System Level Design (and Modelling for Embedded Systems)

Part 2: SystemC

Kim Grüttner <kim.gruettner@dlr.de>
Jörg Walter <joerg.walter@offis.de>
Henning Schlender <henning.schlender@dlr.de>
Sven Mehlhop <sven.mehlhop@offis.de>

Distributed Computation and Communication, R&D Division Manufacturing
OFFIS – Institute for Information Technology

Institute of Systems Engineering for Future Mobility
German Aerospace Center (DLR-SE)

Based on the slides of M. Radetzki 2005–2008
University of Stuttgart

Overview of Part 2

System Level Design (and Modelling for Embedded Systems)

9 – Introduction to SystemC (Part 1)

Kim Grüttner <kim.gruettner@dlr.de>
Jörg Walter <joerg.walter@offis.de>
Henning Schlender <henning.schlender@dlr.de>
Sven Mehlhop <sven.mehlhop@offis.de>

Distributed Computation and Communication, R&D Division Manufacturing
OFFIS – Institute for Information Technology

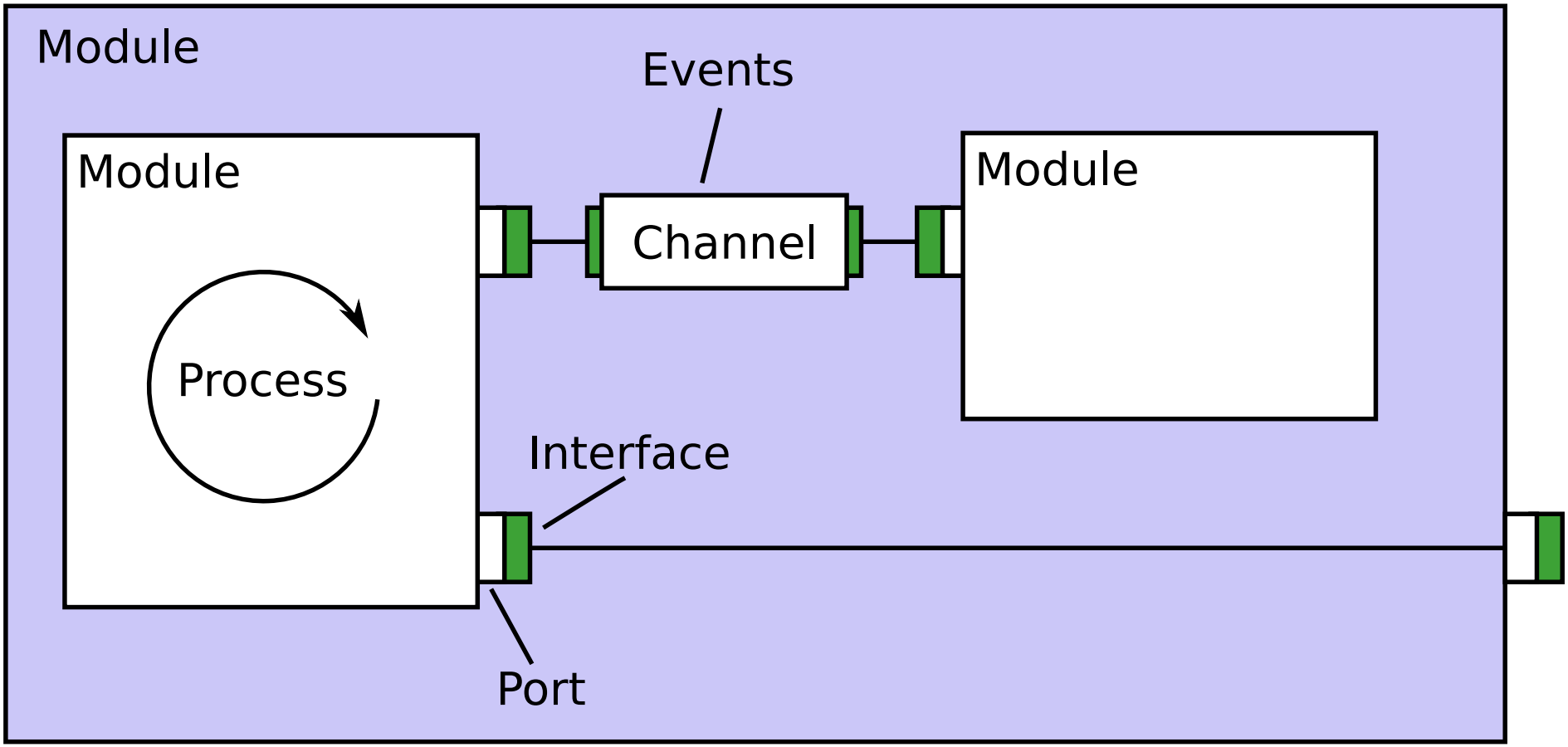
Institute of Systems Engineering for Future Mobility
German Aerospace Center (DLR-SE)

Based on the slides of M. Radetzki 2005–2008
University of Stuttgart

About SystemC

- **Open source C++ class library for system modelling**
- **IEEE standard 1666:2011**
- **Owned by Accellera Systems Initiative**
  - `www.systemc.org` (download, documentation, infos)
- **Supplementary class libraries**
  - TLM: transaction level modelling
  - SCV: functional verification (by simulation)
  - AMS: analog/mixed-signal modelling
  - Third-party libraries, e.g. OSSS, GreenSocs, Synopsys SCML
- **Common use-case: Discrete Event Simulation**
  - Useful for top-down system level design
  - Incremental refinement with continuous validation

Fundamental Concepts



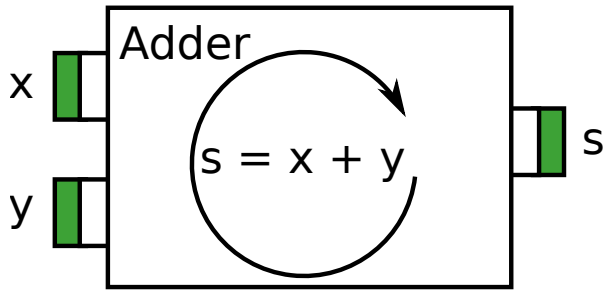
- Structure generated at simulation startup
- (Almost) no reconfiguration during simulation

Model Basics

Modules and Ports

File: adder.h

```
#include <systemc>

SC_MODULE(Adder)
{
  sc_core::sc_in<int> x;
  sc_core::sc_in<int> y;

  sc_core::sc_out<int> s;

  ...

};
```



Module name: **Adder**

Input ports

Output port

Port **data type**

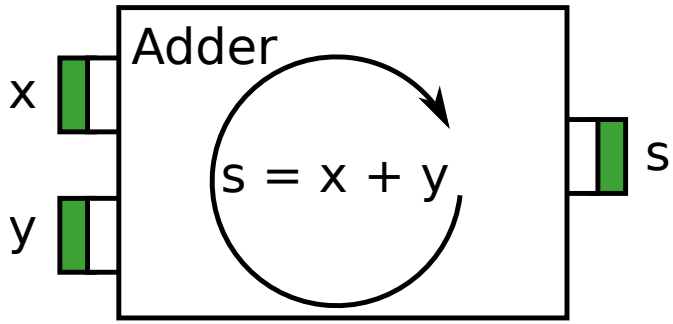
Processes: SC_METHOD, SC_THREAD, SC_CTHREAD

- **Processes are special member functions of modules**
  - Automatically activated by **events**
- **SC_METHOD**
  - **Static** sensitivity (triggered by **predefined** events)
  - Run-to-completion
- **SC_THREAD**
  - **Static and dynamic** sensitivity
  - Execution is suspended by **wait**()-statements
  - Runs from previous **wait**() to next **wait**()
- **SC_CTHREAD (clocked thread)**
  - Specialized **SC_THREAD**
  - Only sensitive to **clock and reset**
  - Good for modeling implicit state machines

SC_METHOD Process Declaration and Implementation

File: adder.h

```
#include <systemc>

SC_MODULE(Adder)
{
  sc_core::sc_in<int> x;
  sc_core::sc_in<int> y;
  sc_core::sc_out<int> s;

  void add();
  SC_CTOR(Adder)
  {
    SC_METHOD(add);
    sensitive << x << y;
  }
};
```

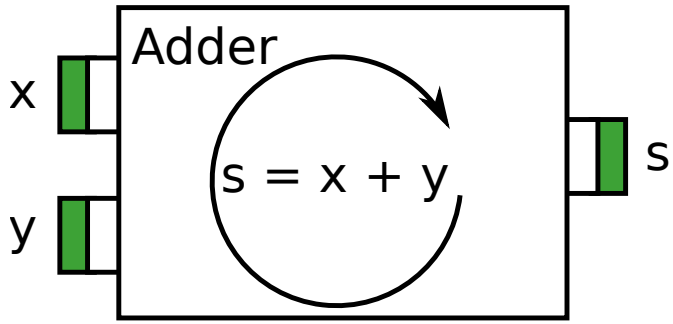


- Function prototype
- Module constructor
- Function registered as a process
- Activation condition of the process: Value change on port x or port y leads to automatic start of add()

SC_METHOD Process Declaration and Implementation

File: adder.h

```
#include <systemc>

SC_MODULE(Adder)
{
  sc_core::sc_in<int> x;
  sc_core::sc_in<int> y;
  sc_core::sc_out<int> s;

  void add();
  SC_CTOR(Adder)
  {
    SC_METHOD(add);
    sensitive << x << y;
  }
};
```



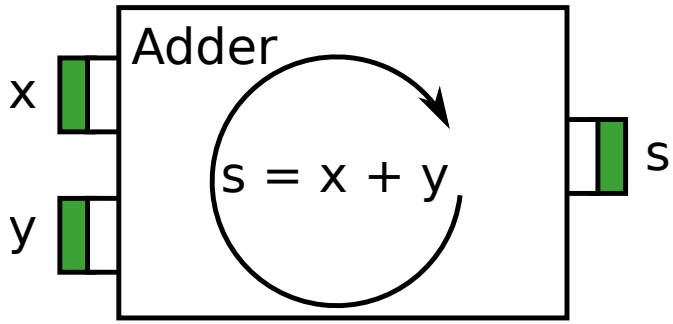
File: adder.cpp

```
#include "adder.h"

void Adder::add()
{
  s = x + y;
  // Alternatively:
  // s.write( x.read() + y.read() );
}
```

SC_THREAD Process Declaration and Implementation

File: adder.h

```
#include <systemc>

SC_MODULE(Adder)
{
  sc_core::sc_in<int> x;
  sc_core::sc_in<int> y;
  sc_core::sc_out<int> s;

  void add();
  SC_CTOR(Adder)
  {
    SC_THREAD(add);
    sensitive << x << y;
  }
};
```



Function prototype

Module constructor

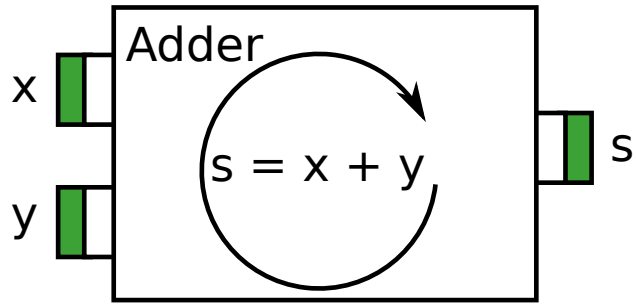
Function registered as a process

Activation condition

SC_THREAD Process Declaration and Implementation

File: adder.cpp

```
#include "adder.h"

void Adder::add()
{
  while (true) {
    wait();

    s = x + y;
  }
}
```



SC_THREAD is started only once, at the beginning of the simulation

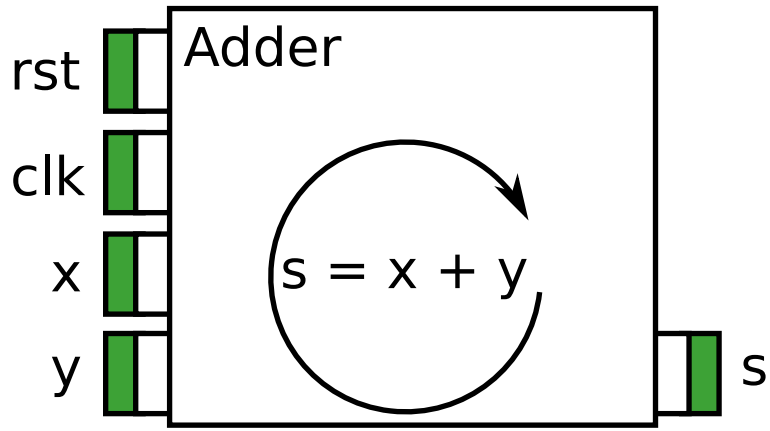
SC_THREAD specifies activation by call to **wait()** function; here: waits for sensitive condition in adder.h: **sensitive** << x << y;

The above **SC_THREAD** implementation has the **same functionality** as the previous **SC_METHOD** implementation.

SC_CTHREAD Process Declaration and Implementation

File: adder.h

```
#include <systemc>

SC_MODULE(Adder)
{
  sc_core::sc_in<bool> clk;
  sc_core::sc_in<bool> rst;
  sc_core::sc_in<int> x, y;
  sc_core::sc_out<int> s;

  void add();
  SC_CTOR(Adder)
  {
    SC_CTHREAD(add, clk.pos());
    reset_signal_is(rst, true);
  }
};
```



Additional clock and reset ports

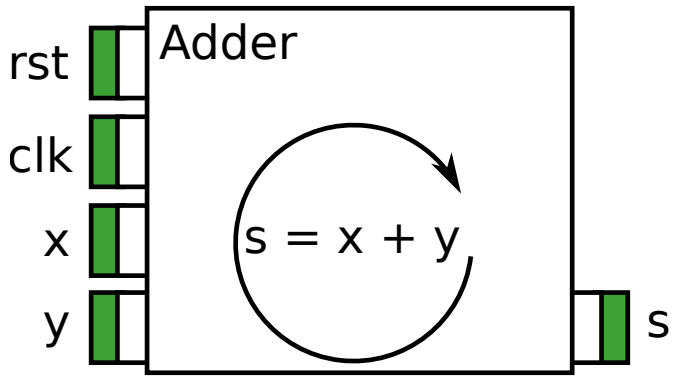
Function registered as a process **sensitive** to rising edge of clk

if `rst` is high, process is **automatically** set back to **start** of function

SC_CTHREAD Process Declaration and Implementation

File: adder.cpp

```
#include "adder.h"

void Adder::add()
{
  s = 0;
  while (true) {
    wait();

    s = x + y;
  }
}
```



Executed at reset, runs until first **wait()**

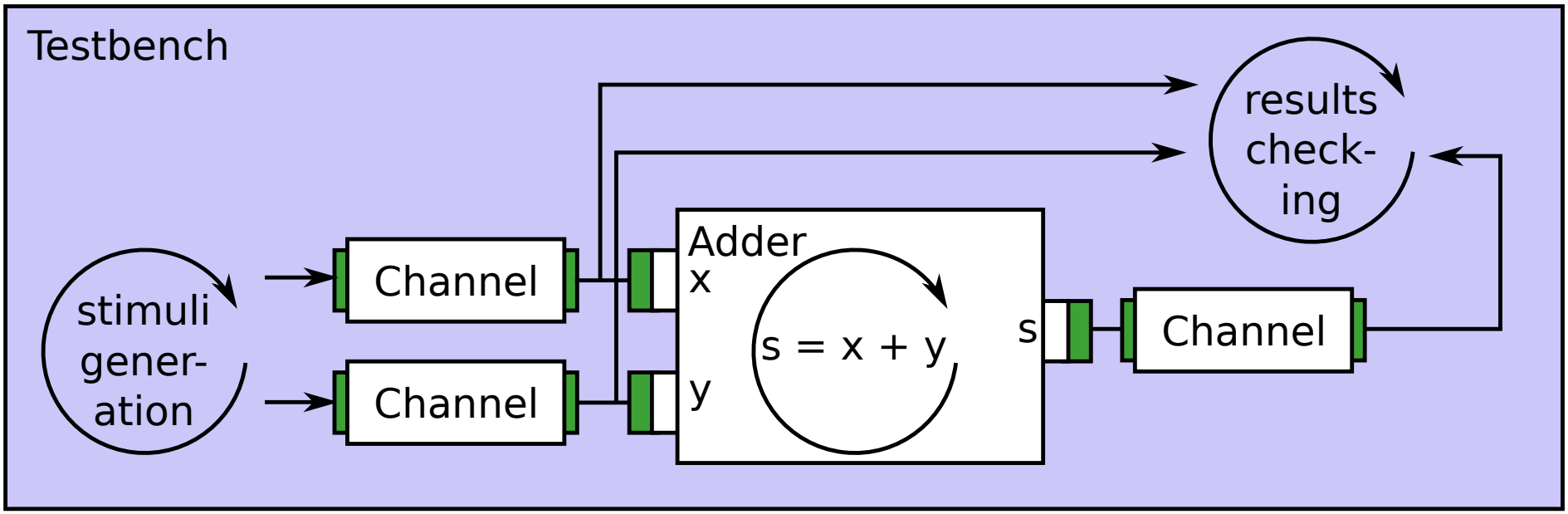
Process waits for next rising clock edge

Same functionality as the previous implementations, but inputs are only evaluated **once** each clock cycle.

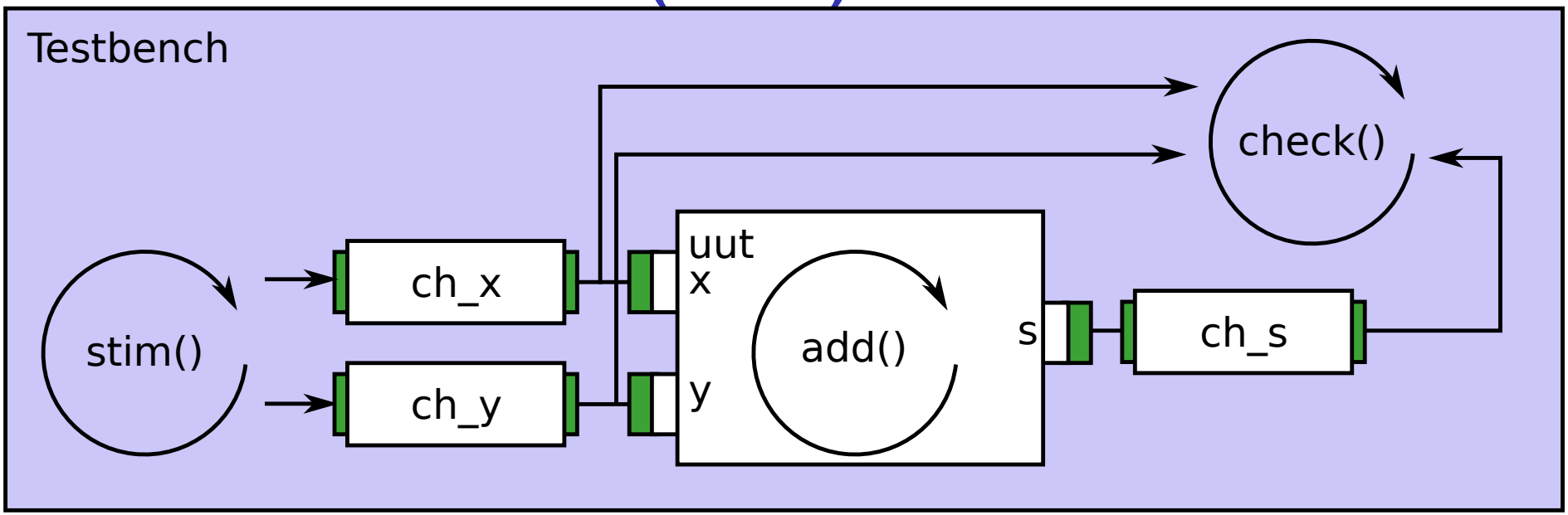
Module Instantiation

- **Building a structural hierarchy of modules**
  - e.g.: ALU contains Adder
- **Defining Connections via channels**

- **Purpose here:**
  - Testing the Adder model by **applying stimuli** to the inputs and checking the computed outputs
  - cf. simulation experiment



Module Instantiation (cont.)



File: `testbench.h`

```
SC_MODULE(Testbench) {
  sc_core::sc_signal<int> ch_x, ch_y, ch_s;
  Adder uut;
  void stim();
  void check();
  ...
```

- Top level module, **no ports**
- Channels
- Adder **instance** uut
- Stimuli **process**
- Checking **process**

Module Constructor

File: testbench.h

```
SC_MODULE(Testbench)
{  // top level; no ports
  sc_core::sc_signal<int> ch_x, ch_y, ch_s;  // channels
  Adder uut;              // Adder instance
  void stim();            // stimuli process
  void check();           // checking process

  SC_CTOR(Testbench)
    : uut("uut")          // initializer list
    , ch_x("ch_x"), ch_y("ch_y"), ch_s("ch_s")
  {
    SC_THREAD(stim);    // without sensitivity
    SC_METHOD(check);
    sensitive << ch_s; // sensitivity for check()
    ...
  }
};
```

Connection of Ports and Channels

File: testbench.h, complete constructor

```
SC_CTOR(Testbench)
  : uut("uut")        // initializer list
  , ch_x("ch_x"), ch_y("ch_y"), ch_s("ch_s")
{
  SC_THREAD(stim);    // without sensitivity
  SC_METHOD(check);
  sensitive << ch_s; // sensitivity for check()

  uut.x(ch_x);  // port x of uut bound to ch_x
  uut.y(ch_y);  // port y of uut bound to ch_y
  uut.s(ch_s);  // port s of uut bound to ch_s
}
```

Syntax for binding ports to channels

<instance name>.<port name>(<channel name>)

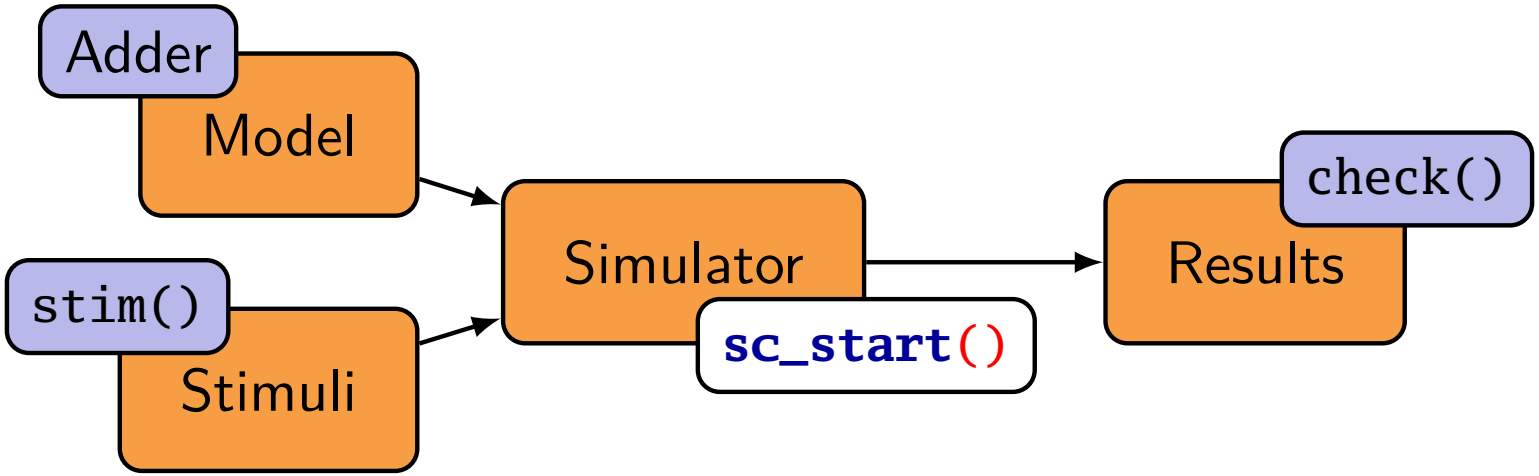
Testbench Implementation

File: testbench.cpp

```
#include "testbench.h"

void Testbench::stim()  // SC_THREAD
{
  ch_x = 3; ch_y = 4;   // first stimulus
  wait(10, SC_NS);      // wait for 10 ns
  ch_x = 7; ch_y = 0;   // second stimulus
  wait(10, SC_NS);      // wait (no sensitivity!)
  ...                   // further stimuli
}

void Testbench::check() // SC_METHOD
{
  std::cout << ch_x << ch_y << ch_s << std::endl; // debug output
  if( ch_s != ch_x + ch_y ) sc_core::sc_stop();   // stop simulation
  else std::cout << "-> OK" << std::endl;
}
```

Running a Simulation

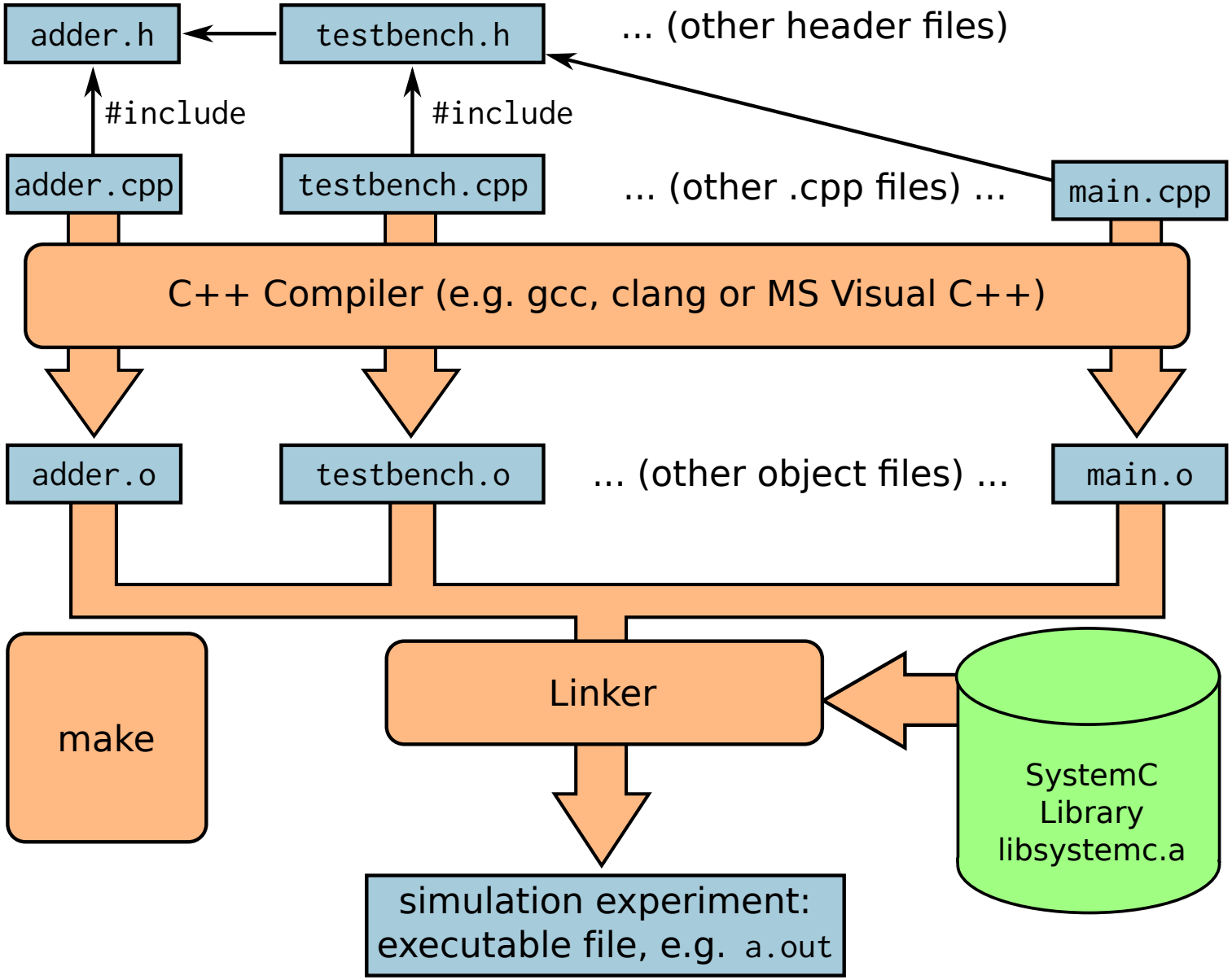
Invoking the Simulation from sc_main



File: main.cpp

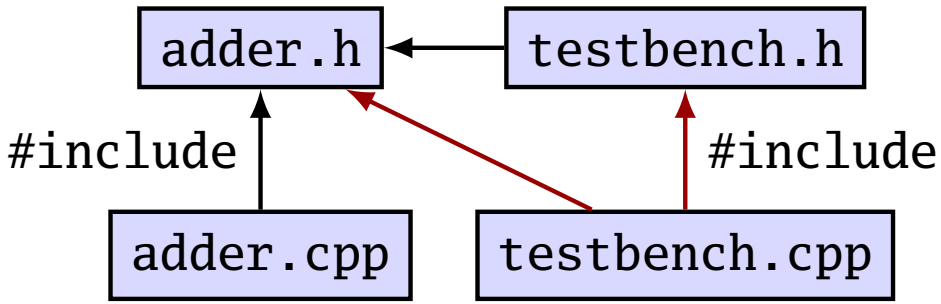
```
#include "testbench.h"
int sc_main(int argc, char *argv[]) // cf. C++ main()
{
  Testbench tb("tb");
  sc_core::sc_start();
  std::cout << "simulation finished" << std::endl;
}
```

Compiling the Source Code Files



Using Sentinels

- to avoid multiple (indirect) inclusions of a header file
- each header file should have a **#ifndef** sentinel



File: adder.h

```
#ifndef ADDER_H_
#define ADDER_H_

... // C++ code for adder

#endif // ADDER_H_
```

File: testbench.h

```
#ifndef TESTBENCH_H_
#define TESTBENCH_H_

... // C++ code

#endif // TESTBENCH_H_
```

Running a Simulation Experiment

1. **Compile:**

```
clang++ -I. -I ~systemc/include -c -o adder.o adder.cpp
clang++ -I. -I ~systemc/include -c -o testbench.o testbench.cpp
clang++ -I. -I ~systemc/include -c -o main.o main.cpp
```

2. **Link:** `clang++ -L. -L ~systemc/lib *.o -lsystemc -lm`
3. **Execute:** `./a.out`
4. **Output:**

```
        SystemC 2.3.1 --- May  15 2014 16:20:29
      Copyright (c) 1996-2014 by all Contributors,
                ALL RIGHTS RESERVED
0+0=0 -> OK
3+4=7 -> OK
8+-4=4 -> OK
simulation finished
```

Stimuli vs. Simulation Results

File: testbench.cpp

```
ch_x = 3; ch_y = 4;
wait(10, SC_NS);
ch_x = 7; ch_y = 0;   (2)
wait(10, SC_NS);
ch_x = 8; ch_y = -4;
wait(10, SC_NS);
ch_x = 8; ch_y = -4;  (3)
wait(10, SC_NS);
```

Output

```
0+0=0   -> OK   (1)
3+4=7   -> OK
8+-4=4  -> OK
simulation finished
```

Activation condition of the process: **Value change** on port x or port y leads to automatic start of add()

From adder.h:

```
sensitive << x << y;
```

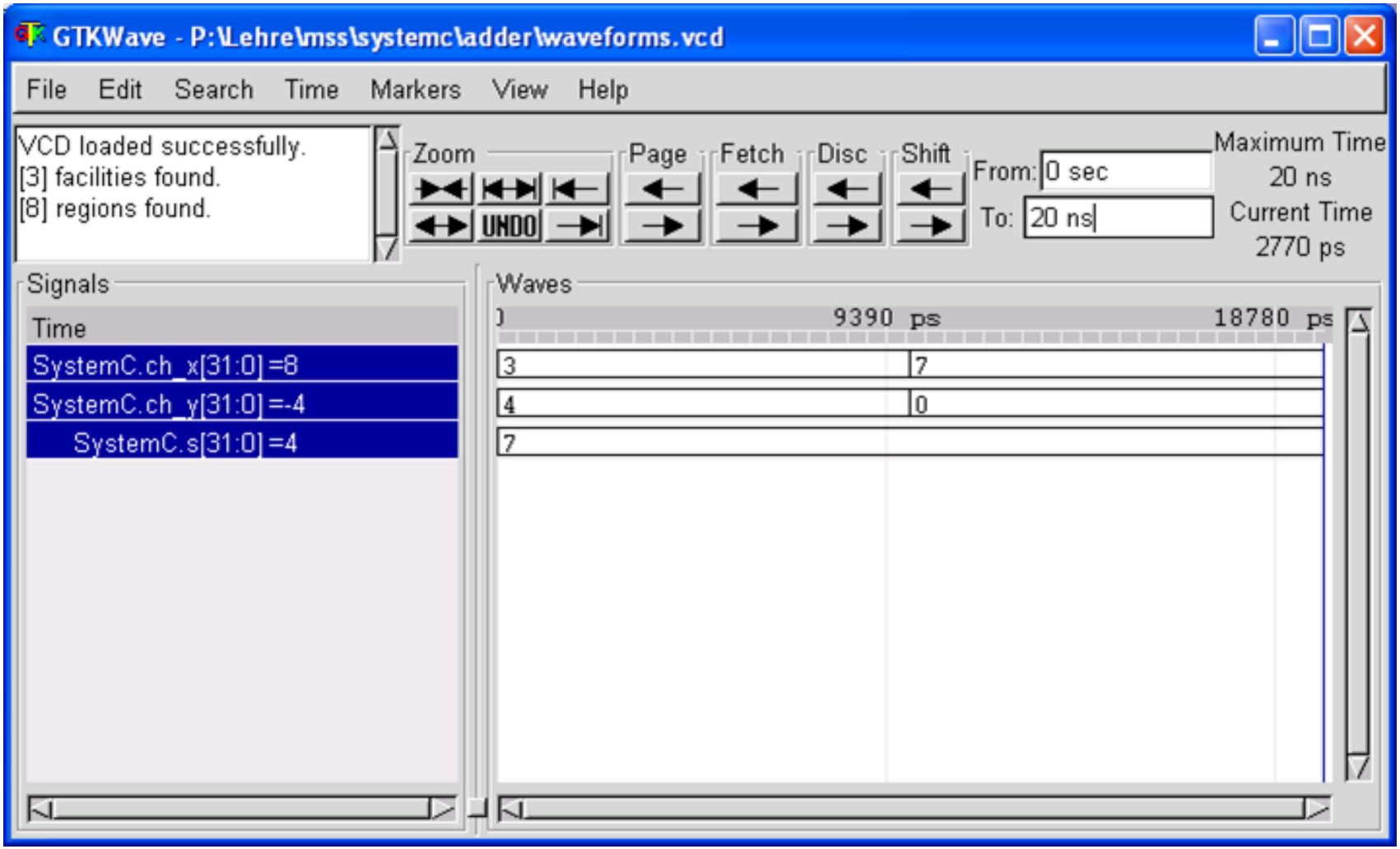
with ports x, y connected to channels ch_x, ch_y.

Waveform Tracing

File: main.cpp

```
#include "testbench.h"
int sc_main(int argc, char *argv[])  // cf. C++ main()
{
  Testbench tb("tb");
  sc_core::sc_trace_file *handle;  // file handle declaration
  handle = sc_core::sc_create_vcd_trace_file("waveforms");
  sc_core::sc_trace(handle, tb.ch_x, "ch_x");
  sc_core::sc_trace(handle, tb.ch_y, "ch_y");
  sc_core::sc_trace(handle, tb.ch_s,   "s");
  sc_core::sc_start();
  std::cout << "simulation finished" << std::endl;
  sc_core::sc_close_vcd_trace_file(handle);
}
```

Waveform Tracing (cont.)



- No value change on ch_s

Stimuli vs. Simulation Results

File: testbench.cpp

```
ch_x = 3; ch_y = 4;
wait(10, SC_NS);
ch_x = 7; ch_y = 0;   (2)
wait(10, SC_NS);
ch_x = 8; ch_y = -4;
wait(10, SC_NS);
ch_x = 8; ch_y = -4;  (3)
wait(10, SC_NS);
```

Output

```
0+0=0   -> OK   (1)
3+4=7   -> OK
8+-4=4  -> OK
simulation finished
```

From adder.h:
sensitive << x << y; with ports x, y connected to channels ch_x, ch_y.

From testbench.h:
sensitive << ch_s;

Activation condition of the process: **value change** on port x or port y leads to automatic start of add()

- (1) later

More on Channels and Processes

sc_buffer Channel

Different Events

```
sensitive << ch; // waits for event on channel ch
sc_core::sc_signal<T> ch; // event: value change
sc_core::sc_buffer<T> ch; // event: any write to channel
```

Change the line `sc_signal<int> ch_x, ch_y, ch_s;`
to `sc_buffer<int> ch_x, ch_y, ch_s;`

Output

```
0+0=0 -> OK
3+4=7 -> OK
7+0=7 -> OK
8+-4=4 -> OK
8+-4=4 -> OK
```

SystemC Built-In Channels

Channel	Matching Ports	Event
sc_signal<T>	**sc_in**<T>, **sc_out**<T>, **sc_inout**<T>	value changed
sc_buffer<T>	**sc_in**<T>, **sc_out**<T>, **sc_inout**<T>	value written (also if same as previous value)
sc_fifo<T>	**sc_fifo_in**<T>, **sc_fifo_out**<T>	fifo contents changed
sc_semaphore<T>	–	–
sc_mutex<T>	–	–
sc_clock<T>	**sc_in**<T>	value changed

sc_fifo Channel

Declaration

```
sc_core::sc_fifo<int> my_fifo(4);
```

Type of data stored in FIFO

Name of the FIFO object

Maximum FIFO size (default: 16)

Blocking Access: read, write do not return until successful

```
my_fifo.write(data); // waits if FIFO is full
data = my_fifo.read(); // waits if FIFO is empty
```

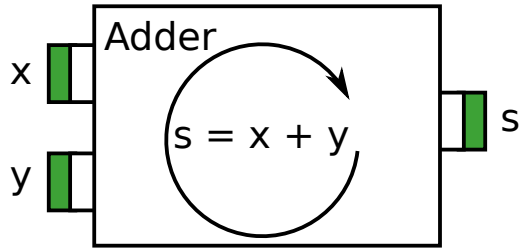
Non-Blocking Access: read, write return immediately

```
bool flag; // returned status flag indicates success
flag = my_fifo.nb_write(data); // false if FIFO is full
flag = my_fifo.nb_read(data); // false if FIFO is empty
```

Adder with FIFOs

File: adder.h

```
#include <systemc>

SC_MODULE(Adder) {
  sc_core::sc_in<bool> clk;
  sc_core::sc_in<bool> rst;
  // Ports connecting to FIFO channels
  sc_core::sc_fifo_in<int> x;
  sc_core::sc_fifo_in<int> y;
  sc_core::sc_fifo_out<int> s;

  void add();
  SC_CTOR(Adder) {
    SC_THREAD(add);
  }
};
```



File: adder.cpp

```
#include "adder.h"

void Adder::add()
{
  int a, b;
  while (true) {
    a = x.read(); b = y.read();
    s.write(a + b); std::cout << ...
  }
}
```

FIFO Testbench

File: testbench.h

```
SC_MODULE(Testbench)
{  // top level; no ports
  sc_core::sc_fifo<int> ch_x, ch_y, ch_s;  // channels
  Adder uut;              // Adder instance
  void stim();            // stimuli process
  void check();           // checking process

  SC_CTOR(Testbench)
  : uut("uut") // initializer list
  , ch_x(4), ch_y(4), ch_s(3)
  {
    SC_THREAD(stim);   // without sensitivity
    SC_THREAD(check);
    uut.x(ch_x); uut.y(ch_y); uut.s(ch_s);
  }
};
```

FIFO Testbench Implementation

File: testbench.cpp

```
#include "testbench.h"

void Testbench::stim()  // SC_THREAD
{
  for (int i = 0; i < 10; ++i) {
    ch_x.write(i); ch_y.write(30-2*i);
    std::cout << "stim x = " << i << ", y = " << 30-2*i << std::endl;
  }
}

void Testbench::check() // SC_THREAD
{
  while (true) {
    // Note: Cannot read values twice from ch_x, ch_y
    std::cout << "  check: x + y = " << ch_s.read() << std::endl;
    wait(10, sc_core::SC_NS);
  }
}
```

Simulation Results

```
stim x = 0, y = 30
stim x = 1, y = 28
stim x = 2, y = 26
stim x = 3, y = 24
add: computed & wrote 0+30=30
add: computed & wrote 1+28=29
add: computed & wrote 2+26=28
stim x = 4, y = 22
stim x = 5, y = 20
stim x = 6, y = 18
check: received x + y = 30
add: computed & wrote 3+24=27
stim x = 7, y = 16
check: received x + y = 29
add: computed & wrote 4+22=26
check: received x + y = 28
add: computed & wrote 5+20=25
stim x = 9, y = 12
check: received x + y = 27
add: computed & wrote 6+18=24
check: received x + y = 26
add: computed & wrote 7+16=23
check: received x + y = 25
add: computed & wrote 8+14=22
check: received x + y = 24
add: computed & wrote 9+12=21
check: received x + y = 23
check: received x + y = 22
check: received x + y = 21
simulation finished
```

SC_METHOD Adder Process

File: adder.h

```
#include <systemc>

SC_MODULE(Adder) {
  sc_core::sc_fifo_in<int> x;
  sc_core::sc_fifo_in<int> y;
  sc_core::sc_fifo_out<int> s;

  void add();
  SC_CTOR(Adder)
  {
    SC_METHOD(add);
    sensitive
      << x.data_written()
      << y.data_written();
  }
};
```

File: adder.cpp

```
#include "adder.h"

void Adder::add()
{
  int a, b;
  // while (true) {
    a = x.read(); b = y.read();
    s.write(a + b); std::cout << ...
  // }
}
```

But: blocking `read()` calls `wait()`!

Simulation error:
(E519) wait() is not
allowed in SC_METHODs.

SC_METHOD Adder Process

File: adder.h

```
#include <systemc>

SC_MODULE(Adder) {
  sc_core::sc_fifo_in<int> x;
  sc_core::sc_fifo_in<int> y;
  sc_core::sc_fifo_out<int> s;

  void add();
  SC_CTOR(Adder)
  {
    SC_METHOD(add);
    sensitive
      << x.data_written()
      << y.data_written();
  }
};
```

File: adder.cpp

```
#include "adder.h"

void Adder::add()
{
  int a, b;
  bool a_ok, b_ok, s_ok;

  a_ok = x.nb_read(a);
  b_ok = y.nb_read(b);

  if (a_ok && b_ok) {
    s_ok = s.nb_write(a + b);
    if (s_ok) std::cout << ...
    else ; // ??
  }
}
```

Simulation Results

```
stim x = 0, y = 30
stim x = 1, y = 28
stim x = 2, y = 26
stim x = 3, y = 24
add: computed & wrote 0+30=30
stim x = 4, y = 22
check: received x + y = 30
add: computed & wrote 1+28=29
stim x = 5, y = 20
add: computed & wrote 2+26=28
stim x = 6, y = 18
add: computed & wrote 3+24=27
stim x = 7, y = 16
stim x = 8, y = 14
stim x = 9, y = 12
check: received x + y = 29
check: received x + y = 28
check: received x + y = 27
simulation finished
```

- **Simulation no longer aborts**
- **Data is lost**
- When using non-blocking calls, the user must **take care of synchronization** in order to **avoid data loss** (cf. shared memory)
- In example, need to
  - **Observe OK flags** more seriously
  - **Re-write data later** in case `nb_write()` fails
  - **Avoid scrapping** read data

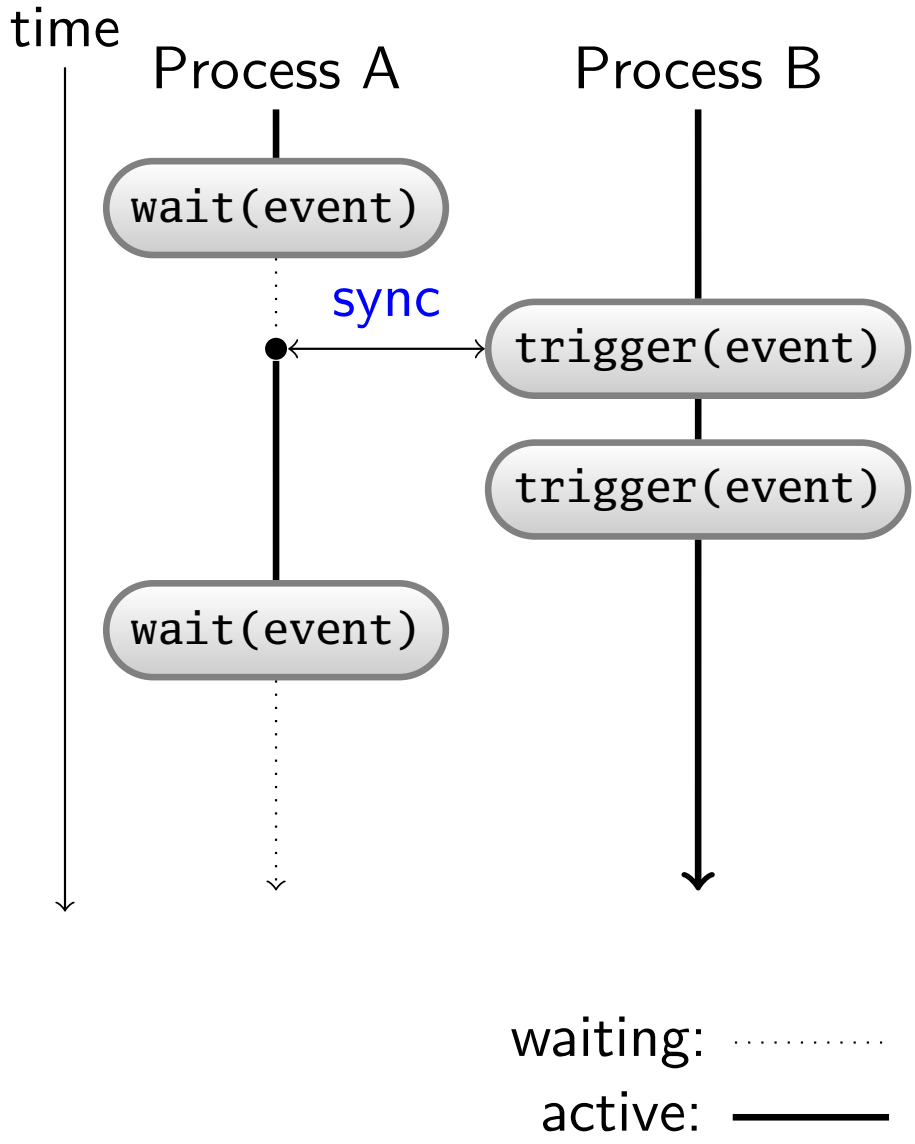
SC_METHOD vs. SC_(C)THREAD Summary

- **SC_METHOD**
  - Started again whenever **activation condition** triggers
  - Must **not** call `wait()`
  - Must **not** block
  - Must **not** call functions that **block** or **call** `wait()`
  - Must **not** contain **infinite loop** (blocks all other processes)
  - May use non-blocking communications **only**
- **SC_THREAD**
  - Started only **once**, at beginning of simulation
  - May (and must) call `wait()`
  - Often contains **infinite loop**
  - May (and must) **block** – allow other processes to execute
  - May use both **non-blocking and blocking** communications
- **SC_CTHREAD**
  - Can only be sensitive to clock and reset, **no dynamic sensitivity**
  - Should **only** call non-blocking interface methods

Events and Timing

Synchronization via Events

- Process can **wait for an event**
- Event can be generated (**triggered**) by another process
- Events are **forgotten** after being triggered
- **Wait operation** must have been invoked **before the trigger** of an event in order **to "receive"** that event



User-Defined Events in SystemC

- Declaration: `sc_core::sc_event name;`
- Immediate trigger: `name.notify();`
- Waiting for occurence: `wait(name);`

Stimuli Process

```
int x; int y;
sc_core::sc_event new_stimulus;
void Testbench::stim()
{
  x = 3; y = 4;
  new_stimulus.notify();
  x = 7; y = 0;
  new_stimulus.notify();
  // stimulus 7, 0 again
  new_stimulus.notify();
  ...
}
```

Checking Process

```
void Testbench::check()
{
  for(;;)
  {
    wait(new_stimulus);
    if (s == x + y)
      std::cout << "OK" << ...;
    else
      std::cout << "ERROR" << ...;
  }
}
```

Time in SystemC

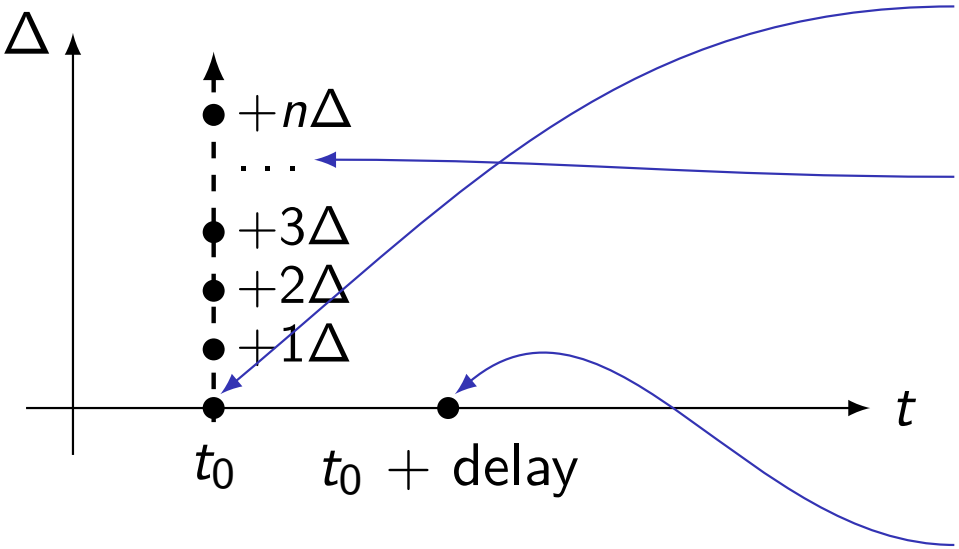
`sc_core::sc_time` is a data type that consists of a **magnitude** and a **unit**.

- **Time units**

Unit	Name	
`sc_core::SC_FS`	femtosecond	1×10^{-15}
`sc_core::SC_PS`	picosecond	1×10^{-12}
`sc_core::SC_NS`	nanosecond	1×10^{-9}
`sc_core::SC_US`	microsecond	1×10^{-6}
`sc_core::SC_MS`	millisecond	1×10^{-3}
`sc_core::SC_SEC`	second	1

- Time object `sc_core::sc_time name(<magnitude>, <unit>);`
  e.g.: `sc_core::sc_time delay(10, sc_core::SC_NS);`
- Usage, e.g.: `wait(delay);`
  - Alternatively: `wait(10, sc_core::SC_NS);`

Timed Events



Immediate notification

- `event.notify();`

Delta notification

- `event.notify(sc_core::SC_ZERO_TIME);`
- `event.notify(0, sc_core::SC_NS);`

Timed notification

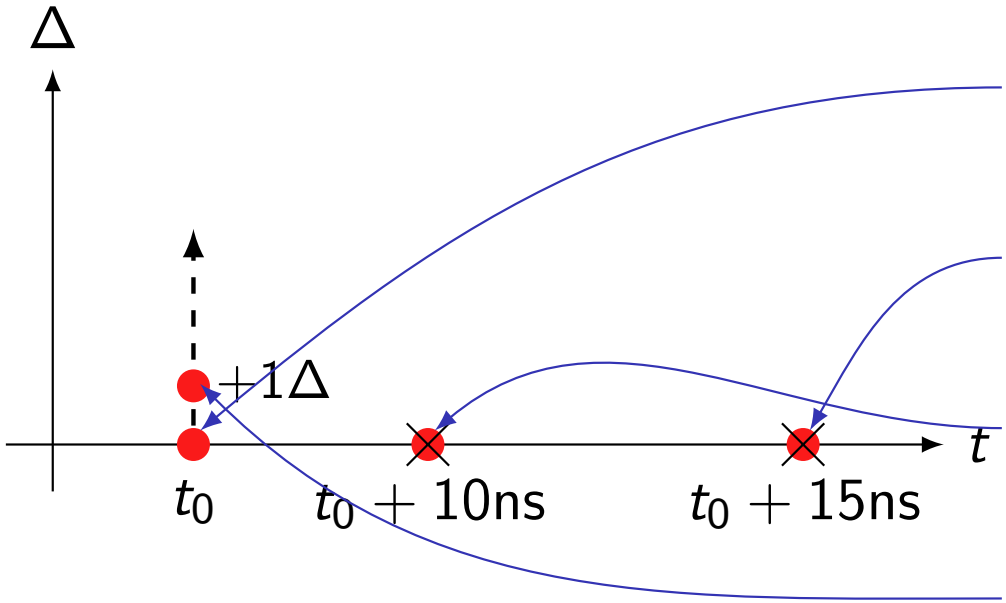
- `event.notify(delay);`
- `event.notify(10, sc_core::SC_NS);`

Cancelling a notification

- `event.cancel();`

Multiple Notifications

- There can be **at most one** pending notification per event
- In case of multiple notifications, the **earlier** one wins




```
event.notify();
```

- **Immediate**, not pending

```
event.notify(15, sc_core::SC_NS);
```

- **Pending**

```
event.notify(10, sc_core::SC_NS);
```

- **Replaces** previous notification

```
event.notify(sc_core::SC_ZERO_TIME);
```

- **Replaces** previous notification

```
event.notify(5, sc_core::SC_NS);
```

- **Ignored**

Waiting on Events

```
sc_core::sc_event a, b, c;
sc_core::sc_time t(...);

wait();

wait( a );

wait( a & b & c );
wait( a | b | c );

wait( t );
wait( t, a & b );
```

Static sensitivity

- `sensitive << ...`

Dynamic sensitivity

- On a single event
- On a combination of events
  - **All** events have happened
  - **At least one** event has happened
- For a **time** period
- **Timeout** (wait no longer than t)

Dynamic Sensitivity for SC_METHOD

```
sc_core::sc_event a, b, c;
sc_core::sc_time t(...);

next_trigger();

next_trigger( a );

next_trigger( a & b & c );
next_trigger( a | b | c );

next_trigger( t );
next_trigger( t, a & b );
```

Switch back to **static sensitivity**

`sensitive << ...`

Dynamic sensitivity for **next activation**

- On a single event
- On a combination of events
  - **All** events have happened
  - **At least one** event has happened
- For a **time** period
- **Timeout** (wait no longer than t)

sc_core::sc_clock Channel

sc_core::sc_clock Declaration

```
sc_core::sc_clock clk("clk", period,
  0.5, delay);
```

Name of the **clock object**

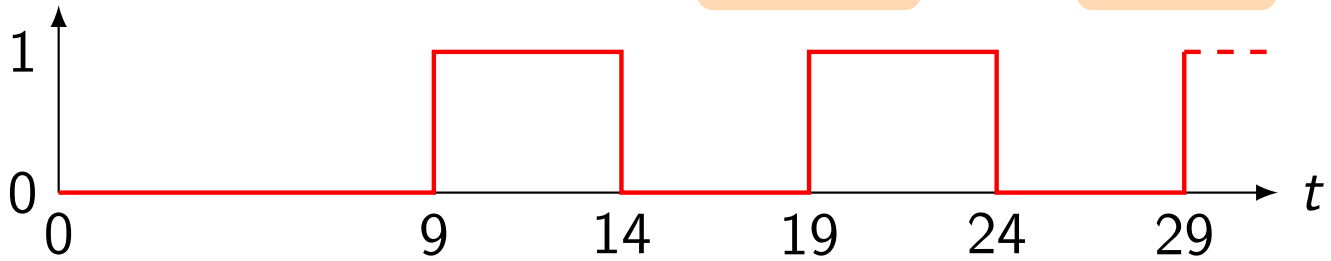
Name as **string**

Clock cycle **duration**, default 1ns

Duty cycle: percentage of time during which the clock has value 1, default: 0.5 (= 50%)

Time of **first clock edge**, default 0ns

```
sc_core::sc_clock clk("clk", 10, SC_NS, 0.5, 9, SC_NS);
```



Events from Built-In Channels

- Channels **notify** special event, e.g. when their values changes
- These events can be **obtained** via methods of the channel

Usage	Properties	
Static sensitivity, **ports**	Method:	`value_changed()`
	Returns:	`sc_event_finder`
	$0 \rightarrow 1$ event:	`pos()`
	$1 \rightarrow 0$ event:	`neg()`
`wait(...)`	Method:	`value_changed_event()`
	Returns:	**`sc_event`**
	$0 \rightarrow 1$ event:	`posedge_event()`
	$1 \rightarrow 0$ event:	`negedge_event()`
Condition `if(...)`	Method:	`event()`
	Returns:	**`bool`**
	$0 \rightarrow 1$ event:	`posedge()`
	$1 \rightarrow 0$ event:	`negedge()`

Example: Sensitivity on Channel Events

Register Module

```
SC_MODULE(Register)
{
  sc_core::sc_in<bool> clk;
  sc_core::sc_in<bool> reset;
  sc_core::sc_in<int>  d;
  sc_core::sc_out<int> q;

  void proc();
  SC_CTOR(Register)
  {
    SC_METHOD(proc);
    sensitive
      << clk.pos() << reset;
  }
};
```

Process Impl.

```
void Register::proc()
{
  if ( reset.negedge() ) {
    q = 0;
  } else if ( clk.event() ) {
    q = d;
  }
}
```

Checking if event has happened

What does this process do, precisely? Is this really what we want?

Example: Waiting on Channel Events

Register Module

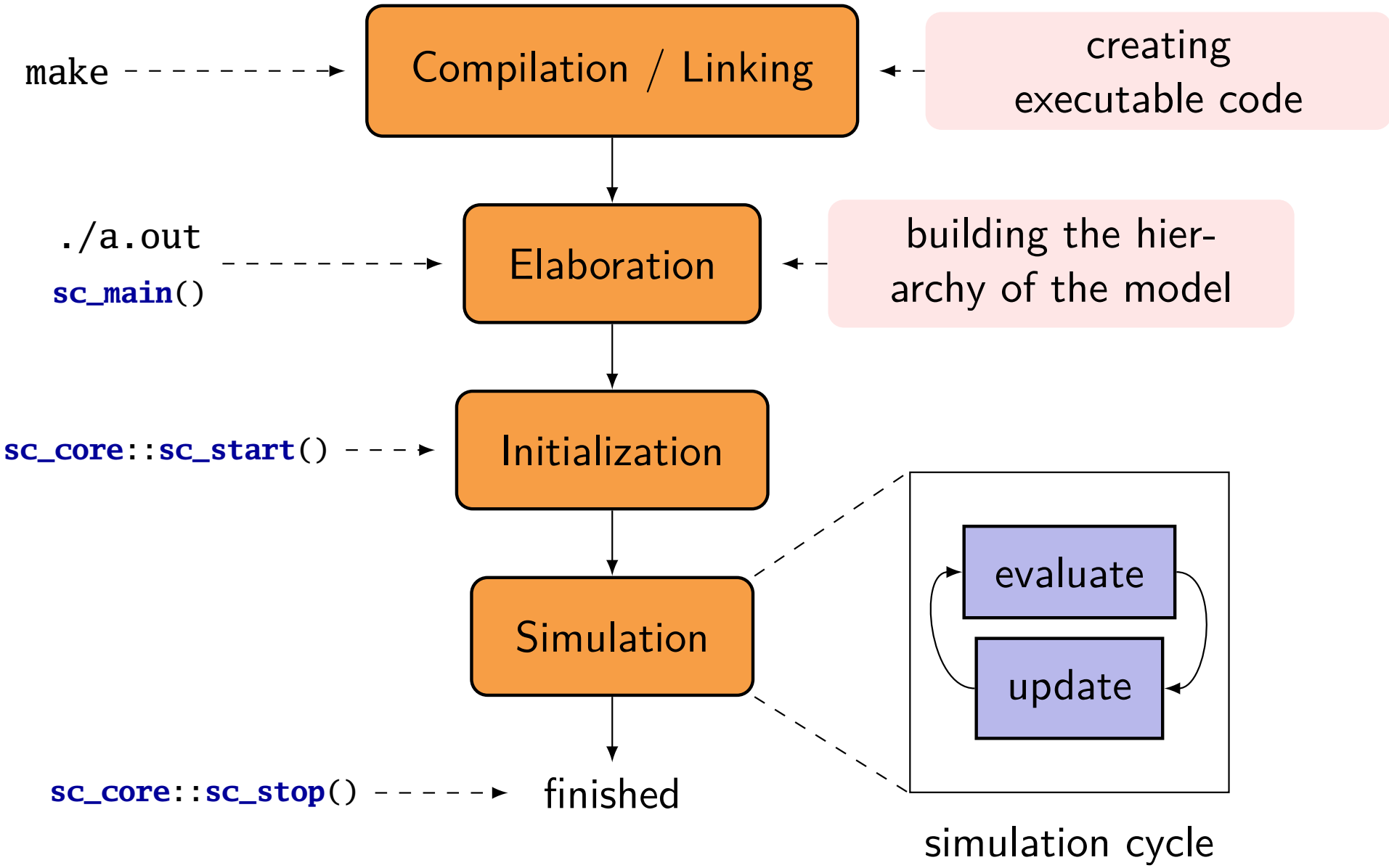
```
SC_MODULE(Register)
{
  sc_core::sc_in<bool> clk;
  sc_core::sc_in<bool> reset;
  sc_core::sc_in<int>  d;
  sc_core::sc_out<int> q;

  void proc();
  SC_CTOR(Register)
  {
    SC_THREAD(proc);
  }
};
```

Process Impl.

```
void Register::proc()
{
  for(;;) {
    wait( clk.posedge_event() |
          reset.value_changed_event());

    if( reset == false ) {
      q = 0;
    } else if( clk.posedge() ) {
      q = d;
    }
  }
}
```

Simulation Mechanism

SystemC Phases



Elaboration

File: main.cpp

```
int sc_main(int, char*[]) {
  Testbench tb("tb");
  sc_core::sc_start();
  return 0;
}
```

calling of module constructors

File: testbench.h

```
SC_MODULE(Testbench) {
  Adder uut;
  SC_CTOR(Testbench)
  : uut("uut"), ch_x("ch_x") /* ... */ {
    SC_THREAD(stim);
    SC_METHOD(check);
    sensitive << ch_s;
    uut.x(ch_x); // ...
  }
};
```

connection of ports and channels

File: adder.h

```
#include <systemc>

SC_MODULE(Adder) {
  sc_core::sc_in<int> x;
  sc_core::sc_in<int> y;
  sc_core::sc_in<int> s;

  void add();
  SC_CTOR(Adder) {
    SC_METHOD(add);
    sensitive << x << y;
  }
};
```

registration of channels and ports

registration of processes

Initialization

- After `sc_core::sc_start()`, simulation time is set to 0
- Every **process** (except `SC_CTHREAD`) is **executed once**
  - `SC_CTHREAD` is triggered on first clock edge only
- This generates **initial events** to get simulation started

`stim()` Process

```
void Testbench::stim() {
  ch_x = 3; ch_y = 4;
  wait(10, sc_core::SC_NS);
  ch_x = 7; ch_y = 0;
  wait(10, sc_core::SC_NS);
  ch_x = 8; ch_y = -4;
  wait(10, sc_core::SC_NS);
  ch_x = 8; ch_y = -4;
  wait(10, sc_core::SC_NS);
}
```

```
0 + 0 = 0 -> OK
3 + 4 = 7 -> OK
8 + -4 = 4 -> OK
simulation finished
```

`check()` Process

```
void Testbench::check() { // SC_METHOD
  std::cout << ch_x << ch_y
            << ch_s << std::endl;
  if (ch_s != ch_x + ch_y )
      sc_core::sc_stop();
  else std::cout << "-> OK" << std::endl;
}
```

!1

Avoiding Initialization

Call to `dont_initialize()` prevents initialization.

Testbench Constructor

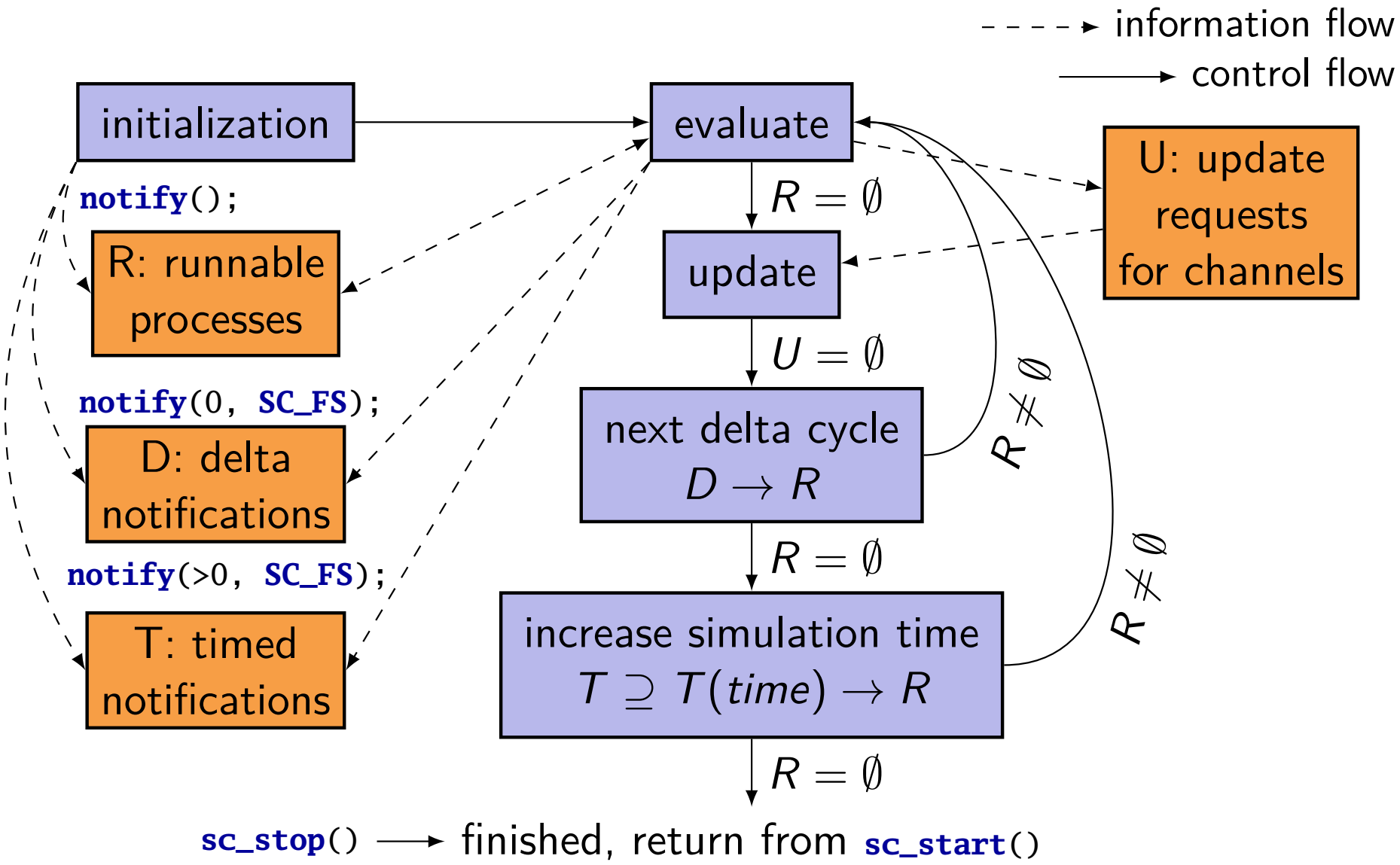
```
SC_MODULE(Testbench) {
  // ...
  SC_CTOR(Testbench)
  : uut("uut"), ch_x("ch_x")
  , ch_y("ch_y"), ch_s("ch_s")
  {
    SC_THREAD(stim);
    SC_METHOD(check);
    sensitive << ch_s;
    dont_initialize();

    uut.x(ch_x);
    uut.y(ch_y);
    uut.s(ch_s);
  }
};
```

```
3 + 4 = 7 -> OK
8 + -4 = 4 -> OK
simulation finished
```

check() Process

```
void Testbench::check() { // SC_METHOD
  std::cout << ch_x << ch_y
            << ch_s << std::endl;
  if (ch_s != ch_x + ch_y )
      sc_core::sc_stop();
  else std::cout << "-> OK" << std::endl;
}
```

Simulation Cycle / Event-Driven Simulation



Evaluate/Update Mechanism

- Update of channel values takes place **after evaluation** of all runnable processes
- Updated values are **not visible immediately**

```
sc_core::sc_signal<int> s;
SC_THREAD(compute);

void compute() { // initial value s = 0
  s = 15;
  s = s + 1;
  std::cout << s << std::endl;
  wait(sc_core::SC_ZERO_TIME);
  std::cout << s << std::endl;
  s = s + 1;
  std::cout << s << std::endl;
  wait(10, sc_core::SC_NS);
  std::cout << s << std::endl;
}
```

Simulation output

```
0
1
1
2
```

Process Evaluation Order

- The order in which parallel processes are evaluated may influence the simulation result if **communication is via shared C++ variables**

```
int a = 5;

SC_METHOD( assign_a );
  sensitive << clk;

void assign_a()
{
  a = b;
}
```

```
int b = 7;

SC_METHOD( assign_b );
  sensitive << clk;

void assign_b()
{
  b = a;
}
```

```
std::cout << "a="<< a << "b="<< b << std::endl;
```

- `assign_a()` evaluated before `assign_b()`: Output a=7 b=7
- `assign_b()` evaluated before `assign_a()`: Output a=5 b=5

Evaluation Order Independence

- The order in which parallel processes are evaluated does not affect the simulation result if **communication is via channels**

```
sc_core::sc_signal<int> a;
a.write(5);
SC_METHOD( assign_a );
  sensitive << clk;

void assign_a()
{
  a.write(b);
}
```

```
sc_core::sc_signal<int> b;
b.write(7);
SC_METHOD( assign_b );
  sensitive << clk;

void assign_b()
{
  b.write(a);
}
```

After at least one delta cycle:

```
std::cout << "a="<< a << "b="<< b << std::endl;
```

- `assign_a()` evaluated before `assign_b()`: Output a=7 b=5
- `assign_b()` evaluated before `assign_a()`: Output a=7 b=5