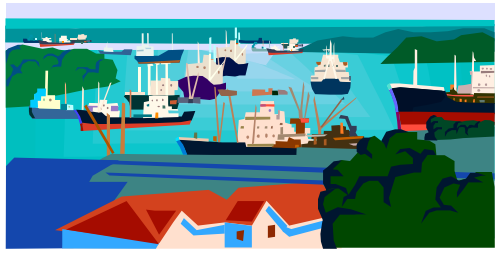
SystemC language architecture



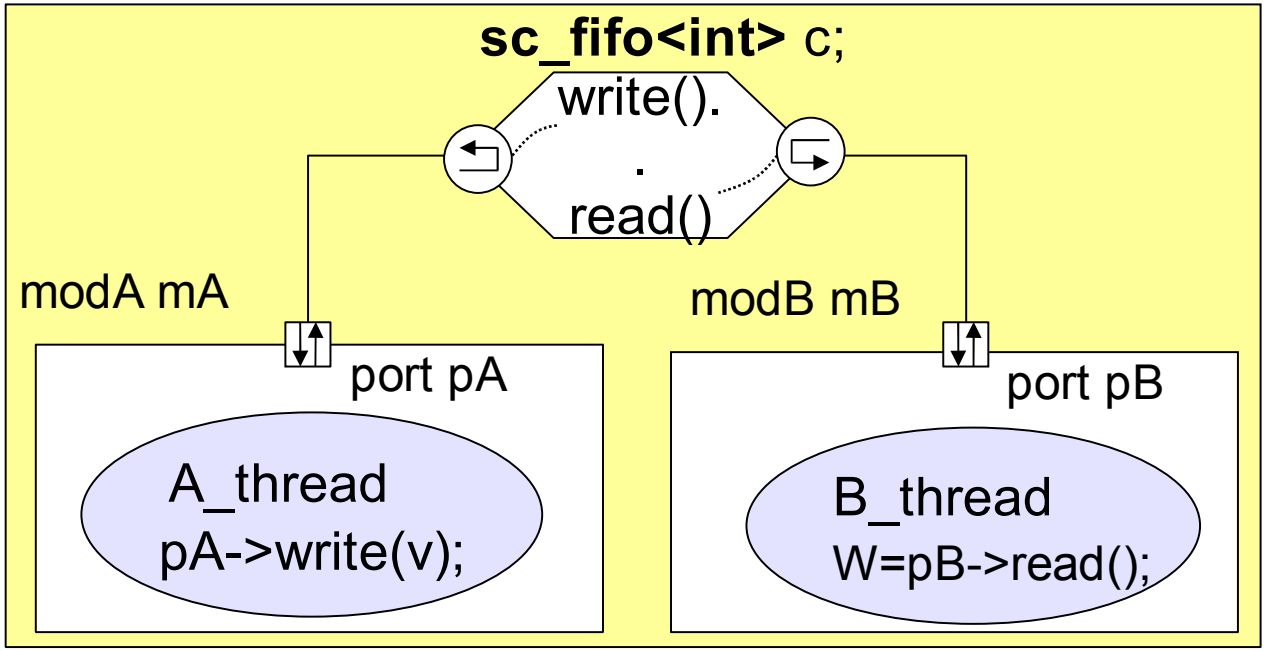
Ports

How to communicate between local modules?

- Implicit interfaces such a global variables should be avoided
- Ports provide the well-defined boundaries through which modules interact
- Ports are connected to **channels** via **interfaces.** These provide the required encapsulation.
- Ports are “kind of” pointers to channels.



Initial example



"A_thread in module modA communicates a value contained in local variable v by calling the **write** method of the parent module's channel c."

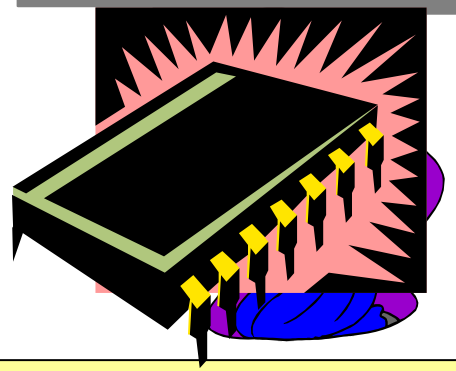
A_thread does not need to use anything but the **write** method.

Separation of concerns enabled by ☞ *interfaces.*

Using virtual methods and polymorphism in C++

```
struct multiport_memory_arch: public my_interface {
  virtual void write(unsigned addr, int data) {
    mem[addr] = data;
  } // end write
  virtual int read(unsigned addr) ) {
    return mem[addr];
  } // end read
  private:  int mem[1024];
};
```

Oops!
Memories!



2 Memory models

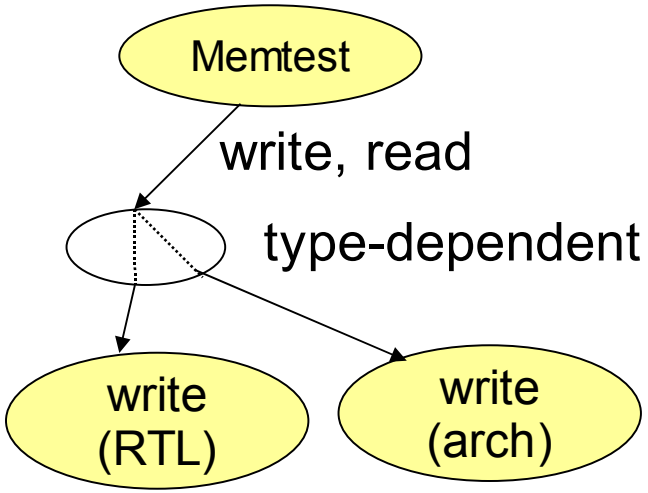
```
struct multiport_memory_RTL: public my_interface {
  virtual void write(unsigned addr, int data) {
    // complex details of memory write
   } // end write
  virtual int read(unsigned addr) ) {
 // complex details of memory read
  } // end read
  private:   // complex details of memory storage
};
```

Application to memory testing

```
void memtest(my_interface mem) {
        //complex memory test
}

multiport_memory_arch fast;
multiport_memory_RTL slow;
memtest(fast);
memtest(slow);
```

Call graph



The same memory test method will use different levels of precision for the memory model.

Interfaces in C++

Abstract classes = Classes never used directly.
Abstract classes usually only contain pure virtual functions.
Example:

```
struct My_interface {
 virtual T1 My_methA(…)=0;
 virtual T2 My_methB(…)=0;
};
```

Polymorphism: calling My_methA results
- in call to My_Derived1::My_methA if object is of type My_Derived1
- and in call to My_Derived2::My_methA if object is of type My_Derived2

```
class My_Derived1
: public My_Interface{
  T1 My_methA(…) {…}
  T2 My_methB(…) {…}
private:
  T5 my_data1;
};
```

```
class My_Derived2
: public My_Interface{
  T1 My_methA(…) {…}
  T2 My_methB(…) {…}
private:
  T7 my_data4;
};
```

Classes containing no data members and only pure virtual methods are called **interface classes.**

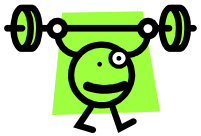
Definition of interfaces and channels

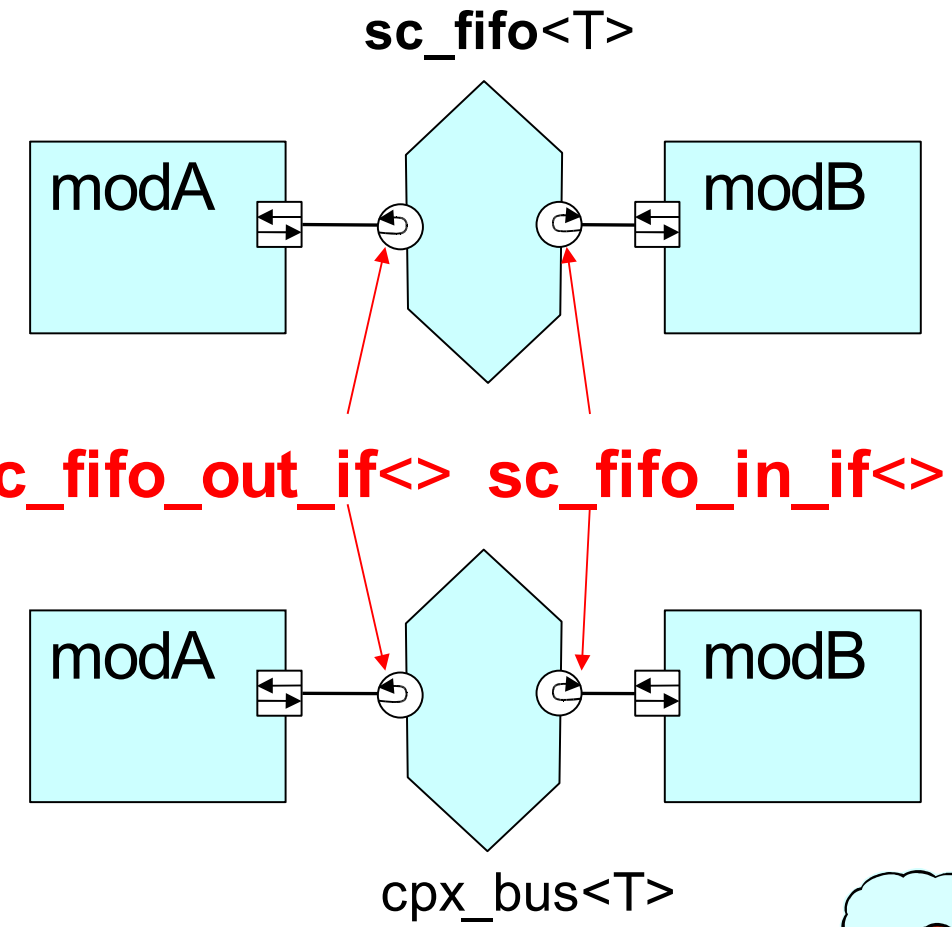
SystemC interface: abstract class inheriting from **sc_interface,** providing only pure virtual declarations of methods referenced by SystemC channels and ports.
No implementations or data are provided in an interface.

SystemC channel: class implementing ≥ 1interface classes, inheriting from **sc_channel** or **sc_prim_channel**.
Channel implements all methods of inherited interface classes.

By using interfaces to connect to channels, we can implement modules *independently* of the implementation details of the communication channels.

Power of interfaces





In one design, modules are connected via a FIFO, in a second design with a complex bus. If the interfaces, in this case **sc_fifo_out_if**<> and **sc_fifo_in_if**<> remain the same, no change to modA and modB is required if we replace the channel.

Key for re-use of intellectual property (IP).

Simple SystemC Port Declarations

Definition: A **SystemC port** is a class templated with and inheriting from a SystemC **interface**.
Ports allow access of channels across module boundaries.

Syntax:
sc_port<*interface*> *portname*; //used @module class definition

Example:
```
SC_MODULE(stereo_amp) {
 sc_port<sc_fifo_in_if<int> > soundin_p;
 sc_port<sc_fifo_out_if<int> > soundin_p;
 …};
```

Do no omit blank!

Summary

- Gas station example
- Channels
  - **sc_mutex,**
  - **sc_semaphore**
  - **sc_fifo,**
  - **sc_signal,**
  - ∀ δ-cycles,
  - **sc_signal_resolved**
- Polymorphism & Interfaces
- Ports (1)