Projeto de Hardware Orientado a Objetos: a abordagem *SystemC*

Márcio Kreutz

Introduction

- Modeling and simulation tool
- Key idea is to bring together software and hardware engineers
- Different abstraction levels
  - Transaction
  - Register transfer
- Widely accepted language
  - C++
- Fast learning curve
- Basic RT types
  - Signals, bit-vector

Features

- Object Oriented compliant
- Metalanguage
  - Classes library; API-based modeling
- General purpose modeling language and simulation
  - “core language” used to support:
    - Methodologies, models of computation and abstraction levels
- Interface-based design
  - Ports, channels
- Processes or Threads execution
- Simulation engine fully integrated within the language syntax
- Code-compiled architectural component simulation

Language

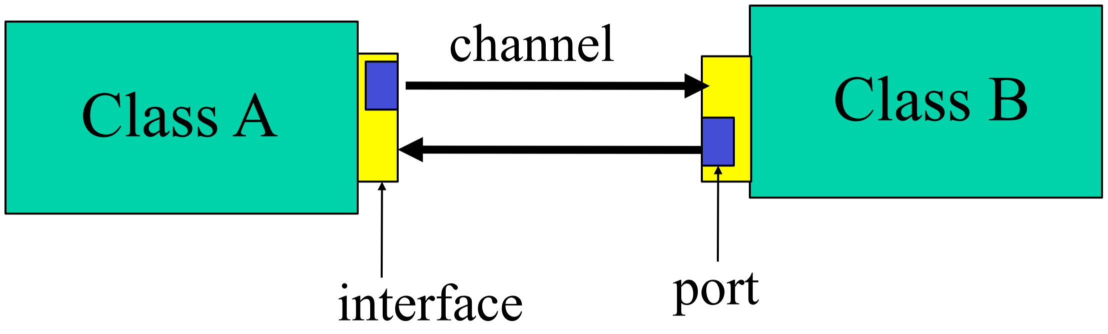
- Same data types as “C”
- No type checking
- Operators
- Multiple inheretance allowed
- Polymorphism
  - Operator, method overloading

Interface-based Design Concept

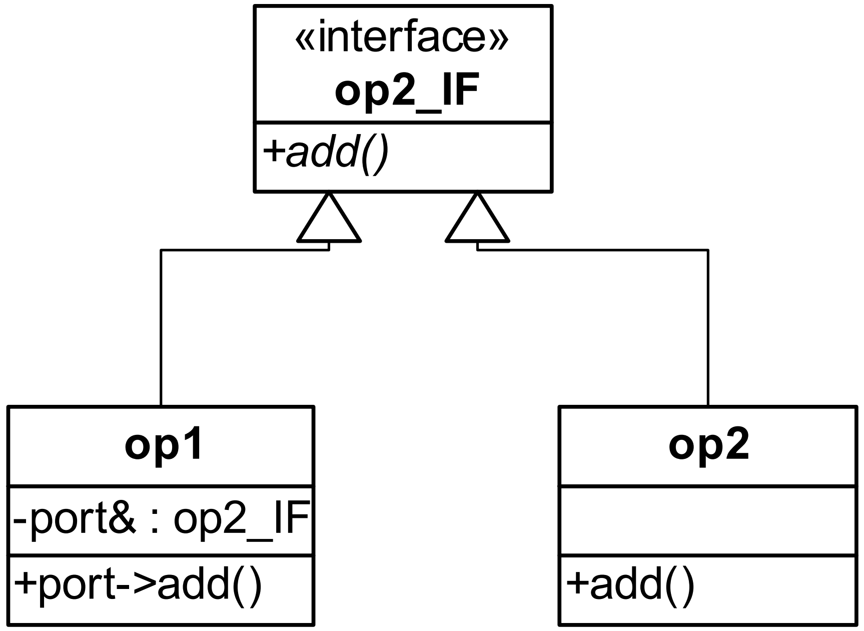
- Interfaces:
  - Virtual classes
  - Expose external methods
  - Accessible by other entities

Interface-based Design Implementation

- Virtual methods implemented as *channels*
- Binded to ports



Interface-based Design
Classes Diagram

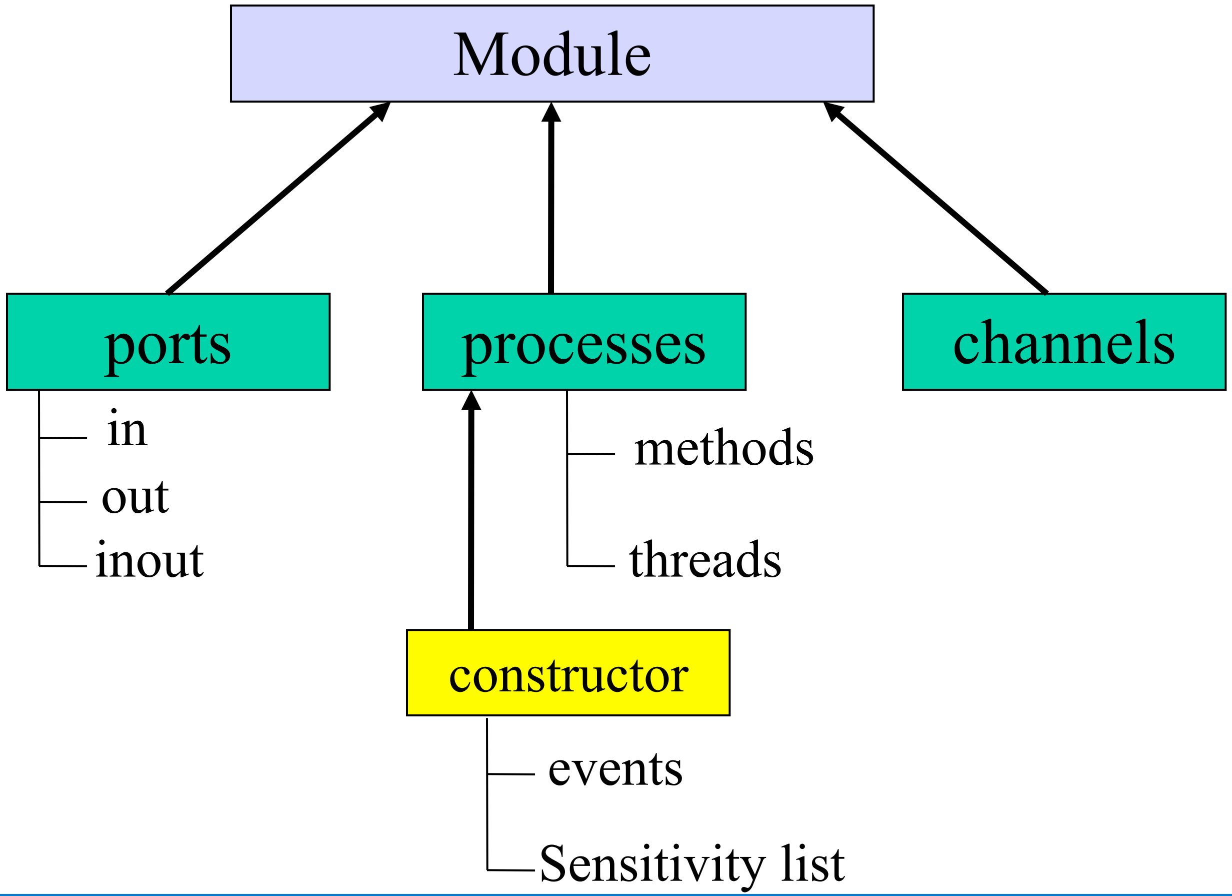


- port->add()
    - communication through channel “op2”

Simulator Syntax

- Integrated in the language
- SC_MODULE
  - Contains methods activated by “events”
  - Can be hierarchical
- Ports declaration
- Constructor
  - CTOR
- Module processes
  - SC_METHOD (method process)
    - Suspended by *next_trigger()*
  - SC_THREAD (thread process)
    - Own thread of execution
    - Suspended by *wait()*
- Events
  - sensitive << clock.pos();

Structure of a SC program



Porting C++ to SystemC

```
#include<systemc.h>

class adder : public sc_module {
        int a, b, c;

public:

        sc_in<bool> clock;
        sc_out<int> out;

        void add();

        adder (sc_module_name name, int _a, int _b) : sc_module(name) {

          SC_THREAD(add);
```

C++ to SC

```
#include<systemc.h>

SC_MODULE (adder ) {
        int a, b, c;

public:

        sc_in<bool> clock;
        sc_out<int> out;

        void add();

        SC_CTOR ( adder ){

            SC_THREAD(add);

            sensitive << clock.pos();

            a= _a; b= _b;
        }
};
```

Execution model

- Co-routine
- Processes have their own threads
- Kernel does not preempt threads
  - Only *wait()* method does
- Code between two *wait()* statements seems to execute at the same time
  - Simulation time advances only when the *wait()* method is called

Simulator Scheduler

- Process-based, non-preemptive scheduling
- *Events* “resume” processes
- User can create their own *events*
  - *sc_event* type
- Time grain is defined by the designer
  - *sc_set_time_resolution()*
    - Default: 1 picosecond: 64 bit integer time data type
- Time advances with the *wait()* method

sc_event type

- Processes can be made “sensitive” to events
- An event executes on time by *notify()*
- Processes are activated automatically by sensitive events
- Events are implemented in *channels* and binded to processes through *ports*
  - sensitive << port.get_event();

Scheduler Algorithm

```
Process P[np]; // processes list

initialization() {
   run_all(P);  // no specific order
   // except "don't_initialize()"
   evaluation_step();
}

evaluation_step() {
int pr= 1; // processes ready
  while(pr)
    while(P[np++].ready()) {
      run(P[np]); // might notify events
      if (P[np].notify() == TRUE) ++pr;
      else --pr;
```

```
ev= find_next_event();
  if(ev == 0) sim_stop();
  else {
     advance_sim_time(ev.time());
     evaluation_step();
  }
}

update_step() {
   while(P[np++].ready())
      if(P[np].request_update() == TRUE)
              P[np].update();
}
```

The *wait()* method

- Suspends and resumes processes
- Static
  - wait(); // wait for the next event been notified
  - wait(e1); // wait for “e1” been notified
- Dynamic
  - wait(e1 & e2)
  - wait(e1 || e2)
  - wait(200, SC_NS)
  - wait(200, SC_NS, e1); // e1, timeout 200 ns

Non-determinism in SC

- No order for threads
- Use of primitive channels
  - Delta cycle synchronism

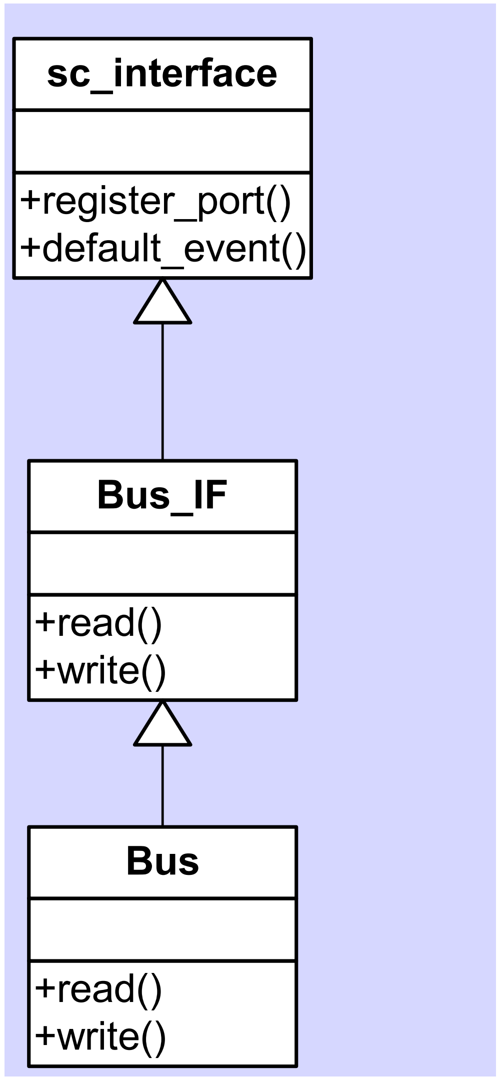
Time in SC

- *sc_set_time_resolution()*
- *sc_get_time_resolution()*
- *sc_time_unit*:
  - SC_FS: femtoseconds
  - SC_PS: picoseconds
  - SC_NS: nanoseconds
  - SC_US: microseconds
  - SC_MS: miliseconds
  - SC_SEC: seconds

Ports, Channels and Interfaces

- Entities communicate through *ports* and *channels*
- Ports are binded to channels at design time through *interfaces* (sc_interface)
  - *sc_port(), sc_channel()* and *sc_interface()*
- Static type checking through interfaces
- Modular design
  - Components reuse

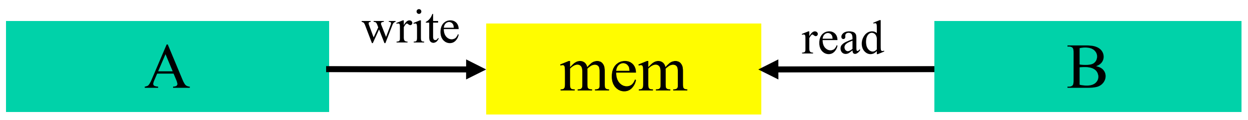
sc_interface



```
template<class T>
class Bus_IF : virtual public sc_interface {

public:
   virtual sc_event default_event() = 0;
   virtual T read() = 0;
   virtual void write(T) = 0;
};
```

Primitive Channels
sc_prim_channel

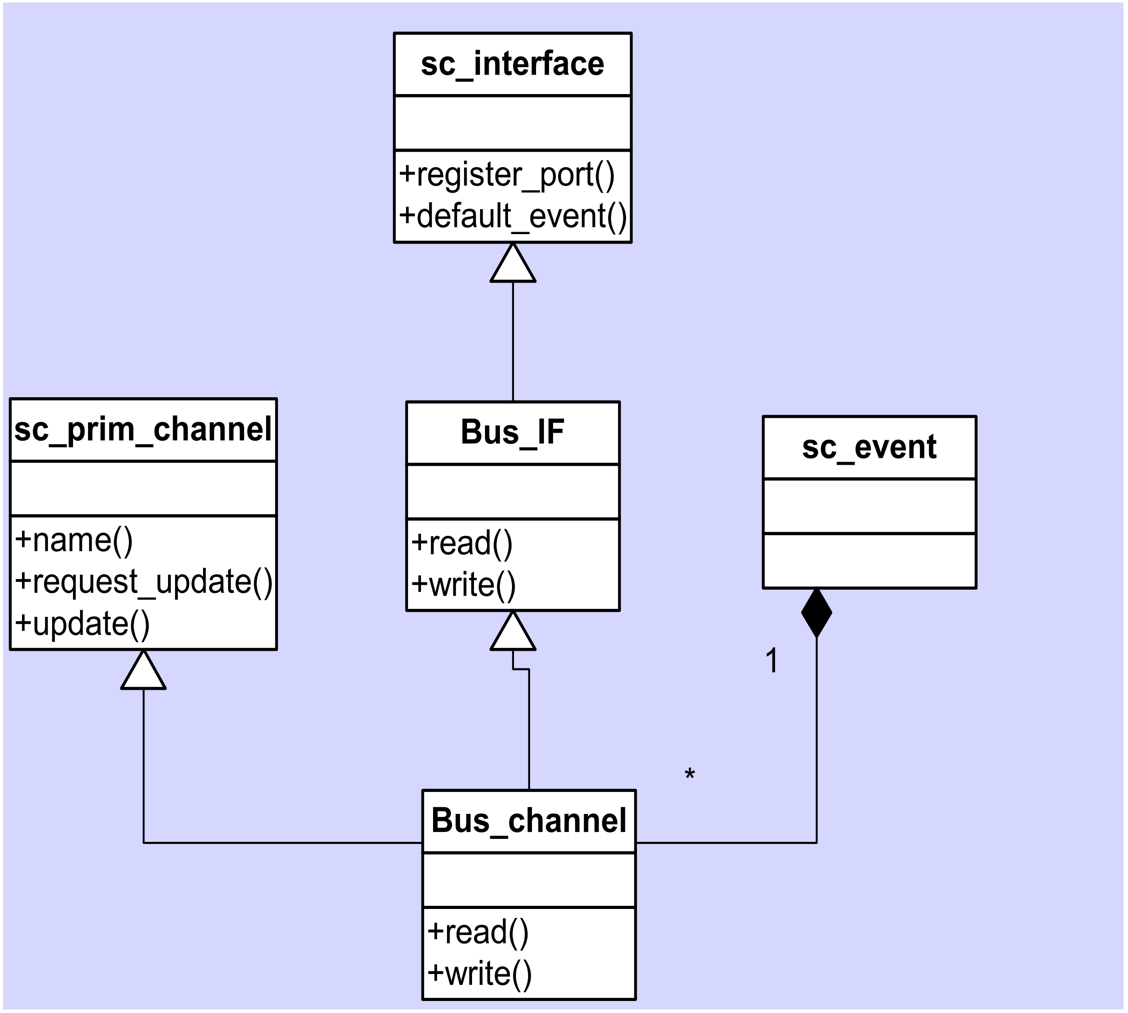
- sc_signal; sc_fifo; sc_mutex
- Execute on two steps
  - Evaluate
  - Update
    - Used when:
      - Simultaneous acesses have to be serialized
      - Arbitration
      - Determinism



sc_prim_channel steps

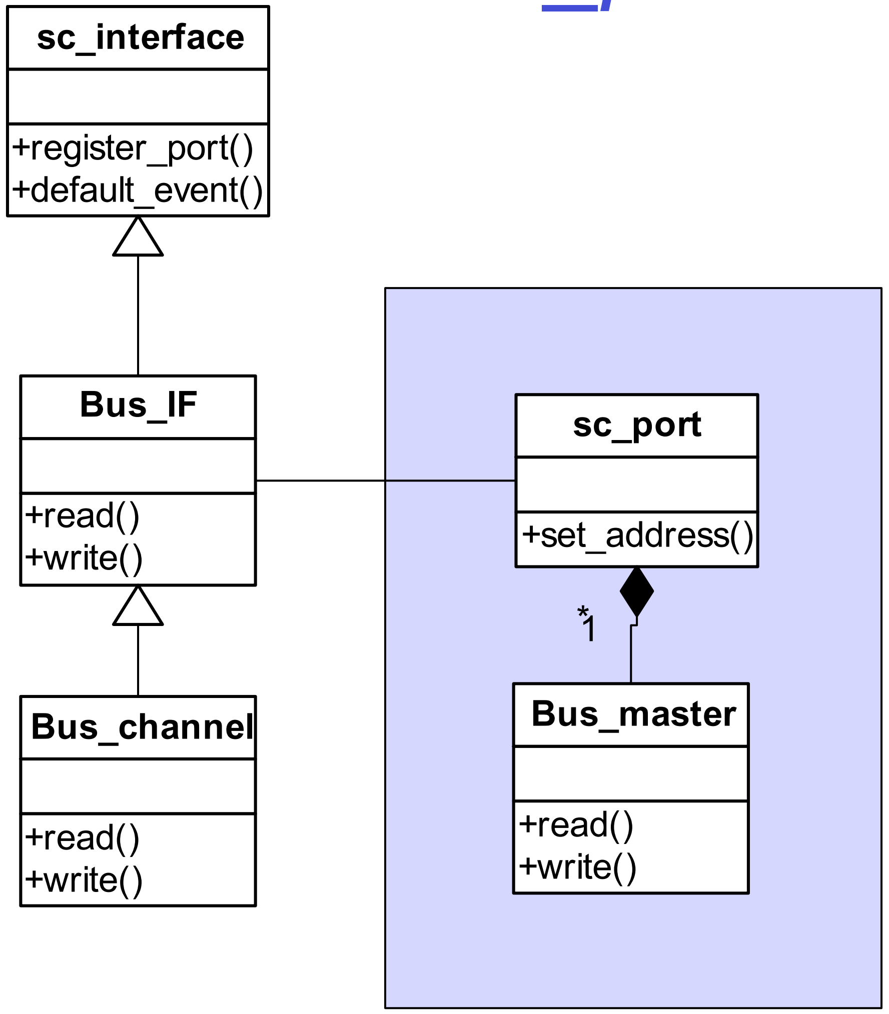
- Evaluate
  - Channel functionality
- Update
  - If *request_update()* was called during “evaluate” step:
    - Kernel calls *update()* method
- *Update()* execution
  - *notify() an event*
  - event.notify(SC_ZERO_TIME)
    - Next delta cycle

sc_prim_channel

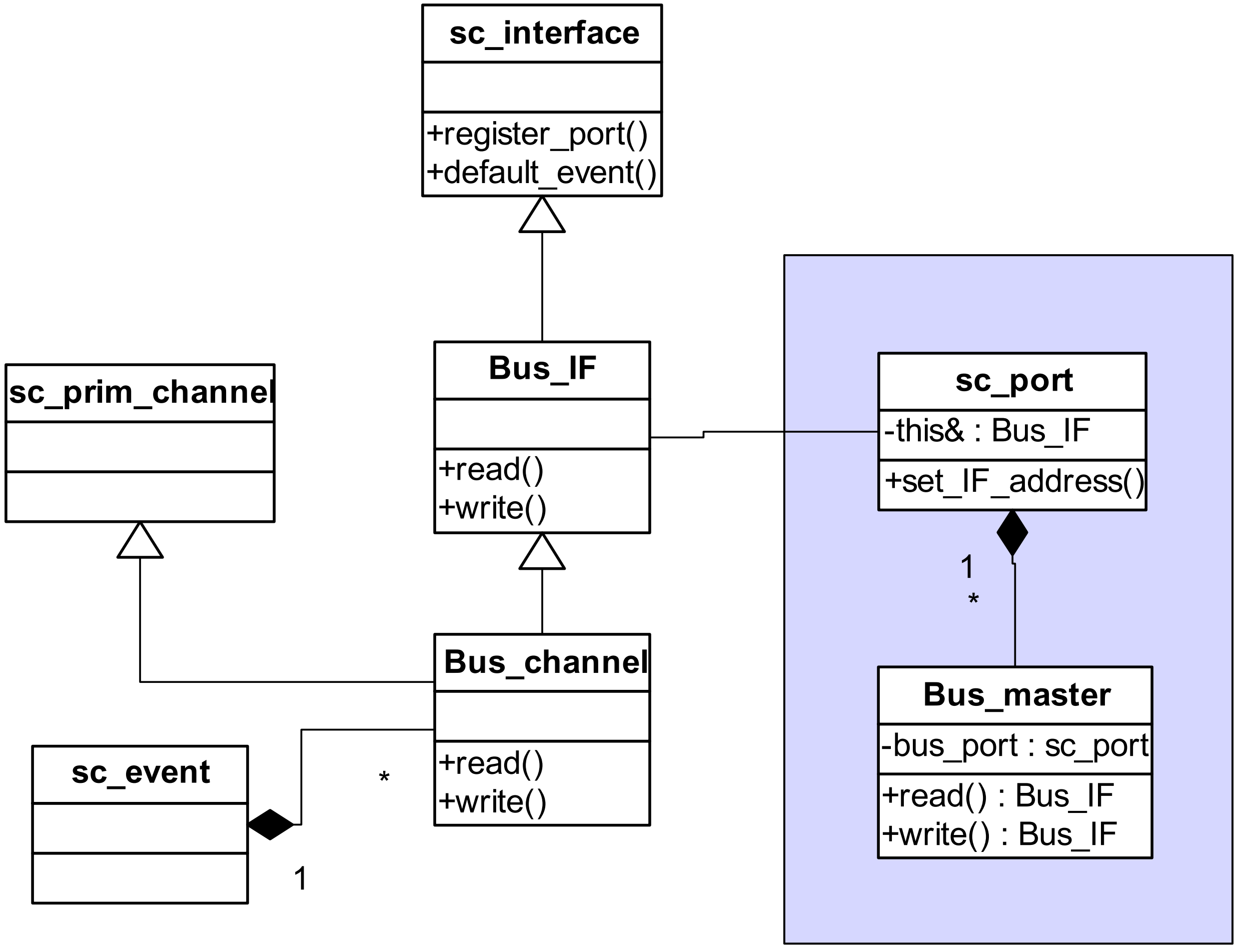


```
template<class T>
class Bus_channel :
        public Bus_IF<T>,
        public sc_prim_channel
{
T  v;
public:
   // IF methods
    sc_event default_event() {
          return(bus_event);
   }
   T read() { return (v); }
   void write(T value) {
       if(v != value) request_update();
       v= value;
   }
   void update() {   // next delta cycle
       bus_event.notify(SC_ZERO_TIME
   }
  sc_event   bus_event;
};
```

sc_port


sc_interface
+register_port()
+default_event()
Bus_IF
+read()
+write()
Bus_channel
+read()
+write()
sc_port
+set_address()
*
1
Bus_master
+read()
+write()

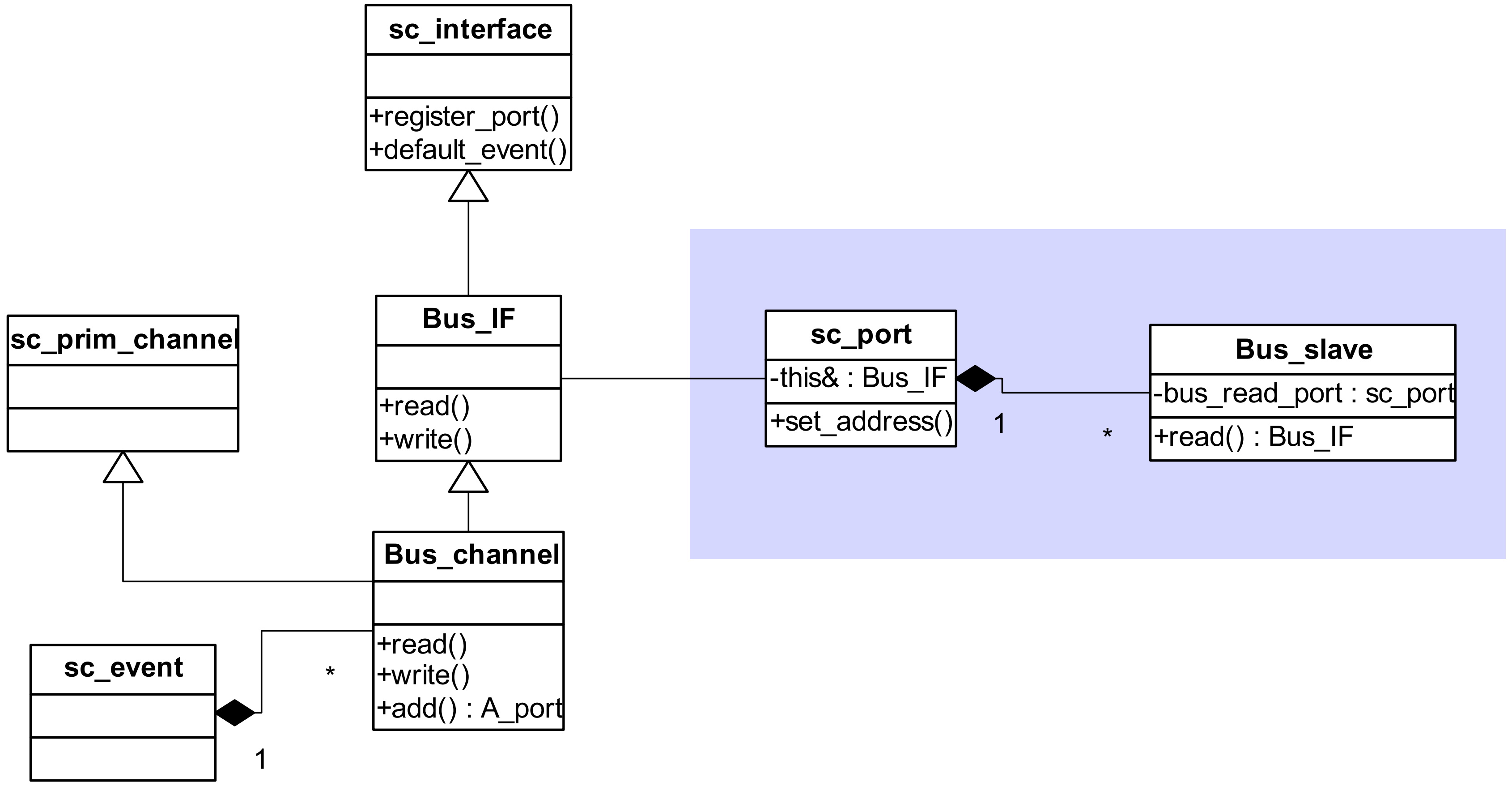
Writing …


sc_interface
+register_port()
+default_event()
sc_prim_channel
Bus_IF
+read()
+write()
sc_port
-this& : Bus_IF
+set_IF_address()
1
*
Bus_channel
+read()
+write()
sc_event
*
1
Bus_master
-bus_port : sc_port
+read() : Bus_IF
+write() : Bus_IF

Writing …

```
SC_MODULE( Bus_master ) {
  sc_port<Bus_IF<int>> bus_port;
  sc_in<Bus_slave_IF<int>> slave_request;

  sc_in_clk clk;

public :
  void write() {
    wait(); // wait on a slave
    bus_port->write(slave_request->get_value());
  }

  void check() { }  // check slaves;

  SC_CTOR( Bus_master ) {
      SC_THREAD( write );
```

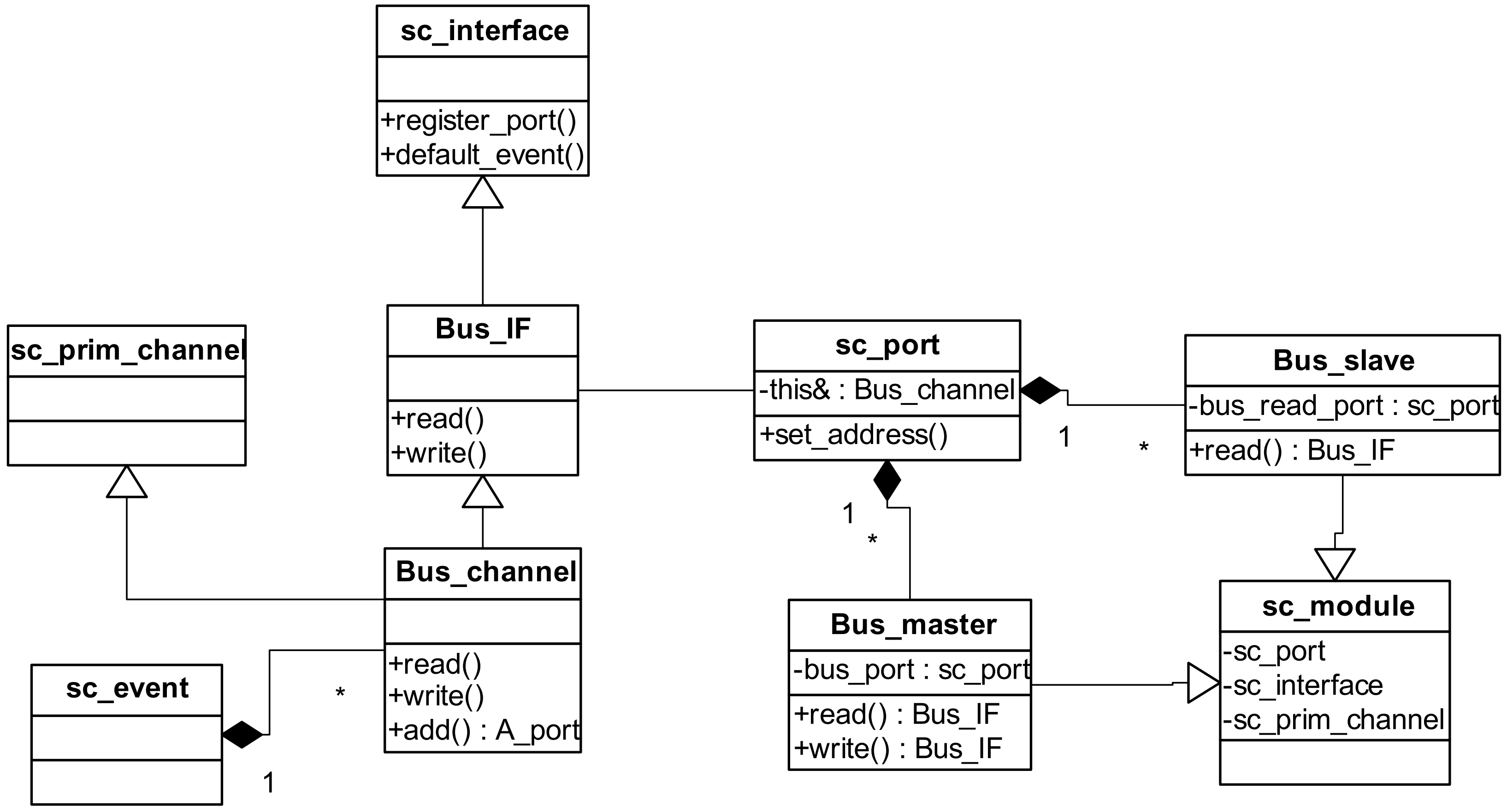
Reading ...


sc_interface
+register_port()
+default_event()
Bus_IF
+read()
+write()
sc_prim_channel
Bus_channel
+read()
+write()
+add() : A_port
sc_event
*
1
sc_port
-this& : Bus_IF
+set_address()
1
*
Bus_slave
-bus_read_port : sc_port
+read() : Bus_IF

Reading …

```
SC_MODULE( Bus_slave ) {
  sc_port<Bus_IF<int>> bus_read_port;
  int  v;

public :
  void read() {
      wait(); // wait until "Bus_master" writes a new value
      v =  bus_read_port->read();
  }

  SC_CTOR( Bus_slave ) {
    SC_THREAD( read );
    sensitive << bus_read_port->default_event();
  }
```

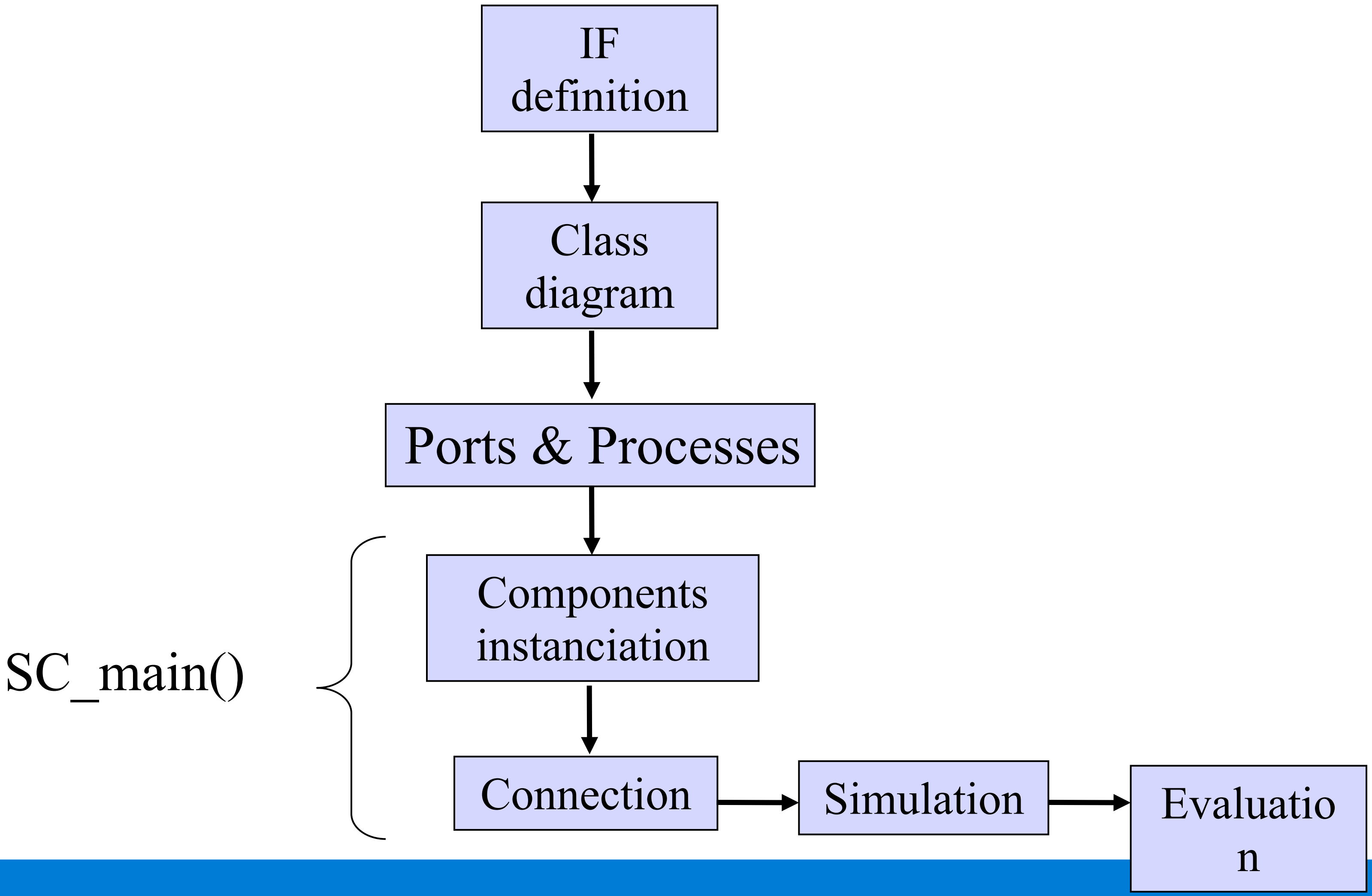
A Program Instance



A Program Instance

```
void sc_main() {

    Bus_channel bc;
    Bus_master bm;
    Bus_slave bs;

    sc_clock clk("clock",10,SC_NS);

    bm.bus_port(bc);
    bs.bus_read_port(bc);
    bus.clock(clk);

    sc_start(10000, SC_NS);
    return(0);
}
```

SystemC Design Flow



Extras

- Observing simulation results
  - *sc-trace()*
- Documentation, application notes and libraries/APIs to download
  - www.systemc.org

Remarks

- Models are modular by construction (OO features)
- Communication-driven modeling (interfaces)
- HW/SW engineers work together
- Use of a commom syntax for HW and for SW
- Widely accepted language (C++)
- Simulator integrated into specification
- Different abstraction levels
  - RT; Transaction
- Synthesis not yet well defined